n° 7987

UNIVERSITÉ DE PARIS-SUD
U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir le grade de

DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI, ORSAY

discipline : informatique

par

Renaud Blanch

Architecture logicielle et outils pour les interfaces hommes-machines graphiques avancées

Soutenue le lundi 26 septembre 2005 devant le jury composé de :

M. JACQUEMIN Christian	président
M. BEAUDOUIN-LAFON Michel	directeur
M^{me} COUTAZ Joëlle	rapporteur
M. PALANQUE Philippe	rapporteur
M. LECOLINET Éric	examinateur

Remerciements

Je remercie en premier lieu Christian Jacquemin qui a accepté avec gentillesse de présider mon jury de thèse.

Je remercie Michel Beaudouin-Lafon, mon directeur de thèse et directeur du Laboratoire de Recherche en Informatique. C'est lui qui, par la qualité des enseignements qu'il m'a dispensé en DEA, m'a donné l'envie de me lancer dans l'aventure de la thèse que ce mémoire rend maintenant tangible. Son écoute, sa patience et sa disponibilité dans les moments de doutes m'ont permis d'arriver au terme de ce voyage.

Je remercie Joëlle Coutaz et Philippe Palanque qui ont accepté d'être les rapporteurs de mon travail. Je remercie Éric Lecolinet qui a accepté de faire partie de mon jury. Ces personnes représentent pour moi des modèles pour la qualité de leurs travaux.

Il en est de même d'Yves Guiard qui est à l'origine des idées développées ensuite pour le pointage sémantique. Une partie de mon travail est le fruit de nos discussions qui ont toujours été enrichissantes et stimulantes.

Je remercie Marie-Claude Gaudel, directrice de l'équipe Programmation et Génie Logiciel. Sa gentillesse et sa rigueur intellectuelle sont pour moi des qualités inestimables.

Je remercie Wendy Mackay, responsable du projet In Situ, pour la diversité des perspectives scientifiques et culturelles qu'elle nous apporte. Merci aussi à tous les membres de l'équipe, ils ont contribué à divers degrés à ce travail et à rendre le cadre de sa réalisation agréable.

Je dois beaucoup à mes amis, à ceux qui ont traversé avec moi ces années, à ceux dont j'ai croisé la route, perdus de vue mais toujours présents au fond du cœur, à ceux qui sont toujours là : Mathieu, Vianney, Gladys, Stéphanie, Xavier, Damien, Florence, Samuel, Grégoire, Alexandre, Guillaume, Lise, Ludovic, Antoine, Jacques, Audrey, Denis, Omar, Stéphane, Caroline, Nicolas, Patrick, Renaud, Agnès, Rodolphe, Fabien, Léonie. C'est ici l'occasion de leur dire combien ils comptent pour moi.

C'est enfin à mes parents et à mon frère que vont mes pensées les plus reconnaissantes. Ils ont toujours été présents à mes côtés dans les moments difficiles, et cette présence est toujours réconfortante.

Paris, le 6 octobre 2005

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Problématique

Ce travail de thèse propose une approche et des outils pour faciliter le développement d'applications interactives. Il est reconnu que pour de telles applications, dont l'exécution implique directement des utilisateurs humains, l'effort de développement le plus important est concentré dans la réalisation de l'interface elle-même [Myers et Rosson, 1992]. Ce constat est le même que l'on s'intéresse aux applications dédiées au grand public, comme celles qui ont trait à la gestion de données personnelles (agenda, photos, musique), ou à celles dédiées à des usages professionnels comme par exemple le contrôle aérien. Cet effort consacré à l'interface est traditionnellement allégé par l'utilisation de boîtes à outils logicielles qui permettent une factorisation et une réutilisation de code existant, principes fondamentaux du génie logiciel. Ces boîtes à outils facilitent le développement en fournissant aux programmeurs des abstractions, robustes et éprouvées, d'un niveau supérieur à celui du système, les plus proches possibles du vocabulaire de l'interaction.

1.1.1 Prêt-à-porter . . .

Cependant, cette approche comporte au moins un travers : elle tend à enfermer le programmeur dans un vocabulaire d'interaction prédéfini et difficilement extensible. La conception des éléments qui composent l'interface est faite non par le programmeur de l'application mais par les concepteurs de la boîte à outils qu'il utilise. Ces derniers ne pouvant sûrement pas connaître les usages auxquels seront destinées les applications développées à l'aide de leur boîte à outils, leurs choix doivent être les plus génériques possible.

C'est ainsi que la plupart des applications courantes obéissent à un paradigme dit WIMP (pour *Window, Icon, Menu* et *Pointing device* — fenêtre, icone,

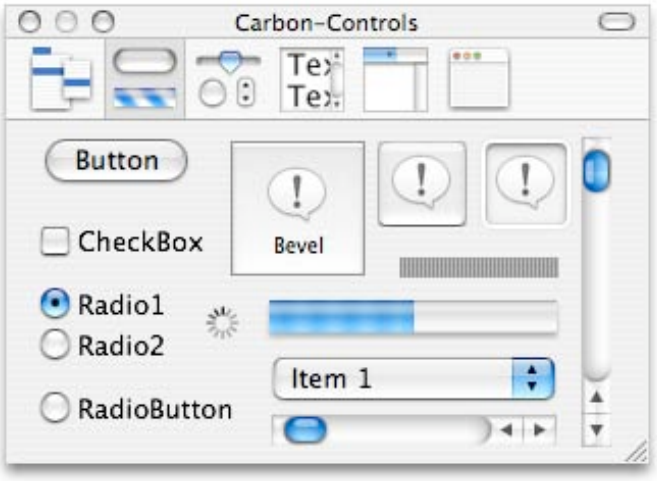


FIG. 1.1 – Exemples d'interacteurs standards de la boîte à outils Carbon de Mac OS X.

menu et dispositif de pointage) en manipulant les données de l'application grâce à un jeu prédéfini d'interacteurs à peu près standard d'une boîte à outils de construction d'interface à l'autre (la Figure 1.1 en présente un sous-ensemble). Ce jeu assez restreint comporte des éléments génériques permettant de déclencher des actions (boutons, menus), des éléments permettant de choisir un ou plusieurs objets parmi un ensemble donné (listes déroulantes ou arbres) ou encore permettant de spécifier des valeurs (champs de saisie de texte ou de valeurs formatées, cases à cocher).

Ces interacteurs permettent de construire des applications aux interfaces "prêtes-à-porter", faciles à produire, stéréotypées, et auxquelles les utilisateurs se sont finalement habitués. Ces interfaces réduisent souvent l'interaction à des actions élémentaires : spécifier la valeur d'une variable, ou déclencher une fonction du programme.

La Figure 1.2 à gauche illustre ce type d'interface pour la sélection d'une couleur à affecter à un objet graphique dans un logiciel de création de présentations électroniques. Cette sélection s'effectue par l'intermédiaire d'une boîte de dialogue exprimant les propriétés graphiques de l'objet manipulé. Ce modèle d'interaction purement WIMP, utilisé pratiquement unanimement pour spécifier la couleur ou les attributs pourtant graphiques d'un objet, nécessite au minimum quatre actions de la part de l'utilisateur : sélection de l'objet, de l'action qu'il veut effectuer conduisant à l'ouverture de la boîte de dialogue pertinente, sélection de la couleur dans la palette proposée, et enfin validation de son choix. Pour peu que l'utilisateur doive changer d'onglet dans la boîte de dialogue, ou que la couleur voulue ne soit pas dans la palette proposée, le nombre d'actions à effectuer pour cette simple modification de l'objet peut pratiquement doubler.

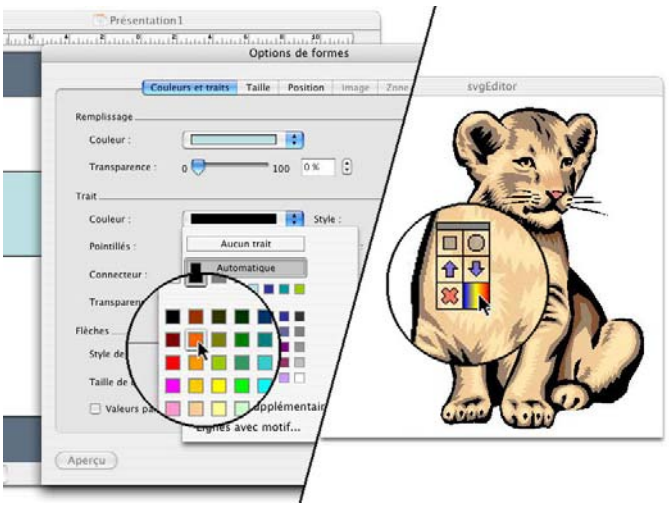


FIG. 1.2 – Interactions WIMP & post-WIMP

1.1.2 . . . ou haute couture ?

Des alternatives utilisant des interactions plus adaptées existent pourtant : la Figure 1.2 montre à droite la même modification d'un attribut graphique d'un objet par la sélection d'une couleur au travers d'une palette semi-transparente manipulée par la main non-dominante, à la manière de la palette d'un peintre. Certes cette technique nécessite l'usage des deux mains et donc requiert la présence de deux dispositifs de pointage, ce qui n'est pas encore courant dans le monde de l'informatique. Cependant il a été montré que ce type de technique, comme d'autres techniques d'interaction récentes ne reposant pas sur des interacteurs standards, est plus efficace et plus utilisable qu'une boîte de dialogue "standard". Et les configurations comportant des périphériques d'entrée multiples ne sont en fait pas si rares : il est courant d'utiliser une souris supplémentaire sur un ordinateur portable. Si le système permettait de dissocier les informations provenant du dispositif de pointage inclus avec le clavier du portable de celles provenant de la souris additionnelle, on pourrait alors profiter de cette disposition pour utiliser l'interaction bimanuelle.

L'avènement de ces interfaces dépassant le modèle WIMP, qualifiées en conséquence de post-WIMP, dans les années quatre-vingt dix a été mis en lumière par van Dam [1997] qui applique cette terminologie aux mondes tridimensionnels ouverts par la réalité virtuelle. Si l'usage d'interacteurs standards est facilement identifiable comme frein à l'immersion requise pour les applications de la réalité virtuelle, ce constat est plus difficile à établir pour les applications classiques. Les interfaces WIMP présentent une cohérence permettant aux utilisateurs de réutiliser une partie de leur expérience d'un logiciel à l'autre. Tout écart à cette norme de fait impose aux utilisateurs un effort supplémentaire d'apprentissage pénalisant pour l'adoption du logiciel. Cependant, quelques exemples issus de travaux de recherche ou de logiciels grand public montrent que si le gain pour l'utilisateur est suffisant, il fera l'effort de l'apprentissage.

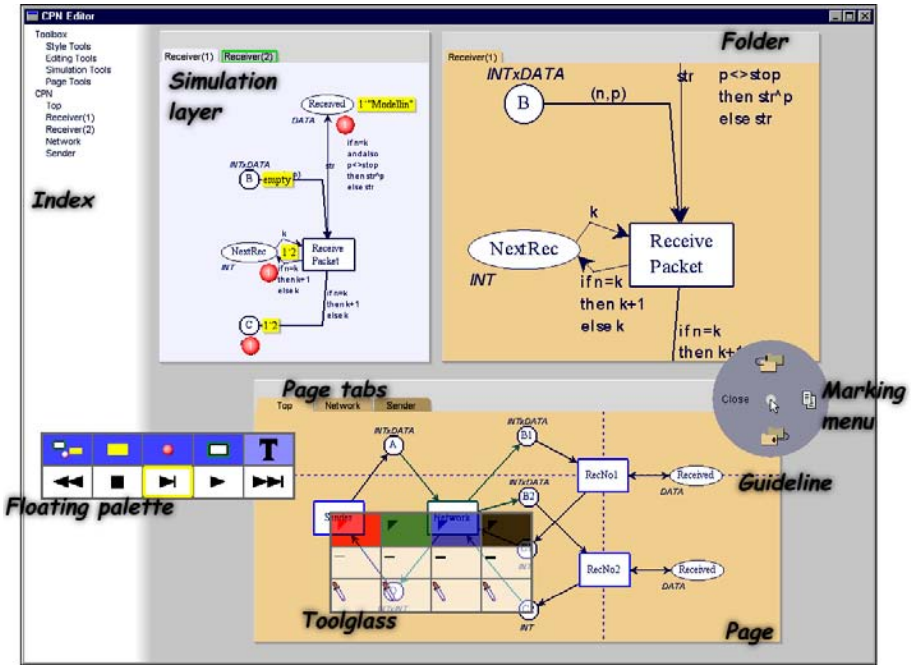


FIG. 1.3 – L'interface de CPN2000

(Illustration extraite de [Beaudouin-Lafon et Mackay, 2000])

C'est le cas dans le domaine des jeux, dont l'intérêt consiste essentiellement dans l'expérience interactive qu'ils procurent à leurs utilisateurs. C'est aussi le cas de logiciels utilisés pour accomplir des tâches complexes, qu'une interaction adaptée facilite. Ainsi, CPN2000 est un éditeur de réseaux de Petri colorés qui est le produit d'un travail de reconception complet de l'interface d'une application complexe existante [Beaudouin-Lafon et Lassen, 2000]. Quelques détails de son interface présentée dans la Figure 1.3 sautent aux yeux : l'absence de menus, ou l'absence de barres de défilement autour des fenêtres. En effet, la plupart des tâches s'effectuent à l'aide de manipulations directes. Par exemple, pour déplacer la zone vue à travers une fenêtre, il suffit d'"attraper" le fond de la vue avec le curseur et de le déplacer. Parmi d'autres particularités, cette application met en œuvre la manipulation bimanuelle, avec par exemple l'usage d'outils semi-transparents.

La conception de l'interface de CPN2000 a été réalisée de manière participative en impliquant ses futurs utilisateurs et en leur proposant des techniques d'interaction non-standards. Lors de cette conception, aucun a priori sur la réalisation n'a été fait. Pour pouvoir développer cette application à partir d'une conception non bridée par des choix de réalisation préalables, aucune boîte à outils existante n'a pu être utilisée. En effet, quelques impératifs simples éliminent la quasi-totalité des boîtes à outils courantes : un modèle graphique riche (permettant par exemple la transparence et le zoom), ou la nécessité de pouvoir gérer indépendamment plusieurs périphériques de pointage. Toute l'interface a donc dû être conçue et réalisée jusqu'au niveau de

détail le plus bas sans le support d'une boîte à outils existante. Les choix de conception ont ainsi été faits par l'équipe concevant le programme lui-même et non par les créateurs d'une boîte à outils n'ayant aucune visibilité sur l'application à réaliser, ce qui a permis d'obtenir un tel résultat. Cette application a été produite sur mesure pour ses utilisateurs, à leur grande satisfaction (plus de 5000 téléchargements et de 2000 licences au début de l'année 2005). Ce processus s'apparente donc plus à de la haute couture qu'à du prêt-à-porter, à l'inverse de celui qui régit la création de la plupart des applications WIMP.

La question suivante se pose alors : est-il possible de faciliter le développement de telles interfaces en fournissant des outils aux programmeurs, une architecture logicielle adaptée à leurs besoins, sans les enfermer aussitôt dans un cadre rigide bridant les possibilités de création et ramenant les interfaces produites dans le champs des interactions stéréotypées ? Nous le croyons et au long du travail de thèse présenté ici nous nous sommes attaché à le démontrer.

1.2 Proposition

Nous proposons ici une boîte à outils, HsmTk (pour *Hierarchical State Machine Toolkit*), visant à faciliter le développement d'applications graphiques interactives post-WIMP. Elle a été conçue en faisant des choix explicités qui bornent le moins possible le domaine de ses applications. Nous nous sommes attaché à réduire la tension créée par le niveau élevé d'abstraction requis pour faciliter la réalisation d'une part, et la finesse du contrôle que l'on doit offrir aux concepteurs pour qu'ils ne soient pas bridés par les idiomes spécifiques introduits par la boîte à outils d'autre part. Pour cela, la conception de la boîte à outils HsmTk a été gouvernée par deux axes :

- l'utilisation qui en est faite par le programmeur, c'est-à-dire rendre le code à écrire le plus concis et le plus signifiant possible ; et
- le degré de contrôle sur le détail, c'est-à-dire laisser effectivement les choix de conception aux concepteurs de l'application eux-mêmes.

Ce dernier point a conduit en particulier à choisir un modèle graphique qui ne soit pas celui du programmeur mais celui du designer graphique. Ce choix a priori, ainsi que d'autres, est explicité par la suite. Du côté de la programmation, ces choix ont conduit à fournir au programmeur une pyramide d'abstractions ayant un grain de plus en plus élevé. Ces abstractions offrent les fonctionnalités les plus élémentaires, puis des objets de plus en plus haut niveau reposant sur ces bases. Elles peuvent être recombinées à tous les niveaux de détail ; ainsi le programmeur n'est pas enfermé dans un stéréotype d'application. Cette approche permet au programmeur de fixer lui-même le compromis entre abstraction et spécialisation du code, et donc de l'application, qu'il écrit. S'il veut utiliser des éléments complètement génériques, il peut le faire très rapidement. Si au contraire, il veut adapter plus finement l'interaction à la tâche à laquelle elle est destinée, il en a le loisir, il y est même encouragé par l'aspect modulaire de la boîte à outils qui lui permettra de réutiliser son travail pour d'autres applications.

Pour aider à la concision, nous avons par ailleurs ajouté au langage mis à la disposition du programmeur une structure de contrôle dédiée à la programmation des interactions. Cette structure, les machines à états hiérarchiques,

inspirée de formalismes adaptés à la spécification de l'interaction, rend l'interaction plus idiomatique qu'elle ne l'est avec les langages impératifs traditionnels. Les machines à états hiérarchiques (HSM pour *Hierarchical State Machine*) encouragent le développement d'interactions riches en offrant une structure permettant de faire de ces interactions des entités à part entière et en facilitant ainsi leur adaptation à d'autres contextes et leur réutilisation.

1.3 Résultats

En parallèle de la conception de la boîte à outils, un ensemble d'applications ont été développées pour guider puis valider les choix effectués. Nous avons vérifié la possibilité de reproduire et d'intégrer facilement des techniques d'interaction classiques, tout en permettant des techniques plus avancées. C'est ainsi que nous avons reproduit sans peine des interacteurs WIMP usuels comme les boutons ou les menus hiérarchiques, mais aussi des techniques d'interactions plus récentes comme les outils bimanuels semi-transparents, ou des interacteurs à base de franchissement.

De nouvelles applications ont été réalisées par la suite pour vérifier a posteriori que la boîte à outils peut supporter des développements non-anticipés. En particulier, dans le cadre du projet RNTL INDIGO[1] (*Interactive Distributed Graphical Objects* ou objets interactifs graphiques distribués), la boîte à outils a été utilisée pour la réalisation d'applications interactives distribuées (jeu multi-joueur, explorateur de fichiers, vue radar). Plusieurs techniques d'interaction originales, dont notamment le pointage sémantique, ont été aussi réalisées à l'aide de la boîte à outils. Le pointage sémantique facilite le pointage de petites cibles à l'écran et permet de repenser assez profondément la conception des interfaces graphiques. Cette technique nécessite d'une part une introspection permettant de connaître le rôle des zones présentées à l'écran, et d'autre part un contrôle fin sur les périphériques de pointage. La boîte à outils HsmTk, en proposant ces deux éléments, a permis son prototypage et sa mise au point, puis sa validation par une expérimentation contrôlée.

La bibliothèque produite a ainsi montré son adéquation au prototypage et à la réalisation de techniques d'interaction avancées et d'applications interactives. Elle est mise à la disposition de la communauté sous une licence logiciel libre[2]. Elle inclut un préprocesseur qui étend le langage de programmation C++ avec les machines à états hiérarchiques.

1.4 Organisation de la thèse

Dans la première partie, nous présentons la boîte à outils HsmTk réalisée durant notre travail de thèse. Nous présentons tout d'abord, au travers d'une revue des techniques d'interaction existantes et des modèles d'interaction qui leurs sont liés, les éléments qui peuvent servir de cahier des charges à la conception d'une boîte à outils. Nous étudions ensuite les approches adoptées par les diverses boîtes à outils existantes, en précisant leurs atouts et leurs limitations. Nous nous appuyons sur ces états de l'art pour justifier les choix de conception que nous avons faits. Nous présentons alors la boîte à outils depuis

[1] Le projet INDIGO n'est pas lié à la technologie homonyme, mais postérieure, de Microsoft.
[2] HsmTk est diponible à l'adresse : `http://insitu.lri.fr/~blanch/projects/Hsm/`.

les abstractions les plus élémentaires jusqu'aux éléments de plus haut niveau. Un dernier chapitre est consacré à la principale contribution du travail : l'extension du langage de programmation pour faciliter la programmation des interactions.

Dans la seconde partie, nous validons la pertinence de notre approche en présentant plusieurs applications développées à l'aide de notre boîte à outils. Ces applications reprennent des techniques d'interaction classiques, des techniques plus avancées, ainsi que des techniques originales fruit de nos recherches. Nous présentons d'abord comment la boîte à outils HsmTk a permis de réaliser des applications graphiques interactives distribuées. Nous présentons ensuite le pointage sémantique et sa mise en œuvre à l'aide de la boîte à outils HsmTk.

Première partie

HsmTk, une boîte à outils pour les interactions avancées

Chapitre 2

Motivations

Nous détaillons ici les motivations de notre travail, en les appuyant sur un état des lieux du développement d'applications interactives et sur une revue de la littérature du domaine. Nous présentons ensuite les lignes directrices que nous nous sommes fixé pour la réalisation d'outils pour le développement d'applications interactives, et nous explicitons enfin les choix faits a priori dans notre approche.

L'interaction homme-machine est une science jeune, en pleine phase exploratoire. Cependant, elle dispose déjà de modèles et d'outils théoriques qui en constituent les fondements. Ces modèles sont de diverses natures : certains s'intéressent aux aspects architecturaux des applications interactives, recouvrant en cela des aspects du génie logiciel, alors que d'autres s'intéressent à la nature de l'interaction elle-même. D'autres encore se concentrent sur certaines tâches élémentaires constituant les interactions de l'utilisateur avec le système et rejoignent ainsi la psychologie expérimentale. D'autres, enfin, inscrivent l'interaction au sein du contexte plus général de la réalisation d'une tâche et cherchent à modéliser les aspects cognitifs de celle-ci. L'ensemble de ces aspects sont importants pour nous. Les premiers parce qu'ils nous fournissent un cadre dans lequel inscrire notre travail, en nous permettant de

mieux comprendre les enjeux du développement d'applications interactives. Les derniers en ce qu'ils nous donnent des critères objectifs montrant que les stéréotypes d'interaction dans lesquels sont confinés les programmeurs utilisant les boîtes à outils standards peuvent être largement améliorés. Nous entamons donc l'exposé de nos motivations par une présentation de ces différents modèles.

Nous explorons ensuite les réalisations existantes en distinguant deux ensembles de travaux. Nous présentons d'abord un ensemble de techniques d'interaction qui ont démontré leur intérêt — grâce aux modèles présentés avant ainsi qu'à des expérimentations contrôlées. L'étude de ces techniques d'interaction nous permettra de dégager des prérequis pour la suite de notre travail. Nous exposons ensuite une série de travaux sur les boîtes à outils de développement d'applications interactives. Ces travaux nous permettent de comprendre comment nos préoccupations ont déjà été abordées, et si des solutions intéressantes leur ont déjà été proposées. Ils nous fournissent par ailleurs des références auxquelles comparer notre travail.

À partir de cet état de l'art, nous présentons ensuite les lignes directrices que nous en avons dégagées pour notre projet. C'est à l'intérieur de ce cadre que se développera l'ensemble de notre travail. Nous explicitons en particulier certains choix que nous avons effectués a priori en en détaillant les motivations.

2.1 Des modèles pour l'interaction homme-machine

Les modèles proposés par la littérature concernant l'interaction homme-machine sont de diverses natures. Certains mettent l'accent sur le système interactif lui-même et proposent pour celui-ci des modèles d'architecture à des niveaux d'abstraction variable. D'autres s'intéressent à l'interaction elle-même et proposent de formaliser la notion d'échange ou de dialogue entre le système et ses utilisateurs. Enfin, certains s'intéressent à la modélisation de la performance de l'utilisateur pour offrir des moyens objectifs et quantitatifs d'évaluer une technique d'interaction ou un système. Nous abordons ici ces trois aspects successivement. Pour une vue plus large sur les modèles pour l'interaction, on pourra se reporter au panorama dressé par Carrol [2003].

2.1.1 Modèles d'architecture

Les différents modèles proposés pour modéliser les systèmes interactifs considèrent généralement qu'une application interactive comporte un noyau fonctionnel, souvent préexistant, qui ne relève pas forcément du modèle lui-même. Ce noyau fonctionnel réalise toutes les fonctions liées au domaine de l'application, sur des objets abstraits, indépendants de la représentation qui en est fournie aux utilisateurs. L'interface est ainsi définie comme la partie de l'application qui établit, au travers de divers mécanismes, une correspondance entre le vocabulaire de l'utilisateur et celui du noyau fonctionnel. C'est l'architecture de cette interface qui est l'objet des modèles présentés ici.

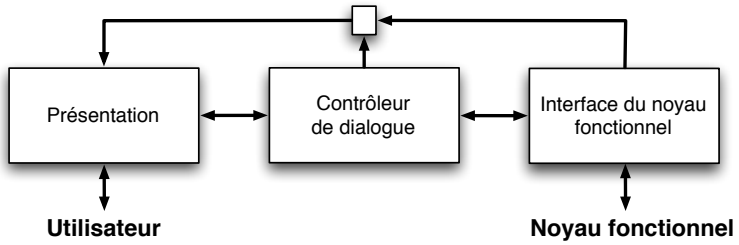


FIG. 2.1 – Le modèle de Seeheim

Modèles linguistiques

Les premiers modèles proposés reposent sur la notion de traduction du vocabulaire de l'interaction vers celui du noyau fonctionnel. Les problématiques de la traduction ont été défrichées depuis longtemps en informatique par les recherches sur la compilation des langages de programmation. Les modèles linguistiques identifient ainsi trois niveaux dans l'interaction :

- le niveau *lexical* qui définit le vocabulaire d'entrée ou de sortie (clic, déplacement du curseur et affichage de texte, d'icones par exemple) ;
- le niveau *syntaxique* qui permet de segmenter des séquence d'entrées (double clic par exemple) ou de structurer des sorties ;
- le niveau *sémantique* qui spécifie l'action effective à effectuer par le noyau fonctionnel.

Le modèle de Seeheim. Le premier modèle proposé pour les systèmes interactifs, issu d'un atelier sur les systèmes interactifs tenu à Seeheim en 1983 [Pfaff, 1985], sépare ainsi l'interface en trois couches (Figure 2.1) :

- la *présentation* qui est la couche gérant le dialogue au niveau lexical en réalisant les entrées-sorties sur les périphériques physiques ;
- le *contrôleur de dialogue* qui maintient un état permettant de réunir des séquences d'interactions élémentaires en commandes et d'introduire des modes dans l'interaction ;
- l'*interface du noyau fonctionnel* qui adapte finalement les commandes au noyau fonctionnel et inversement qui transforme les objets abstraits de l'application en objets présentables à l'utilisateur.

Un canal de communication direct entre la couche de présentation et la couche d'interface du noyau fonctionnel, court-circuitant ainsi le contrôleur de dialogue, a été ajouté pour des question d'efficacité.

Ce modèle a été proposé initialement pour des systèmes purement textuels. Le cadre qu'il propose reste très abstrait et peu détaillé. Il a cependant été raffiné par la suite alors que les boîtes à outils graphiques enrichissaient le vocabulaire de l'interaction.

Le modèle Arch. Le modèle Arch [1992] scinde la couche présentation du modèle de Seeheim en une couche de présentation, qui communique toujours avec le contrôleur de dialogue, et une couche d'*interaction*, à qui revient la charge de la communication avec l'utilisateur. La couche d'interaction est constituées des interacteurs de la boîte à outils utilisée, et communique avec

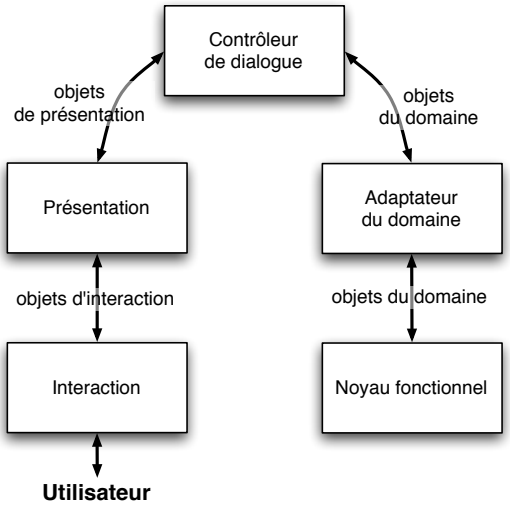


FIG. 2.2 – Le modèle Arch

les dispositifs physiques. La couche de présentation est une première abstraction de ces interacteurs, indépendante de la plate-forme, qui sert de médiateur entre le composant d'interaction et le contrôleur de dialogue. L'interface du noyau fonctionnel est rebaptisée *adaptateur du domaine*. Cet adaptateur permet en particulier la réalisation de tâches portant sur les objets du domaine mais ne faisant a priori pas partie du noyau fonctionnel. Par exemple, il peut donner des informations nécessaires à un retour sémantique au contrôleur de dialogue.

Le modèle Arch réorganise ces couches pour en faire une arche (Figure 2.2) dont les deux fondations sont le noyau fonctionnel et la couche d'interaction, tous deux préexistants, puisqu'ils représentent respectivement le cœur de l'application et la boîte à outils utilisée pour rendre celle-ci interactive.

Par ailleurs, le modèle Arch précise un peu la nature des communications entre les divers acteurs du modèle. Ainsi, trois types d'objets transitent entre les composants :

- les *objets du domaine* qui contiennent des données provenant du noyau fonctionnel et qui sont échangés par le noyau fonctionnel, l'adaptateur du domaine et le contrôleur de dialogue ;
- les *objets de présentation* qui représentent les actions de l'utilisateurs et les objets qui lui sont présentés, mais de manière indépendante de leur réalisation effective ; et
- les *objets d'interaction* qui sont des objets propres à la boîte à outils elle-même qui réalisent l'interaction proprement dite.

Modèles à agents

Avec l'apparition des langages à objets, des modèles ayant une granularité plus fine que celle des modèles linguistiques ont été proposés. Ils fournissent une description plus opérationnelle des composants de l'interface sous la forme d'agents collaborant entre eux.

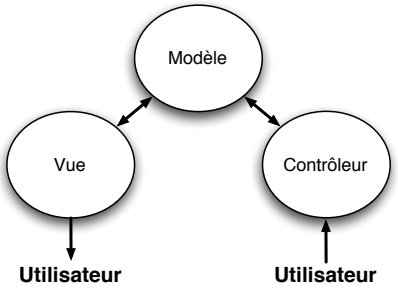


FIG. 2.3 – Le modèle MVC

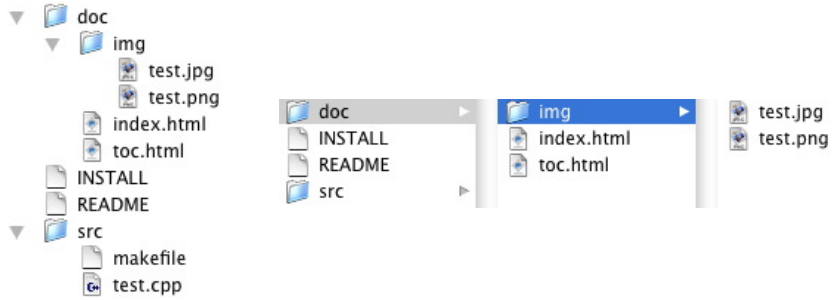


FIG. 2.4 – Deux vues d'un même modèle

MVC. Le premier modèle à agents est très lié au langage Smalltalk et à son environnement de développement et d'exécution [Goldberg et Robson, 1981]. Il s'agit du modèle MVC *(Model, View, Controller)* [Schmucker, 1986, Krasner et Pope, 1988]. MVC sépare chaque objet de l'interface (ou agent MVC) en trois composants communicant entre eux (Figure 2.3) :

- le *modèle* qui est le noyau fonctionnel de l'agent, sa forme la plus simple pouvant être une simple valeur. Ce composant est chargé de notifier les vues, potentiellement multiples, des changements de l'état du modèle induit par l'application ou par les contrôleurs ;
- la *vue* qui offre une représentation du modèle à l'utilisateur et se met à jour en fonction de l'évolution du modèle ;
- le *contrôleur* qui est chargé d'interpréter les actions de l'utilisateur pour notifier le modèle de celles-ci.

Ce modèle à agents s'adapte bien aux concepts des langages à objets, et les trois composants MVC sont souvent réalisés par trois objets distincts. En utilisant les mécanismes du polymorphisme, des vues différentes peuvent alors être utilisées pour un même modèle. La Figure 2.4 illustre ainsi deux vues différentes d'un même modèle : un système de fichiers arborescent.

Le fait est, par contre, que vue et contrôleur sont intimement liés et ont été souvent regroupés au sein d'une même composant par la suite. Ils forment ce qui est appelé couramment dans les boîtes à outils à base d'interacteurs le *look & feel* des éléments de l'interface graphique. Le modèle PAC présenté ci-dessous les réunit ainsi au sein de son composant "présentation". D'autres variantes de MVC ont été proposées, adaptées à des contextes par-

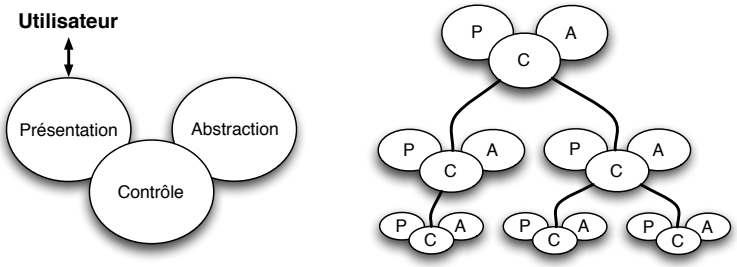


FIG. 2.5 – Le modèle PAC et une hiérarchie d'agents PAC

ticuliers. Le modèle ALV *(Abstraction-Link-View)*, par exemple, met en œuvre des contraintes pour maintenir un lien entre une abstraction et des vues (qui réunissent elles aussi la vue et le contrôleur de MVC) sur cette abstraction [Hill, 1992]. $MV_{ZM}C$ a été adapté pour l'interaction distribuée à l'aide de pointeurs multiples [Dragicevic et Fekete, 1999]. Par ailleurs, le modèle MVC est souvent cité en référence pour introduire une architecture dans les applications Web.

PAC. Le modèle PAC (Présentation, Abstraction, Contrôle) [Coutaz, 1987] propose un modèle d'agents interactifs de granularité similaire à celle de MVC. Il regroupe vue et contrôleur au sein du même composant de l'agent : la *présentation*. Le modèle de MVC devient quant à lui l'*abstraction* et encapsule toujours la sémantique du noyau fonctionnel. Enfin, la synchronisation entre ces deux composants est confiée explicitement à un troisième intervenant : le *contrôle*. Ce nouveau composant permet aussi de rendre explicites les relations des divers agents PAC qui peuvent communiquer entre eux au travers de ces contrôleurs. Le modèle PAC permet, par cette communication entre contrôleurs, de construire une décomposition hiérarchique de l'application interactive en adaptant le niveau de détail de chaque agent à des niveaux de précision de plus en plus fins (Figure 2.5).

PAC/Amodeus. Le modèle PAC/Amodeus [Nigay et Coutaz, 1991] propose une unification des modèles linguistiques et à agents en reprenant le modèle Arch et en raffinant son contrôleur de dialogue par l'utilisation du modèle PAC. Le contrôleur de dialogue devient ainsi une hiérarchie PAC. Ses communications avec l'adaptateur du domaine se font au travers des composants "abstraction" des agents PAC, alors que celles qui concernent la présentation se font au travers des composants "présentation". PAC/Amodeus a été utilisé avec succès, notamment pour modéliser des applications multimodales [Nigay et Coutaz, 1995]. De l'expérience acquise par l'utilisation du modèle PAC et de ses dérivés a été dégagé un ensemble de règles heuristiques permettant de guider concrètement la structuration en agents PAC des applications interactives [Nigay et Coutaz, 1991]. La première suggère ainsi par exemple de modéliser les fenêtres qui concrétisent un espace de travail par un agent. La seconde, par exemple encore, propose que les vues multiples d'un même concept doivent être gérées par un agent "vue multiple", père des agents qui représentent chacun l'une de ces vues.

2.1.2 Modèles de l'interaction

Les modèles de l'interaction s'intéressent à l'interaction du point de vue de l'utilisateur. S'ils ne proposent pas directement de techniques d'interaction, ils donnent par contre des principes à suivre lors de leur conception. Ils regroupent ainsi les techniques d'interaction au sein de grandes familles.

Les interfaces conversationnelles

L'un des premiers modèles d'interaction utilisé fut celui de l'interaction conversationnelle. C'est ce modèle qui est utilisé par les langages de commandes : l'utilisateur spécifie à l'invite du système une commande ainsi que ses arguments. Il reçoit en réponse le résultat de cette commande ou, si le résultat est uniquement en un effet de bord modifiant l'état du système, un retour le renseignant sur l'accomplissement de la commande. Ce retour n'est d'ailleurs pas toujours fourni, auquel cas seule l'expérience de l'utilisateur ou une l'utilisation d'une commande explicite permettra de connaître le résultat de l'action.

Ce type d'application est maintenant réservé à des domaines spécifiques, en général proches du système informatique lui-même. Il structure par ailleurs l'application assez directement autour d'une boucle qui effectue la lecture ligne à ligne d'une commande dans une console, l'évalue, et affiche son résultat. C'est sans doute le système interactif le plus primitif tant l'interface (une simple invite) est peu suggestive et requiert de l'utilisateur une connaissance préalable du vocabulaire et de la syntaxe compris par le programme.

Pour réduire cet effort de mémorisation, il est possible de placer les commandes dans des menus. Une autre alternative, pour effectuer des tâches particulières, et de donner l'initiative à la machine qui demande alors simplement la saisie de paramètres grâce à une succession de formulaires. Ces styles d'interaction peuvent être entremêlés et cohabitent souvent dans les applications.

La manipulation directe

La manipulation directe de Shneiderman [1983, 1998], partant de l'observation des systèmes interactifs de l'époque, et notamment des tableurs qui viennent d'apparaître, et des jeux répandus sur les bornes d'arcade, propose quelques principes pour les interfaces graphiques. Ces principes doivent guider la réalisation des interfaces et se déclinent ainsi :

- les objets de l'application doivent posséder une *représentation graphique persistante* (par exemple un fichier sera représenté par un icone dans un explorateur de fichiers) ;
- les actions sur ces objets doivent se réaliser par des *actions physiques* et non par l'invocation de commandes d'un langage (par exemple, la suppression d'un fichier se fera en le déplaçant dans une poubelle, cette dernière incarnant la commande de suppression) ;
- Les actions doivent être *incrémentales* (donc rapides) et *réversibles* (on peut "sortir" un fichier de la poubelle).

Ces principes facilitent l'apprentissage en réduisant les risques et en encourageant ainsi la découverte par l'exploration de l'interface. Ils réduisent ainsi l'apprentissage préalable nécessaire à la prise en main du programme et per-

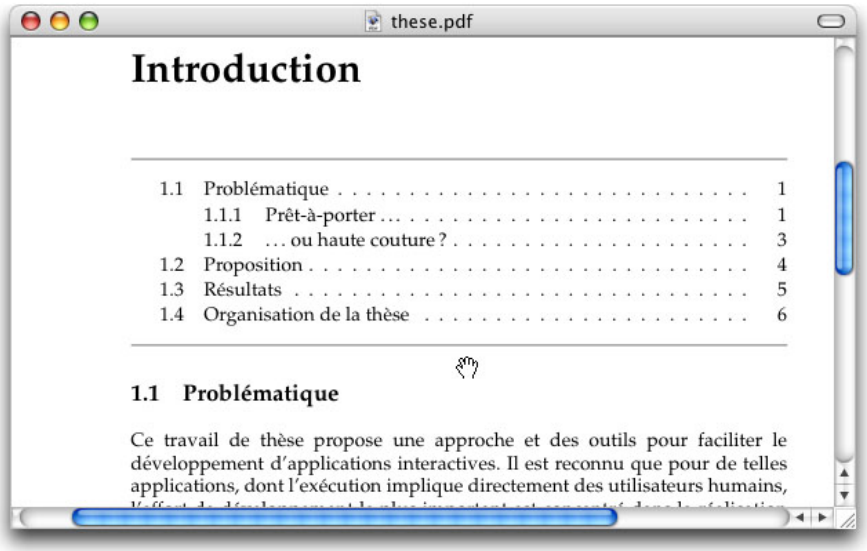


FIG. 2.6 – La manipulation directe d'un document

mettent la découverte par l'utilisateur de vocabulaires de manipulation a priori intuitifs.

La manipulation directe est présente dans toutes les interfaces graphiques actuelles. Cependant, l'introduction des interacteurs comme briques élémentaires des applications interactives a un effet pervers sur la manipulation directe. En effet, si les interacteurs sont manipulés directement, ils sont rarement les réels objets d'intérêt — documents, fichiers, etc. Ils ne servent en fait souvent qu'à les manipuler indirectement. Par exemple, une application présentant un document trop grand pour tenir entièrement dans le cadre de la fenêtre permet de modifier la vue en manipulant directement des barres de défilement mais rarement le document lui-même. Certaines applications proposent tout de même un outil (souvent illustré par un curseur en forme de main) qui permet de manipuler le document directement (Figure 2.6).

L'interaction instrumentale

On peut constater que l'indirection observée ci-dessus est en fait toujours présente, au moins dans sa forme la plus banale : le curseur présenté à l'écran est déjà un objet qui est lui-même manipulé directement par le dispositif physique de la souris, et qui manipule à son tour les objets de l'application. C'est ainsi que Beaudouin-Lafon [1997, 2000] propose d'introduire le concept d'instrument qui caractérise la plupart des systèmes utilisés par les humains.

Dans la manipulation instrumentale, les actions sur les objets d'intérêt s'effectuent grâce à des instruments, eux-mêmes manipulés par l'utilisateur. Les barres de défilement citées ci-dessus sont dans ce modèle des instruments (Figure 2.7). Elles réagissent aux actions de l'utilisateur (manipulation des divers composants de la barre de défilement) en modifiant l'objet d'intérêt. En plus de la modification de l'objet lui-même, l'instrument peut offrir son propre

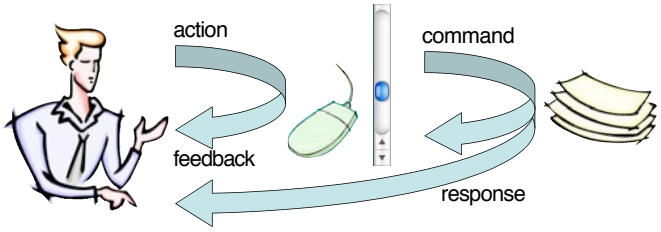


FIG. 2.7 – L'instrument "barre de défilement"

(Illustration extraite de [Beaudouin-Lafon, 2004])

retour visuel (ici, la modification de la position du curseur dans la barre de défilement) pour l'informer des effets de ses actions.

Un aspect intéressant de ce modèle de l'interaction est qu'il fournit des critères permettant d'évaluer l'adéquation d'une technique d'interaction à la tâche qu'elle permet de réaliser. Ainsi Beaudouin-Lafon [2000] propose trois mesures pour les instruments :

- le *degré d'indirection* qui mesure la distance spatiale et temporelle entre l'instrument et l'objet d'intérêt — les poignées de redimensionnement d'une figure géométrique ont par exemple un faible degré d'indirection spatiale, alors qu'un bouton dans une barre d'outil pour modifier la graisse d'une fonte en a un grand, de même, l'apparition d'une boîte de dialogue demandant la confirmation d'une modification introduit une forte indirection entre une action et son résultat, tant spatialement que temporellement ;
- le *degré d'intégration* qui mesure le rapport entre le nombre de dimensions physiques et le nombre de dimensions logiques qui sont mises en jeu lors de l'utilisation de l'instrument — une barre de défilement manipulée par le curseur de la souris se déplaçant dans le plan alors que les déplacements du curseur de la barre de défilement sont contraints à un axe unidimensionnel a un moins bon degré d'intégration que l'outil représenté par une main (Figure 2.6) qui permet de manipuler un document dans les deux dimensions du plan simultanément ;
- le *degré de compatibilité* qui mesure la similarité entre les manipulations qui sont effectuées sur l'instrument et leurs effets sur l'objet d'intérêt — de ce point de vue, la barre de défilement est moins bien adaptée que l'outil "main" puisque, avec la barre de défilement, le déplacement du curseur vers le bas entraîne un déplacement de l'objet d'intérêt vers le haut.

Autres modèles

Dans des domaines aux problématiques complexes comme les interface multimodales ou la réalité virtuelle qui mettent en jeu par exemple la reconnaissance de gestes ou le traitement du langage naturel, des modèles adaptés ont été proposés. Nous n'en faisons pas état ici, et bornons notre étude à des modalités d'interaction plus faciles à mettre en œuvre.

2.1.3 Modèles de performance de l'utilisateur

L'un des critères facile à mesurer, et donc couramment utilisé pour évaluer les techniques d'interaction, est le temps requis par un utilisateur pour effectuer un tâche donnée. En plus d'être purement quantitatif et facilement mesurable, ce critère de temps a le mérite de bénéficier, pour certaines tâches élémentaires, de modèles empiriques qui en permettent la prédiction de manière assez robuste.

Modèle du choix : loi de Hick

Le modèle de Hick [1952] prédit le temps nécessaire pour choisir un élément parmi un ensemble donné. Il ne modélise pas le temps de réflexion pour élaborer le choix, ni la recherche du bon élément, mais simplement la sélection d'un élément particulier. Ce temps est en fait proportionnel à la quantité d'information (ou l'entropie au sens de la théorie de l'information) qu'apporte le choix :

$$t = b \cdot \log_2 (n + 1) \tag{2.1}$$

avec t le temps pris par le choix, b une constante, et n le nombre de possibilités en jeu. b caractérise ainsi la bande passante de l'utilisateur vers le système puisqu'il mesure le temps nécessaire pour transmettre un bit d'information (typiquement 150 ms).

Le modèle de Hick permet de mieux comprendre le comportement de l'utilisateur et de faire des choix de conception adaptés. Par exemple, réduire le nombre d'éléments présents dans une liste ou dans un menu permet de ne pas surcharger l'utilisateur et d'éviter de le perdre dans des alternatives complexes.

Modèle du pointage : loi de Fitts

Le modèle de Fitts [1954] fournit une prédiction de la performance pour la tâche élémentaire sur laquelle repose la plupart des interactions au sein des interfaces graphiques : le pointage. Il indique que le temps passé pour atteindre une cible ayant une taille donnée située à une distance donnée est une fonction affine du logarithme du rapport entre la taille et la distance de la cible :

$$t = a + b \cdot \log_2 \left(\frac{2D}{W} \right) \tag{2.2}$$

avec t le temps de pointage, a et b deux constantes et D et W caractérisant respectivement l'amplitude et la précision de la tâche (voir la Figure 2.8). Là encore, b caractérise la bande passante de l'utilisateur vers le système et mesure le temps nécessaire pour transmettre un bit d'information (typiquement 100 ms soit une bande passante de 10 bits par seconde).

Le modèle de Fitts met en évidence que la difficulté d'une tâche de pointage est liée au rapport sans dimension entre la taille de la cible et sa distance. Ainsi sélectionner une cible deux fois plus petite, mais située deux fois plus près qu'une autre présente la même difficulté et prendra le même temps. La grandeur qui caractérise cette difficulté est dénommée indice de difficulté (ID)

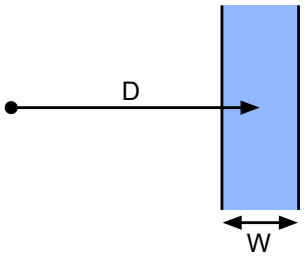


FIG. 2.8 – La tâche Fitts

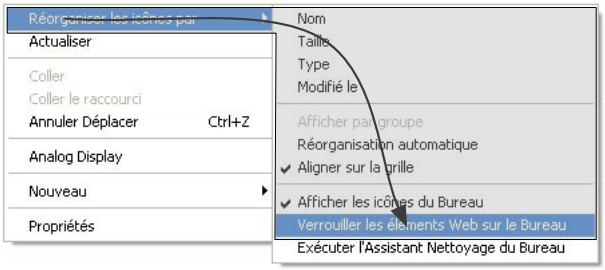


FIG. 2.9 – Suivi de tunnel dans un menu hiérarchique mal conçu

et comporte le terme logarithmique de la formule précédente :

$$ID = \log_2 \left(\frac{2D}{W} \right). \tag{2.3}$$

L'indice de difficulté caractérise, comme le terme logarithmique du modèle de Hick, une quantité d'information transmise — le choix de la cible par rapport au reste de l'environnement. Son unité est ici le bit, puisque le logarithme est en base 2.

Outre ses capacités prédictives, le modèle de Fitts formalise avec l'indice de difficulté une notion intuitive qui est ainsi explicitée : plus une cible est située loin du curseur de la souris, et plus elle est petite, plus elle est difficile à atteindre (ID augmente). Ce constat est la source d'inspiration de beaucoup de techniques d'interaction décrites ci-après, à la Section 2.2.1, qui cherchent d'une manière ou d'une autre à réduire la difficulté du pointage en rapprochant les cibles potentielles de la souris, et/ou en augmentant leur taille.

Modèle de suivi de tunnel : loi de *steering*

Plus récemment, le modèle de Fitts a été utilisé pour construire un modèle prédisant le temps de franchissement d'un tunnel [Accot et Zhai, 1997]. Ce modèle permet en particulier de prédire des temps d'exécution de parcours contraints comme le choix dans des menus hiérarchiques mal conçus qui obligent à ne pas sortir de l'élément courant pour aller sélectionner l'un de ses fils (voir la Figure 2.9).

Pour la traversée d'un tunnel rectangulaire de longueur A et de largeur W, le modèle donne pour t, temps de traversée :

$$t = a + b \cdot \frac{A}{W} \tag{2.4}$$

avec a et b deux constantes déterminées empiriquement. Ce modèle se généralise pour des tunnels dont la largeur varie le long du parcours et qui ne sont pas rectilignes.

Modèles de tâches prédictifs

Des modèles de plus haut niveau ont été proposés pour décrire des tâches moins élémentaires que celles de pointage ou de choix. Keystroke et GOMS de Card *et al.* [1980, 1983] permettent par exemple, étant donné la décomposition d'une tâche en tâches élémentaires de type pointage (Fitts) et choix (Hick), de prédire le temps d'exécution de cette tâche. Ces modèles ajoutent aux termes de performance pure (pointage et choix), des opérations cognitives modélisant la réflexion de l'utilisateur.

Plus récemment le modèle CIS proposé par Appert *et al.* [2004] affine ce genre de modèles prédictifs en prenant en compte le contexte — état de l'interface, degré de parallélisme des entrées — dans lequel a lieu l'interaction pour fournir des évaluations précises du temps nécessaire à la réalisation d'une tâche à l'aide d'une technique d'interaction donnée.

2.1.4 Conclusion

Nous avons présenté trois nivaux de modèles pour l'interaction homme-machine : des modèles d'architecture, des modèles d'interaction, et des modèles de performance. Tous trois nous apportent des enseignements.

Les modèles d'architecture proposent des cadres plus ou moins abstraits pour la réalisation d'applications interactives, et donnent des lignes directrices pour les structurer. Ils se sont raffinés et adaptés aux nouvelles spécificités des langages et des outils disponibles (langages à objets, environnements graphiques) ainsi qu'aux nouvelles capacités de communication des ordinateurs avec les utilisateurs et leur environnement (terminaux graphiques, nouvelles modalités). Cependant, s'ils peuvent fournir des guides très concrets pour structurer les applications interactives, ils ne modélisent pas l'interaction en tant que telle. Ils ne peuvent donc aider pour les choix de conception de l'interaction elle-même, ces choix ayant pourtant un impact décisif sur l'utilisabilité de l'application interactive.

Les modèles d'interaction s'intéressent directement au rapport entre l'utilisateur et l'application. Ils fournissent des éléments permettant de guider les choix de conception de l'interaction elle-même. Mieux, le modèle de l'interaction instrumentale donne trois critères permettant d'évaluer la pertinence d'un instrument d'interaction pour la réalisation d'une tâche, et donne ainsi des éléments pour évaluer à quel point une manipulation est "directe". Ce pas vers l'introduction de critères permettant d'évaluer les techniques d'interaction les unes par rapport aux autres est donc décisif pour progresser dans l'exploration de l'espace de conception qu'elles offrent.

Enfin, les modèles de performance permettent d'évaluer objectivement l'apport de nouvelles techniques d'interaction. Ils permettent ainsi de montrer que l'introduction de techniques d'interaction avancées permet de repousser certaines limites des interfaces traditionnelles. Et c'est bien parce que ces techniques constituent de réels progrès qu'il est nécessaire de pouvoir les intégrer facilement dans les applications que nous développons.

2.2 Techniques d'interaction

Les techniques d'interaction utilisées par les applications et par l'environnement des ordinateurs personnels n'ont pratiquement pas évolué depuis l'apparition de la métaphore du bureau et sa popularisation [Beaudouin-Lafon, 2004]. Pourtant, les techniques facilitant l'interaction foisonnent dans la littérature de l'interaction homme-machine. Il est particulièrement frappant de noter comme la plupart des techniques d'interaction récentes — certaines, comme les menus circulaires [Callahan *et al.*, 1988, Hopkins, 1991] ont tout de même plus de quinze ans — ne sont pas adoptées dans les logiciels d'utilisation courante, alors que leurs apports ont souvent été démontrés par des analyses et des expérimentations contrôlées. L'informatique grand public ayant pourtant comme principal moteur commercial l'ajout de fonctionnalités "visibles", il s'agit là d'un paradoxe, ou du moins d'un symptôme du fait que de telles interactions sont difficiles à mettre en œuvre dans un système existant.

Pour prétendre offrir des outils permettant de réaliser facilement des techniques d'interaction, ainsi que d'en concevoir de nouvelles, il nous faut étudier les techniques existantes et dégager les aspects nécessaires à leur réalisation. En explorant la diversité de ces techniques d'interaction avancées, même si nous nous limitons ici à celles qui ont leur place sur le bureau d'aujourd'hui[1], nous pouvons remarquer une tendance générale. Alors que l'interacteur classique est caractéristique d'une interaction complètement séquencée, les techniques récentes tendent vers une interaction de plus en plus continue, qui a été qualifiée de "fluide" (voir par exemple [Guimbretière *et al.*, 2001, Ramos et Balakrishnan, 2003]). Nous présentons quelques exemples illustrant cette marche vers la fluidité de l'interaction.

2.2.1 Facilitation du pointage

Le modèle de Fitts a permis de prendre conscience de l'importance de la disposition des cibles par rapport au curseur pour la facilité avec laquelle elles peuvent être atteintes. Ce résultat de psychologie expérimentale a une influence directe sur bon nombre de techniques qui cherchent à réduire le temps de pointage nécessaire à la réalisation de certaines tâches (Balakrishnan [2004] en a répertorié un vaste échantillon). Nous présentons quelques-unes des plus significatives ici.

[1] Les mondes tridimensionnels explorés par la réalité virtuelle en particulier ont leurs problématiques propres que nous n'aborderons pas ici.

FIG. 2.10 – Le Dock de Mac OS X

Sélection de cible

La désignation d'un objet à l'écran est l'une des tâches élémentaires de l'interaction. Beaucoup de techniques d'interaction ont donc pour motivation de faciliter cette désignation. Cette facilitation peut intervenir, d'après la loi de Fitts, à deux nivaux : en jouant sur la taille de la cible elle-même et sur sa distance au curseur.

Augmenter la taille. Plusieurs techniques cherchent ainsi à augmenter la taille des cibles. La place à l'écran étant une ressource limitée, afficher des objets de grande taille est incompatible avec afficher de grandes quantités d'objets. Pour assouplir ce compromis, des techniques comme celle utilisée par le Dock de Mac OS X (Figure 2.10), changent transitoirement la taille des objets lorsque l'utilisateur est susceptible de vouloir les atteindre, c'est-à-dire lorsque le curseur se trouve à leur proximité.

Une étude [McGuffin et Balakrishnan, 2002] a ainsi montré que le fait d'augmenter la taille d'une cible, cette augmentation ayant lieu uniquement lorsque le curseur en est à proximité, rend la tâche de pointage aussi facile que si la cible avait initialement sa taille grandie. De même, ajouter des "oreilles" autour d'objets de faible taille de l'interface — les bords des fenêtres ou les poignées de redimensionnement par exemple — comme suggéré par [Cockburn et Firth, 2003] permet de faciliter leur préhension. Peu après, il a été montré par [Zhai *et al.*, 2003] que cet effet n'était pas dû à une anticipation de l'utilisateur puisque l'effet persiste même si le grossissement n'est pas prévisible. Ces études ont cependant montré que la technique du Dock, elle, n'améliore pas la performance du pointage : le déplacement de la cible qui accompagne son grossissement annule l'avantage que procure la taille accrue.

Réduire la distance. Parallèlement à l'augmentation de la taille, d'autres travaux ont étudié la possibilité de réduire la distance. Le *drag-and-pop* de Baudish *et al.* [2003] approche par exemple temporairement du curseur les cibles potentielles d'un mouvement de glissé sur le bureau dès que celui-ci est initié (Figure 2.11). Cette technique utilise la compatibilité potentielle avec l'objet déplacé pour sélectionner les cibles potentielles. Celles-ci sont alors approchées en gardant un lien visuel avec leurs positions originales et en maintenant certaines relations géométriques entre les objets déplacés pour que leurs positions relatives restent similaires. Pour des objets situés à grande distance, cette technique améliore significativement le temps de pointage.

Hascoët [2003] puis Collomb et Hascoët [2004] ont proposé des techniques pour lesquelles c'est au contraire l'objet du déplacement qui est projeté vers les cibles potentielles. Ils utilisent ainsi une métaphore de tir à l'arc (*drag-and-*

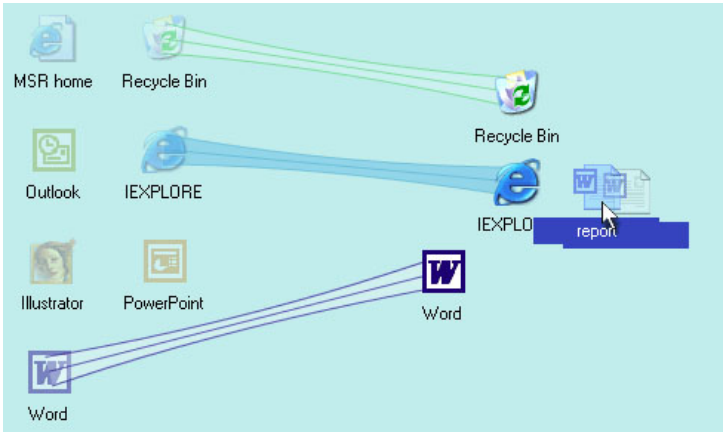


FIG. 2.11 – Le *drag-and-pop*

(Illustration extraite de [Baudisch *et al.*, 2003])

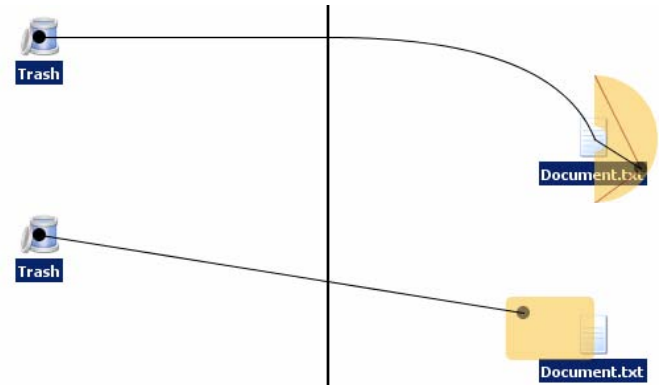


FIG. 2.12 – Métaphores de lancer pour réduire la distance aux cibles

(Illustration extraite de [Collomb et Hascoët, 2004])

throw, Figure 2.12 en haut) ou de pentographe (*push-and-throw*, Figure 2.12 en bas) pour permettre d'envoyer l'objet sur des cibles distantes tout en ne nécessitant que des mouvements de faible amplitude. Ils appliquent leurs techniques en particulier à des configurations comportant des écrans multiples, dans lesquelles les objets à déplacer et les cibles ne sont pas forcément situés dans le même espace de travail. Hasoët *et al.* [2005] présentent par ailleurs une étude comparative de ces techniques et de toutes celles dérivées du cliquer-tirer *(drag-and-drop).*

Une autre technique, utilisée quotidiennement, est celle des menus contextuels linéaires (Figure 2.13 à gauche) qui apparaissent directement sous le curseur de la souris, en général après un clic droit, et proposent une liste d'actions possibles en tenant compte du contexte dans lequel se trouve le curseur. Ils mettent ainsi les actions les plus probables de l'utilisateur "à portée de main" en réduisant la distance de ces commandes au curseur. Cette idée simple a été poussée beaucoup plus loin avec les menus circulaires.

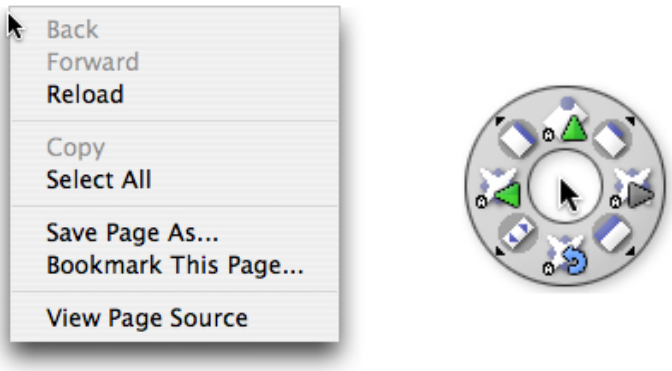


FIG. 2.13 – Un menu contextuel linéaire et un menu circulaire pour la navigation hypertexte

Menus circulaires

Les menus circulaires ou *pie-menus* [Hopkins, 1991] ont été proposés pour remplacer les menus contextuels linéaires habituels qui apparaissent sous la souris lors d'un clic du bouton droit. Leur principal intérêt est de positionner tous les éléments du menu à proximité immédiate du curseur en les plaçant en cercle autour de celui-ci. La Figure 2.13 montre à gauche un menu contextuel classique pour la navigation dans un document hypertexte. La partie droite de la Figure 2.13 montre quant à elle un menu contextuel circulaire offrant les mêmes fonctionnalités. Le positionnement en cercle réduit la distance du pointage au minimum pour chaque élément du menu et le temps de pointage devient ainsi à la fois constant et minimal pour tous les éléments, ce qui améliore sensiblement la performance obtenue avec les menus contextuels classiques [Callahan *et al.*, 1988].

Un autre avantage de cette disposition, par rapport à une disposition purement linéaire rencontrée dans les menus contextuels classiques, est de permettre aux utilisateurs de tirer parti de leur expérience en mémorisant la direction des éléments souvent utilisés. Il est en effet plus facile de mémoriser la direction d'un geste que son amplitude qui, dans le cas du menu linéaire, doit être choisie précisément pour pouvoir sélectionner une cible donnée. Cette faculté de mémorisation est exploitée par les *marking-menus* [Kurtenbach, 1993, Kurtenbach et Buxton, 1993], une extension des menus circulaires. Cette variante consiste à ne faire apparaître le menu qu'après un certain temps si le curseur n'a pas bougé après le clic initial. Ainsi, si l'utilisateur effectue le mouvement de sélection d'un élément du menu rapidement, celui-ci n'a même pas le temps d'apparaître et l'interaction peut continuer de manière fluide. Cette technique permet d'effectuer une transition continue entre un mode débutant, pour lequel le menu apparaît, et un mode expert, pour lequel l'interaction devient purement gestuelle. En effet, les menus circulaires pouvant être utilisés en cascade, la sélection d'un élément particulier dans la hiérarchie des menus conduit à dessiner une trace qui est reconnue par le système. La Figure 2.14 montre deux variantes dans lesquelles la trace consiste en un seul geste (en haut) ou en un ensemble de marques simples

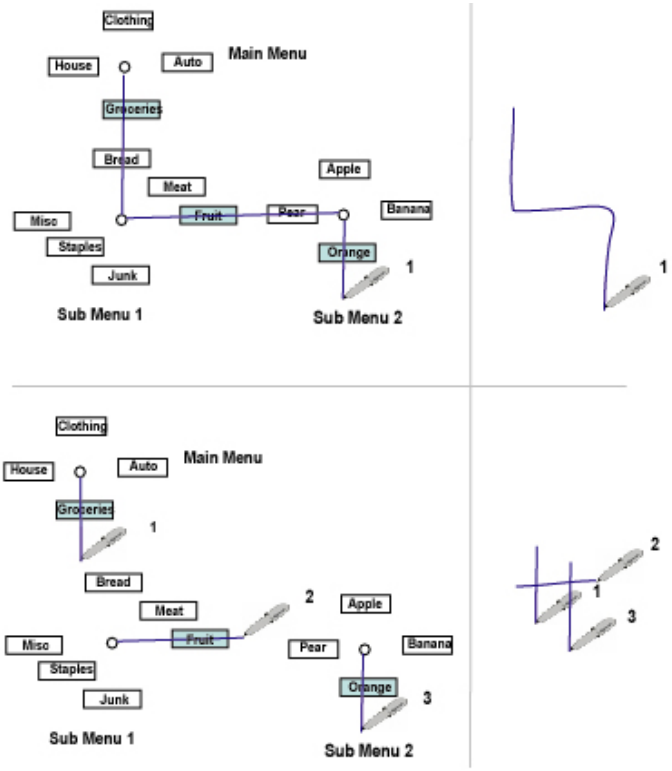


FIG. 2.14 – Reconnaissance de traces dans un menu circulaire hiérarchique (Illustration extraite de [Zhao et Balakrishnan, 2004])

qui peuvent se superposer et ainsi s'effectuer dans un espace plus réduit (en bas) [Zhao et Balakrishnan, 2004]. L'apprentissage nécessaire à une telle technique d'interaction est donc ici facilité par la transition naturelle d'un mode à l'autre.

Les menus circulaires ont été encore améliorés par Pook [2000], qui propose de prolonger le geste de sélection d'un élément d'un *marking menu* par un geste qui paramètre la commande choisie. Ainsi, un unique geste spécifie l'objet de la commande par son origine, la commande elle-même par la sélection dans le menu, et enfin les paramètres de la commande par la fin du geste. Pook [2000] propose par exemple d'appliquer son *control menu* à la navigation dans les espaces zoomables. Il dispose ainsi à gauche et à droite du menu les commandes qui permettent de dézoomer et de zoomer l'espace, la suite du mouvement du curseur fixant continûment le facteur de zoom (Figures 2.15 et 2.16). Ce type d'enchaînement entre sélection et paramètrage a été poussé encore plus loin par les *flow menu* [Guimbretière et Winograd, 2000] qui permettent d'enchaîner plusieurs gestes pour spécifier des interactions complexes, voire pour saisir du texte ou des valeurs numériques (la Figure 2.17 montre ainsi une interaction de zoom pour laquelle l'échelle est choisie numériquement).

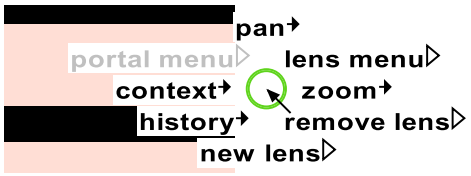


FIG. 2.15 – Un *control menu* pour (dé)zoomer

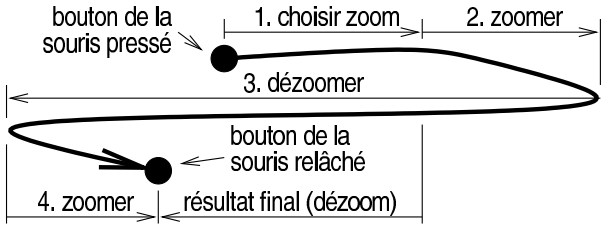


FIG. 2.16 – Principe du *control menu*

(Figures extraites de [Pook *et al.*, 2000])

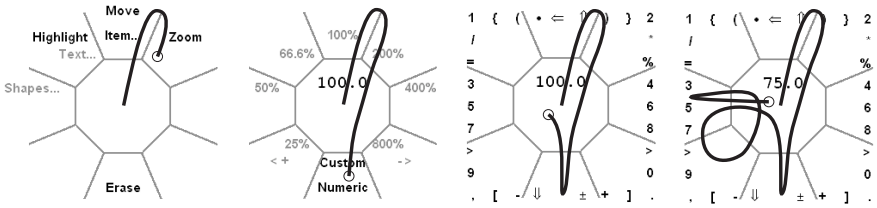


FIG. 2.17 – Une autre interaction pour zoomer à l'aide d'un *flow menu*

(Illustration extraite de [Guimbretière et Winograd, 2000])

2.2.2 Manipulation à l'aide de périphériques non-standard

On note qu'avec les techniques précédentes, s'amorce un mouvement qui s'éloigne progressivement des interactions classiques, localisées dans le temps et dans l'espace au sein d'une zone rectangulaire qui délimite l'interacteur, et se dirige vers des techniques d'interaction plus fluides et plus continues. Ce mouvement est particulièrement perceptible dès lors que l'on s'intéresse à des périphériques moins standards que la souris comme moyen d'interaction. L'émergence de terminaux mobiles, n'ayant pour périphériques d'entrée ni clavier, ni souris, mais un stylet, et celle de dispositifs d'affichage muraux, ont stimulé la recherche dans cette direction. De même, les plates-formes déviant de la norme clavier/souris par l'ajout d'autres périphériques, notamment l'utilisation conjointe de deux périphériques de pointage, offrent de nouvelles possibilités qui ont, elles aussi, été explorées et qui révèlent un fort potentiel.

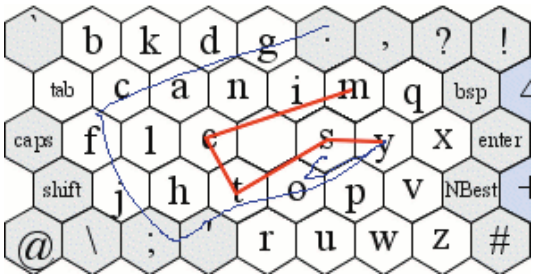


FIG. 2.18 – Le clavier virtuel Shark

(Illustration extraite de [Kristensson et Zhai, 2004])

Interactions pour le stylet

La miniaturisation des terminaux a ouvert la voie à leur mobilité. Toute une gamme de périphériques avec des tailles allant de celle de l'ordinateur portable à celle du téléphone mobile est maintenant offerte au grand public. Entre ces deux extrêmes, on trouve des dispositifs dont les fonctionnalités convergent petit à petit : mini ordinateurs portables, tablette numériques *(tablet PC)*, assistants numériques personnels (PDA), appareils photos, lecteurs de musique, de films, de photos. La plupart de ces dispositifs nécessitent l'entrée d'information de la part de l'utilisateur et, faute de place, ne peuvent proposer un clavier. Il en est de même pour les murs interactifs du fait de la position debout qui est requise pour travailler avec. Une alternative souvent proposée consiste en un écran tactile qui permet d'interagir avec la surface d'affichage en faisant de celle-ci un double usage : dispositif de sortie grâce à l'écran et dispositif d'entrée grâce à un stylet ou au doigt. Cette solution est particulièrement intéressante puisque idéalement, elle permet une réelle manipulation directe en supprimant l'indirection introduite par le pointeur de la souris.

Cependant, les premiers usages répandus de ce type de périphérique ont surtout cherché à proposer un substitut au clavier pour entrer du texte lettre à lettre. La première solution à ce problème consiste à afficher un clavier à l'écran et à l'utiliser avec le stylet. Ce système de "clavier virtuel" est peu satisfaisant car il utilise une grande partie de l'écran et propose des touches petites et donc difficiles à atteindre. Des agencements optimisés de touches ont été proposés pour les rendre plus efficaces (par exemple [Zhai *et al.*, 2000] propose de rapprocher les touches des lettres des digrammes fréquents). Il a aussi été proposé récemment d'utiliser un clavier comme support au geste pour baser la reconnaissance non sur des lettres isolées mais sur des mots complets grâce au clavier virtuel Shark [Zhai et Kristensson, 2003, Kristensson et Zhai, 2004] (la Figure 2.18 montre la trace utilisée pour entrer le mot "*system*" et la trace idéale correspondant à cette entrée).

Des solutions basées sur la reconnaissance de gestes réduisent la taille de la zone consacrée à la saisie tout en permettant à l'utilisateur de fixer son attention sur le document qu'il édite et non sur le clavier. Pour obtenir de bonnes performances sur des périphériques aux ressources limitées, des alphabets simplifiés constitués de traces (trait unique de stylo) ont été proposés,

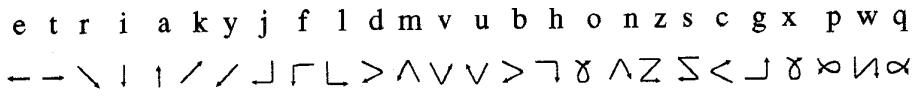


FIG. 2.19 – L'alphabet *Unistrokes*

(Figure extraite de [Goldberg et Richardson, 1993])

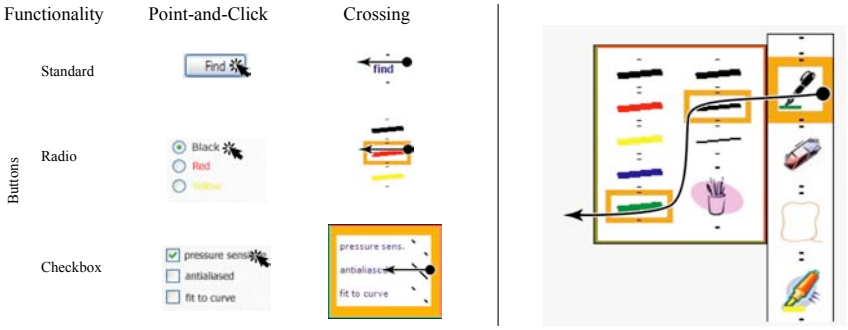


FIG. 2.20 – Quelques interacteurs de *CrossY*

(Figure extraite de [Apitz et Guimbretière, 2004])

par exemple le précurseur *Unistrokes* [Goldberg et Richardson, 1993] (voir la Figure 2.19), puis utilisés commercialement avec succès, comme l'alphabet *Graffiti* [Blickenstorfer, 1995] utilisé sur les *Palm Pilots*. Des améliorations à ce type de système sont constamment proposées, *EdgeWrite*, par exemple, utilise un guide physique contraignant les déplacements du stylet dans un rectangle pour faciliter la saisie, ce qui permet aussi de simplifier la reconnaissance des traces [Wobbrock *et al.*, 2003].

Alors que les applications précédentes tentent de fournir un substitut au clavier à l'aide du stylet, en voyant dans celui-ci un moyen d'interaction dégradé, des techniques d'interaction spécifiquement conçues pour tirer parti de ce périphérique ont été proposées. Le constat a été fait que les interacteurs traditionnels sont adaptés par leur conception à la manipulation à la souris mais peu à celle au stylet. En particulier l'action de cliquer, qui est la base des interactions de sélection, n'est pas aisée avec le stylet. Pourtant, les dispositifs comme les tablettes numériques utilisent les systèmes d'exploitation standards des machines de bureau. L'application *CrossY* [Apitz et Guimbretière, 2004] utilise le paradigme du franchissement (le *crossing*) pour remplacer celui du clic et présente un ensemble d'interacteurs de substitution pour les interacteurs habituels. Le franchissement, proposé originellement par Accot et Zhai [2002], consiste à traverser un segment matérialisé par ses deux extrémités pour déclencher une action plutôt que cliquer à l'intérieur d'une zone. La Figure 2.20 à gauche montre ainsi les équivalents des différents types de boutons pour le paradigme du franchissement. Un intérêt supplémentaire de ce paradigme est que les gestes peuvent être enchaînés, si l'interface a été conçue pour le permettre, et constituer ainsi des commandes complexes, permettant une interaction plus fluide (Figure 2.20 à droite).

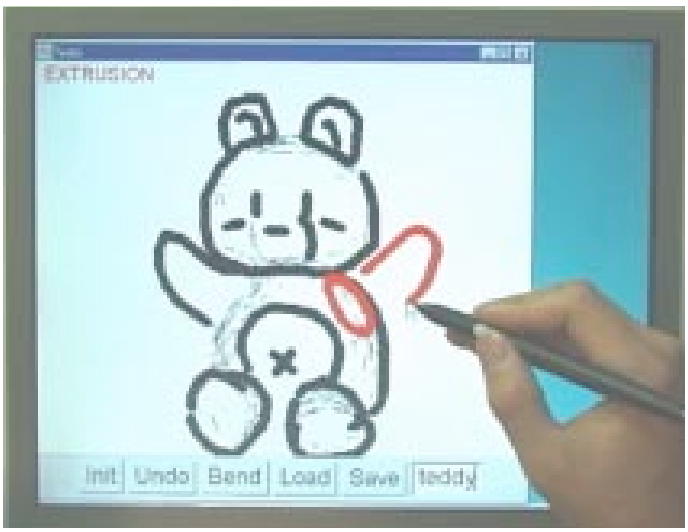


FIG. 2.21 – L'interface de *Teddy*

(Figure extraite de [Igarashi *et al.*, 1999])

Enfin, des travaux originaux proposent des techniques d'interaction complètement nouvelles adaptées au stylet. Le programme *Teddy* présenté par Igarashi *et al.* [1999] met ainsi en œuvre un ensemble de techniques d'interaction qui permettent de créer des modèles polygonaux tridimensionnels (3D) à l'aide de simples gestes. La Figure 2.21 montre l'interface de ce programme, dépourvue de tout interacteur classique pour les opérations 3D. L'utilisateur est en train d'ajouter un bras au modèle, en spécifiant par un tracé fermé sur l'objet, la surface de raccord, et par un second tracé, la silhouette voulue pour cette excroissance. La force de *Teddy* est de proposer un ensemble de techniques d'interaction, chacune très simple, formant un tout cohérent, reposant sur l'interprétation de gestes. Cette interprétation est dirigée par le contexte dans lequel est fait le geste et rend ainsi la manipulation complètement naturelle à l'utilisateur au bout de quelques minutes seulement.

Interaction bimanuelle

L'idée de permettre l'utilisation simultanée des deux mains pour effectuer des tâches est assez ancienne. On la trouve déjà dans les systèmes Sketchpad et NLS/Augment, présentés plus bas, qui datent tous deux des années soixante. Elle a été validée expérimentalement par Buxton et Myers [1986] pour vérifier que les utilisateurs étaient capables d'effectuer deux parties d'une tâche en parallèle. La tâche consistait à déplacer et redimensionner un carré pour le faire coïncider avec une cible (La Figure 2.22 montre à gauche la tâche et à droite l'appareillage utilisé pour l'expérimentation). L'expérience montre que les utilisateurs sont en effet capables de réaliser les deux sous-tâches, au moins partiellement, en parallèle.

Ce travail a trouvé une justification par les travaux de Guiard [1987] qui proposent un modèle de l'activité bimanuelle. En se basant sur ce travail théorique, des techniques d'interaction bimanuelles mettant en œuvre

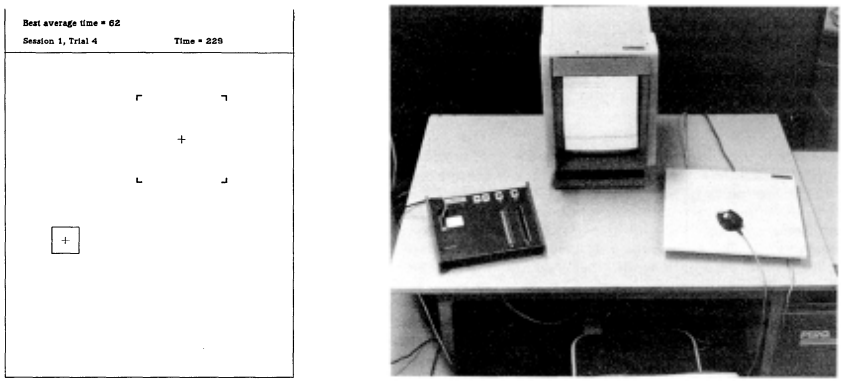


FIG. 2.22 – Une tâche bimanuelle de déplacement et de redimensionnement (Illustrations extraites de [Buxton et Myers, 1986])

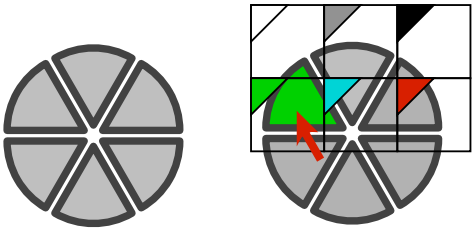


FIG. 2.23 – Modification d'une couleur à l'aide d'un outil transparent (Illustration extraite de [Bier *et al.*, 1993])

la collaboration entre les deux mains ont été proposées par [Bier *et al.*, 1993, Bier *et al.*, 1994] puis [Kabbash *et al.*, 1994, Kurtenbach *et al.*, 1997]. Ces techniques combinent l'utilisation de la transparence et de l'usage simultané des deux mains pour proposer des outils transparents *(see-through tools)*. Le principe général de ces outils est de proposer une palette partiellement transparente, manipulée par la main gauche, qui permet de superposer aux objets d'intérêt de l'application considérée des outils *(toolglass)* ou d'altérer leur présentation (lentille magique ou *magic lenses*). La main droite permet alors soit d'interagir avec le rendu altéré, soit de sélectionner dans une seule action la commande à appliquer et l'objet de cette commande en cliquant au travers d'un outil dans la palette transparente. La Figure 2.23 montre par exemple comment on peut choisir simultanément la couleur de remplissage et l'objet auquel on l'applique par un clic au travers d'une palette transparente.

2.2.3 Conclusion

Les techniques d'interaction présentées ici ont en commun de pratiquement toutes remettre en cause, certes à des degrés divers, le modèle établi qui consiste pour l'utilisateur à cliquer à l'intérieur d'une zone rectangulaire pour déclencher une action. On voit ainsi émerger des techniques pour lesquelles l'important n'est pas une position précise mais une succession de po-

sitions constituant une trajectoire, pour lesquelles ce n'est pas le fait d'être à l'intérieur ou à l'extérieur d'une zone qui est important mais le franchissement d'une frontière. S'ajoute donc à la localité des actions le besoin de prendre en compte leur temporalité pour donner un sens à une succession d'actions élémentaires.

Par ailleurs, on constate aussi que le modèle graphique des interacteurs classiques, constitués de rectangles emboîtés marque lui aussi le pas. Sortir de la forme rectangulaire est par exemple requis par les menus circulaires. De même, l'utilisation de modèles graphiques permettant de jouer sur les superpositions et la transparence, sur les transformations géométriques comme le changement d'échelle ou la rotation est essentielle pour la plupart de ces techniques d'interaction. Ces modèles graphiques étant de mieux en mieux supportés par le matériel lui-même par le biais des cartes accélératrices, il est nécessaire d'en faire bénéficier les concepteurs de techniques d'interaction et d'applications interactives.

Enfin, les dispositifs d'interaction se limitent de moins en moins à la configuration "standard" comportant un clavier, une souris et un écran pour une machine. Des dispositifs nomades, ne comportant qu'un stylet comme moyen d'interaction, aux ordinateurs portables de plus en plus souvent utilisés comme machines de bureau ayant alors souvent plusieurs écrans, une souris en plus du dispositif de pointage inclus avec la machine, les configurations possibles sont multiples. De même, les écrans muraux et les tables interactives se répandent et nécessitent eux aussi une adéquation des interactions avec le contexte dans lequel ils sont utilisés. Pour pouvoir tirer parti au maximum des capacités propres à chaque plate-forme, il faut fournir un support à l'accès à ces périphériques, ce qui permettra d'adapter l'interaction à leurs particularités.

2.3 Systèmes et boîtes à outils avancés

L'intérêt des techniques d'interaction présentées ci-dessus a souvent été démontré, même si c'est parfois sur des exemples d'utilisation particulièrement bien choisis pour mettre en valeur les avantages de chacune des techniques, comme le montre Appert *et al.* [2004]. Quelques systèmes ont cependant été réalisés qui intégrent plusieurs techniques d'interaction avancées, permettant ainsi de mieux se rendre compte de l'impact que peut avoir la conception complète d'une interface sans a priori WIMP sur l'utilisabilité d'une application.

Par ailleurs, l'expérience acquise lors de la réalisation de ces systèmes a permis de jeter les bases de la réalisation de boîtes à outils qui auraient permis, si elles avaient existé, de réaliser ces systèmes. Nous présentons ici quelques-uns de ces systèmes, puis un ensemble de boîtes à outils qui ont été proposées pour faciliter divers aspects du développement d'applications interactives post-WIMP.

2.3.1 Systèmes précurseurs

La mise au point d'outils pour le développement d'applications interactives nécessite de cerner les besoins de celles-ci. C'est ainsi que souvent des

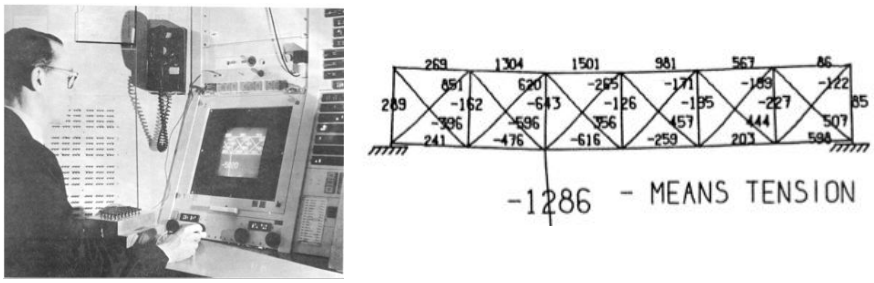


FIG. 2.24 – Le système *Sketchpad* utilisé par Sutherland

(Illustrations extraites de [Sutherland, 2003])

systèmes mettant en œuvre des interactions complexes ont été développés sans le support direct d'une boîte à outils, celles-ci faisant soit complètement défaut à l'époque, ou ne présentant pas les qualités requises pour réaliser certaines techniques d'interaction.

Systèmes historiques

Certains systèmes ont marqué l'histoire de l'interaction homme-machine par leur qualité visionnaire et les perspectives de recherche qu'ils ont ouvert. Les premiers ont vu le jour dans les années soixante, alors que l'interaction homme-machine se résumait à la perforation de cartes et à l'attente de la sortie des imprimantes.

Sketchpad. Fruit du travail de thèse de Ivan Sutherland le système *Sketchpad* date de 1963. Sa thèse a été rééditée en 2003 par l'université de Cambridge [Sutherland, 2003]. *Sketchpad* propose en particulier un système d'édition de schémas vectoriels utilisant la manipulation directe à l'aide d'un crayon optique et un système de résolution de contraintes pour réaliser diverses opérations de placement géométrique. La Figure 2.24 montre à gauche Ivan Sutherland utilisant *Sketchpad* et à droite un exemple de schéma utilisant le système de contraintes pour modéliser la résistance mécanique d'un objet). *Sketchpad* est sous doute le premier système interactif graphique, et il utilise déjà la manipulation directe.

NLS/Augment. Autre système marquant, *NLS/Augment* a été développé lui aussi dans les années soixante sous l'impulsion de Douglas Engelbart. Ce système a été le premier à utiliser certains objets ou concepts qui nous sont familiers de nos jours. Nous pouvons noter parmi ceux-ci la souris, une architecture client/serveur distribuée, ou encore la gestion de versions de documents [Engelbart, 1998]. Ce système a été conçu pour le travail collaboratif, y compris distant, avec l'intégration d'un lien vidéo entre les sites, la possibilité d'annoter les documents, la présence du courrier électronique. Un tel environnement reste pratiquement inégalé de nos jours.

FIG. 2.25 – Le système *NLS/Augment* en situation collaborative et ses périphériques

(Illustrations extraites d'une vidéo de démonstration de *NLS/Augment*[2])

La Figure 2.25 montre des images extraites d'une vidéo de Douglas Engelbart présentant l'ensemble des fonctionnalité de *NLS/Augment*[2]. À gauche, on note l'intégration d'un flux vidéo dans l'espace travail permettant de communiquer avec un collaborateur distant. On peut aussi discerner sur cette image la présence d'un télépointeur montrant l'activité de ce collaborateur. À droite, la Figure 2.25 montre les périphériques d'interaction du système. Outre le clavier, on note la présence d'une souris comportant trois boutons à la main droite, et à la main gauche un ensemble de 5 touches pour composer des accords permettant la sélection d'outils et de fonctions.

Xerox "Star". Au début des années quatre-vingts, un autre système allait marquer l'histoire : le *Xerox "Star"* (Johnson *et al.* [1989] proposent une rétrospective de ce système). Ce système, lui aussi fonctionnant en réseau et disposant d'une souris, offre une interface utilisateur graphique proposant en particulier un système de fenêtrage, et le WYSIWYG (*what you see is what you get* ou "vous obtiendrez ce que vous voyez"[3]). L'interface du *Xerox "Star"* repose sur la métaphore du bureau (Figure 2.26), et propose des icones pour représenter les objets d'intérêt — documents, boîtes de courrier, imprimantes — sur celui-ci. L'interaction est centrée sur les documents, permettant de composer à partir d'images, de formules, de tableaux et de texte des documents qui apparaissent à l'écran tels qu'ils seront imprimés.

Du point de vue de l'interaction, le *Star* propose un ensemble de commandes génériques qui peuvent s'appliquer à divers objets, accessibles grâce à des touches dédiées situées sur la gauche du clavier. L'interaction, utilisant la souris, le clavier et ces touches de fonctions, repose sur la manipulation directe et bimanuelle : les touches de fonctions modifient en effet les actions de la souris.

Systèmes récents

Plus récemment, d'autres systèmes ont introduit des paradigmes novateurs pour l'interaction ou ont dépassé le stade de la démonstration pour bâtir

[2] Cette vidéo, tournée en 1968, est accessible à l'adresse : `http://sloan.stanford.edu/mousesite/1968Demo.html`.

[3] Le WYSIWYG fait référence au fait que le document apparaît à l'écran tel qu'il sera une fois imprimé. Le *Star* permet en effet l'usage de fontes proportionnelles.

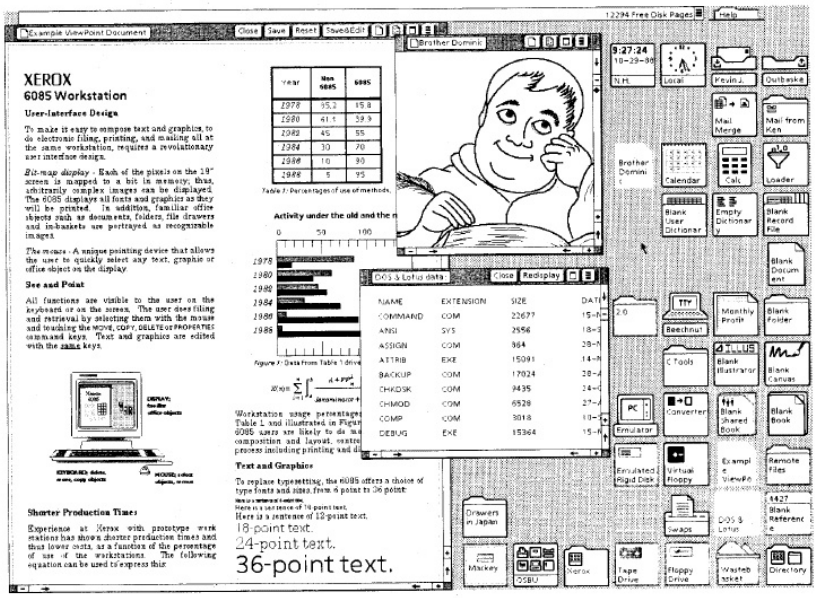


FIG. 2.26 – Le bureau du système *Xerox "Star"*
(Illustrations extraites de [Johnson *et al.*, 1989])

des applications complètement inédites en intégrant de manière fonctionnelle des travaux antérieurs. *CrossY* présenté avant (Figure 2.20, page 30) est un exemple d'une telle application démontrant la faisabilité d'un système ayant pour paradigme d'interaction la manipulation à l'aide du stylet et le franchissement comme moyen élémentaire de sélection plutôt que le clic.

DigiStrips. L'application DigiStrips [Mertz *et al.*, 2000], développée au Centre d'Étude de la Navigation Aérienne, est un prototype qui explore l'usage des écrans tactiles et l'impact du design graphique pour les interfaces dédiées au contrôle aérien. Cette application permet de gérer électroniquement les bandes de papiers *(strips)* qui représentent les vols et leurs paramètres. L'interaction est basée sur la reconnaissance de gestes et sur la présence d'un retour visuel élaboré. Ce dernier, constitué en particulier d'animations, a été conçu pour favoriser la conscience de groupe, le contrôle aérien étant une activité fortement collaborative. La Figure 2.27 montre par exemple les étapes de l'animation utilisée lors de la sélection d'une nouvelle valeur de cap dans un menu. En particulier, le nouveau cap est mis en valeur par une animation après sa sélection (au centre), ce qui permet de confirmer à l'utilisateur son choix.

CPN2000. L'application CPN2000 est le fruit d'un travail de reconception de l'interface d'un logiciel existant qui offrait un environnement pour l'édition et l'exécution de réseaux de Petri colorés [Beaudouin-Lafon et Lassen, 2000]. Les interactions proposées par CPN2000 intègrent des techniques diverses (outils transparents, interaction bimanuelle, menus contextuels circulaires, espaces

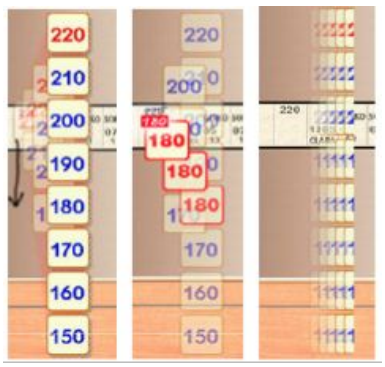


FIG. 2.27 – Un menu animé de DigiStrips

(Illustration extraite de [Mertz *et al.*, 2000])

zoomables) au sein d'une interface cohérente conçue en utilisant les trois principes pour la conception d'interfaces mis en évidence par Beaudouin-Lafon et Mackay [2000] : la réification, le polymorphisme, et la réutilisation. La Figure 1.3, page 4, présente l'interface de CPN2000 et illustre quelques-unes des techniques d'interaction que celle-ci propose.

2.3.2 Boîtes à outils pour l'interaction

Ces quelques exemples d'applications montrent que lorsque l'expressivité du modèle proposé aux concepteurs d'applications n'est pas bridée par des choix de conception faits a priori par les créateurs de boîtes à outils, les interfaces qu'il est possible de réaliser peuvent être adaptées finement à leurs usages et à leurs utilisateurs. Néanmoins, réaliser des applications en repartant de zéro est extrêmement coûteux et ne permet pas de capitaliser sur les réalisations passées. Le seul moyen de mutualiser les solutions à des problèmes communs est l'usage de bibliothèques et de boîtes à outils. De nombreux travaux de recherche proposent ainsi des boîtes à outils pour développer des applications interactives.

Nous présentons ici des boîtes à outils qui, d'une manière ou d'une autre, essaient de proposer des modèles d'interface plus élaborés que celui de l'assemblage d'interacteurs proposé par la plupart des boîtes à outils d'interface usuelles, soit en se focalisant sur un domaine particulier d'application, soit en proposant d'explorer de nouveaux paradigmes de programmation.

Boîtes à outils généralistes

subArctic. La boîte à outils *subArctic*[4] est écrite en Java et propose des mécanismes avancés pour gérer l'affichage, notamment les animations, les outils transparents [Hudson *et al.*, 1997]. Elle propose aussi un système de contraintes permettant de placer facilement les objets graphiques de l'interface [Edwards *et al.*, 1997]. *subArctic* est le successeur de *Artkit*, développée en C++, et elle en reprend plusieurs caractéristiques, notamment son

[4] La boîte à outils *subArctic* et sa documentation sont disponibles à l'adresse suivante : `http://www.cc.gatech.edu/gvu/ui/sub_arctic/` .

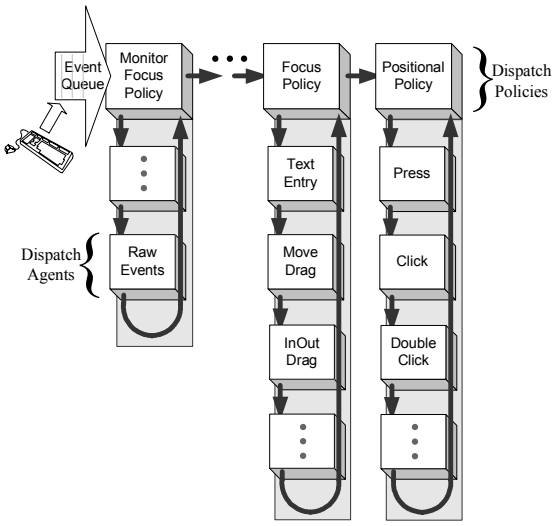


FIG. 2.28 – Le modèle de gestion des événements de *subArctic*

(Illustration extraite de [Hudson *et al.*, 2005])

système élaboré de gestion des événements en entrée [Henry *et al.*, 1990, Hudson *et al.*, 2005]. Ce système repose sur une cascade de politiques d'aiguillage qui permettent de diriger les événements vers les objets de l'interface désignés par exemple par le focus, ou encore par la position du curseur. Ces politiques de distribution des événements sont représentées sur la Figure 2.28 par les éléments de la ligne supérieure. Chacune de ces politiques comporte une liste ordonnée d'agents d'aiguillage, représentés par les éléments disposés verticalement sous chaque politique sur le schéma.

Ces agents essaient d'interpréter tour à tour les événements provenant des périphériques, et s'ils ne les concernent pas, ils les passent à l'agent suivant. Ce processus est répété par chacune des politiques tant que les événements n'ont pas été interprétés. Les flèches symbolisent donc sur le schéma le cheminement des événements de politique en politique et d'agent en agent. La première politique (à gauche sur le schéma) diffère des suivantes en ce qu'elle ne consomme pas les événements, mais introduit dans le cycle des agents qui permettent d'observer les événements transitant dans le système, et ce à des fins de mise au point.

OpenDPI/DoPIdom. DPI est un modèle introduit par Beaudoux [2004] qui met au cœur des systèmes interactifs les documents et non plus les applications. Il a mis en œuvre ce modèle avec les boîtes à outils OpenDPI, puis DoPIdom qui lui a succédé. Cette boîte à outils propose des outils génériques qui peuvent agir sur n'importe quel objet compatible d'un document. Elle offre aussi des mécanismes avancés pour supporter la collaboration par le biais de points de partage qui permettent de répliquer avec un couplage plus ou moins fort, les objets édités [Beaudoux et Beaudouin-Lafon, 2001, Beaudoux et Beaudouin-Lafon, 2005].

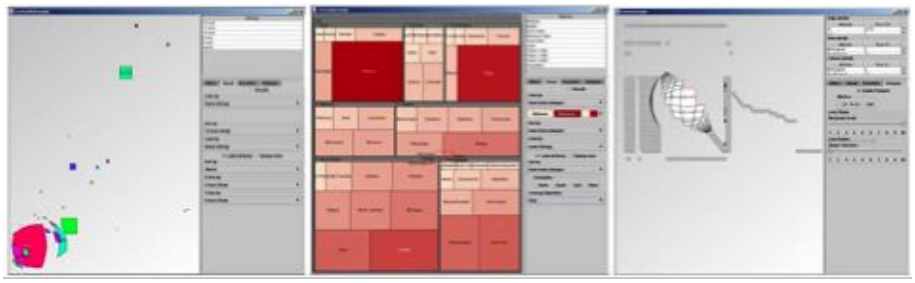


FIG. 2.29 – Exemples de visualisation obtenue avec la boîte à outils *InfoVis* (Illustrations extraites de [Fekete, 2004])

Boîte à outils dédiées

Certaines boîtes à outils ont été créées pour répondre à des besoins particuliers et se focalisent dessus. Assez logiquement, ces boîtes à outils se concentrent sur les aspects fortement limités par les boîtes à outils traditionnelles : le modèle graphique proposé en sortie, ou le modèle de périphériques proposé en entrée.

Espaces zoomables. La boîte à outils *Pad++* et ses descendantes *Jazz* puis *Piccolo* proposent ainsi un cadre pour créer des interfaces zoomables [Bederson et Hollan, 1994, Bederson *et al.*, 2000, Bederson *et al.*, 2004]. Ce paradigme graphique permet d'afficher de grandes quantités d'information en structurant l'affichage à travers différents niveaux de détails. Il requiert des techniques d'interaction appropriées pour pouvoir naviguer dans l'espace a priori infini présenté à l'utilisateur, et ce tant en position qu'en échelle. Par ailleurs, pour obtenir de bonnes performances graphiques, les interfaces zoomables nécessitent l'utilisation de niveaux de détails adaptés à la taille apparente des objets. Ces mécanismes sont fournis par la boîte à outils.

De même, la boîte à outils ZVTM *(Zoomable Visual Transformation Machine)* de Pietriga [2005], facilite l'utilisation de ce type de techniques dans le domaine particulier des environnements de programmation visuelle.

Visualisation d'information. La visualisation interactive d'information comporte elle aussi des problématiques propres auxquelles les boîtes à outils peuvent apporter des solutions génériques. Pouvoir projeter facilement les propriétés des objets visualisés sur les attributs graphiques de leurs représentations, filtrer interactivement les données, proposer des agencements pertinents sont autant de fonctionnalités qu'offrent les boîtes à outils *InfoVis* de Fekete [2004] ou *prefuse* de Heer *et al.* [2005]. La Figure 2.29 montre ainsi quelques visualisations construites avec la boîte à outils *InfoVis*.

Gestion d'entrées multiples. Un autre aspect important des nouvelles techniques d'interaction est leur adaptation aux périphériques d'entrée. Que ce soit par l'usage d'un stylet comme périphérique de pointage, ou par l'usage simultané de plusieurs périphériques pour mettre en œuvre des interactions bimanuelles, l'utilisation de périphériques non conventionnels nécessite une

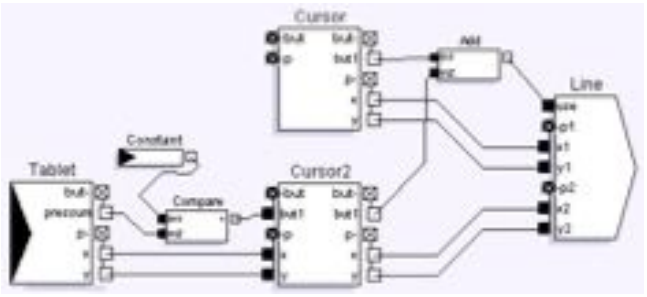


FIG. 2.30 – Une configuration bimanuelle spécifiée par ICON
(Illustration extraite de [Dragicevic et Fekete, 2001])

prise en compte adaptée pour offrir des posibilités intéressantes. Peu de boîtes à outils offrent ce support, et la plupart des systèmes tendent au contraire à gommer les spécificités des périphériques d'interaction pour utiliser les mêmes interactions quels que soient les dispositifs d'entrée disponibles.

La boîte à outils ICON de Dragicevic et Fekete [2001] va à l'encontre de cette démarche puisqu'elle propose de découpler totalement les applications des périphériques d'entrée qui permettent de réaliser les interactions. Ces dernières sont adaptables aux configurations d'entrée grâce à un outil spécialisé qui permet de spécifier les techniques d'interaction par un formalisme réactif. La Figure 2.30 montre par exemple un configuration de ICON adaptant un stylet et la souris du système en entrée pour permettre de tracer des lignes de manière bimanuelle dans un éditeur graphique.

Le serveur de souris (UMS pour *Ubit Mouse Server*) de Lecolinet permet quant à lui d'utiliser simultanément plusieurs souris sur un ou plusieurs systèmes et d'explorer ainsi les usages de différentes configurations comme par exemple un utilisateur unique avec deux périphériques — il s'agit alors d'interaction bimanuelle —, ou encore plusieurs utilisateurs (distants ou non) avec chacun un périphérique — il s'agit d'usages collaboratifs.

Nouveaux idiomes pour la programmation

Ce serveur de souris est intégré à une boîte à outils complète, Ubit *(Ubiquitous Brick Interaction Toolkit)* qui propose par ailleurs un modèle original pour créer des interfaces [Lecolinet, 2003]. Ubit substitue à l'arbre traditionnel d'interacteurs un graphe acyclique dont les nœuds ont une granularité plus fine. Ceux-ci peuvent être du texte, des images, ou encore des attributs graphiques. Par composition, ils permettent de recréer les interacteurs traditionnels mais aussi tout une panoplie de nouveaux interacteurs. Par ailleurs, la structure introduite par le graphe permet de réutiliser des éléments de celui-ci, tout en pouvant appliquer aux divers exemplaires des transformations ou des styles différents. Il est ainsi par exemple facile de créer des vues différentes mais synchrones sur les mêmes objets. Outre ce modèle graphique original, Ubit propose par ailleurs de simplifier le code nécessaire à la spécification de l'interface. Cette simplification passe par l'introduction d'un idiome déclaratif, supporté par la possibilité de surcharger les opérateurs, au sein du langage impératif, le C++, utilisé par la boîte à outils.

Cette approche, l'introduction d'un langage dédié, permet en fait de contourner un obstacle réel sur lequel butent beaucoup d'outils : les langages de programmation impératifs ne sont pas adaptés à l'interaction. Leur utilisation introduit une lourdeur syntaxique nécessaire pour faire coïncider les structures du langage avec celles de l'interaction. Historiquement, l'évolution des concepts liés à la programmation a souvent été liée à celle des langages de programmation. Ainsi, le modèle MVC est associé à Smalltalk. De même, la boîte à outils Garnet [Myers *et al.*, 1990] est liée au langage Lisp enrichi d'un modèle à objets reposant sur un mécanisme prototypes/instance, [Myers *et al.*, 1992]. Même la boîte à outils Amulet [Myers *et al.*, 1997], qui a succédé à Garnet, a conservé le modèle prototype/instance de son prédécesseur bien qu'ayant abandonné le Lisp.

Plus récemment, des boîtes à outils comme Intuikit [Chatty *et al.*, 2004] ou MaggLite [Huot *et al.*, 2004] associent à un graphe de scène décrivant l'interface une structure permettant de spécifier l'interaction. Dans le cas d'Intuikit, il s'agit de machines à états qui sont insérées dans le graphe de scène et qui associent un comportement aux nœuds concernés. Dans MaggLite, l'interaction est gérée dans un graphe séparé de celui de l'affichage constitué du modèle réactif des périphériques et des interactions proposé par ICON. Des connexions entre les deux graphes se créent alors dynamiquement, en fonction de l'évolution de l'interaction.

2.3.3 Conclusion

Nous avons vu que les techniques d'interaction avancées remettent en cause les modèles courants, basés essentiellement sur les interacteurs classiques. Ces techniques, qui avaient démontré leurs intérêts individuellement, se révèlent en pratique, utilisées conjointement et à bon escient, très prometteuses. Les divers systèmes les utilisant l'attestent. Comment, alors, faciliter leur utilisation pour la création de nouvelles applications ?

La réponse du génie logiciel à cette question est le principe de la réutilisation et passe par l'intégration de ces techniques à des boîtes à outils. Cette approche a produit des résultats eux aussi très prometteurs puisqu'ils permettent de créer des applications utilisant de nouveaux paradigmes et de les tester ainsi en vraie grandeur. Cependant, il ne faut pas perdre de vue que l'utilisation de boîtes à outils (certes de conception plus ancienne) est précisément l'obstacle actuel à l'adoption de techniques d'interaction nouvelles, puisqu'elles enferment les développeurs dans des choix de conceptions réalisés par ceux qui ont créé la boîte à outils. C'est donc cette tension entre l'utilisation d'une boîte à outils, qui permet de mutualiser l'existant, d'une part, et la possibilité d'utiliser des paradigmes complétement inédits, d'autre part, qu'il faut s'attacher à réduire.

2.4 Feuille de route

Cet objectif ambitieux nécessite l'identification et la résolution d'un certain nombre de problèmes précis. Il nous faut par ailleurs établir des critères permettant d'évaluer la réussite de notre entreprise.

2.4.1 Trois directions

Nous avons choisi de concentrer nos efforts sur trois aspects pour fournir des outils adaptés au développement d'interfaces hommes-machines avancées :

- le choix d'un modèle graphique élaboré laissant libre cours à la créativité et facilitant le prototypage de nouvelles techniques ;
- la proposition d'un modèle des périphériques d'entrée adapté à leur emploi dans des interactions complexes ; et
- la mise en place d'un support explicite, au niveau du langage de programmation, à la création d'interactions avancées.

Pour le modèle graphique, nous avons décidé d'adopter SVG *(Scalable Vector Graphics)*. SVG est un format graphique vectoriel et structuré. Il permet en particulier l'utilisation de formes vectorielles arbitraires, de texte, de dégradés, d'images. Il permet aussi les transformations géométriques, l'usage de la transparence. Il permet de plus de structurer les graphiques au sein d'un graphe de scène. Il se limite cependant à un monde bidimensionnel. Cependant le format SVG dispose d'atouts majeurs pour nous : c'est un format que les outils professionnels des graphistes comme Adobe Illustrator sont capables de produire : la spécification de SVG concerne le format lui-même mais aussi l'API qui permet de le manipuler ; et enfin, il existe des bibliothèques permettant son rendu de manière suffisamment rapide pour permettre l'interaction.

Pour les deux autres points, nous avons proposé des solutions originales qui sont détaillées dans les chapitres 3 et 4.

2.4.2 Cahier des charges

Par ailleurs, et de manière orthogonale aux trois objectifs principaux que nous avons définis ci-dessus, il faut apporter une réponse aux exigences a priori contradictoires que nous fixons à notre boîte à outils :

- la réutilisation aisée de l'existant qui est la raison d'être d'une boîte à outils ; et
- la capacité de supporter des développements non anticipés en n'enfermant pas dans un modèle trop étroit d'applications le programmeur.

Pour concilier ces exigences, nous avons décidé de fournir différents niveaux d'abstraction aux programmeurs. Ainsi, c'est à eux que revient le choix du compromis réutilisation/réimplémentation auquel ils se placent. Si à un niveau, les objets proposés ne correspondent pas exactement aux besoins, il est toujours possible de les recomposer à partir de briques plus élémentaires, en réutilisant les parties qui conviennent, et en adaptant les autres.

Comment vérifier alors que cette approche est bonne ? Le premier test à passer est bien entendu celui de la non-régression, c'est-à-dire vérifier dans notre cas que nous pouvons au moins reproduire l'existant à l'aide de notre boîte à outils. Nous montrons en effet que nous pouvons recréer l'état de l'art des techniques d'interaction, tant WIMP que post-WIMP. Cependant cela n'est pas suffisant, notre objectif étant de permettre l'innovation. Nous montrons donc de plus que notre boîte à outils est adaptée au prototypage de nouvelles techniques d'interaction avancées. Ces démonstrations font l'objet des chapitres 5 et 6.

Chapitre 3

L'abstraction du système

Nous présentons dans ce chapitre les abstractions fournies par la boîte à outils HsmTk. Ces abstractions décrivent le système depuis les abstractions pertinentes pour les utilisateurs (périphériques d'entrée et de sortie), jusqu'au niveau le plus bas pertinent pour le programmeur (modèles de concurrence et objets de communication et de synchronisation).

Nous avons vu que les interactions deviennent de plus en plus fluides pour être de plus en plus efficaces et s'adapter à de nouveaux contextes. Pour parvenir à cette fluidité, des mécanismes précis et bien réglés doivent être mis en jeu. C'est en effet souvent dans le réglage des détails que se jouent l'efficacité et l'utilisabilité des techniques d'interaction. Proposer au programmeur l'interacteur comme niveau d'abstraction lui facilite la tâche mais enferme sa production dans un stéréotype d'interaction, certes universel et polyvalent, mais souvent inadapté à l'application particulière qu'il construit. Nous proposons plutôt de mettre à sa disposition un ensemble d'abstractions de différents niveaux, donnant accès aux fonctionnalités de la plate-forme, des plus basses aux plus élaborées. Ces abstractions sont construites les unes au-dessus des autres et permettent au programmeur de choisir lui-même le compromis qu'il est prêt à faire, entre réutilisation de solution générique et développement de techniques spécifiques. Si un aspect de haut niveau ne lui convient pas, il peut

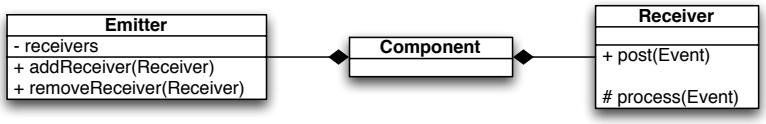


FIG. 3.1 – Aspects émetteurs et récepteurs des composants

ainsi le recomposer en utilisant un niveau inférieur d'abstraction. Il sera d'autant plus enclin à le faire que le résultat de son travail sera lui aussi réutilisable pour d'autres projets.

La brique élémentaire proposée par la boîte à outils HsmTk est un composant logiciel qui réalise les fonctions de communication avec les autres composants. Il propose un mécanisme de composition qui permet de créer des arbres de composants. Les objets complexes peuvent ainsi être décomposés en sous-éléments. Ce composant est utilisé comme base pour proposer un ensemble d'abstractions élémentaires et génériques que nous présentons ici. Nous présentons ensuite les divers objets plus proches des utilisateurs qui sont construits à l'aide de ces abstractions élémentaires. Ainsi, une représentation structurée des périphériques d'entrée et de sortie est fournie. C'est à ce niveau aussi que la boîte à outils offre un support direct à l'interaction en explicitant la notion de protocole d'interaction, et en proposant des mécanismes permettant de les réaliser facilement. Enfin, nous présentons les abstractions de plus bas niveau sur lesquelles l'ensemble de la boîte à outils repose. Ces abstractions ont pour but de s'affranchir des particularités des divers systèmes d'exploitation et de fenêtrage sans pour autant interdire au programmeur l'accès aux fonctionnalités les plus fines qu'ils permettent.

3.1 Abstractions élémentaires

3.1.1 Composant de base

HsmTk propose un composant de base qu'elle utilise comme fondation pour échafauder l'ensemble de ses propres abstractions. Ce composant est un objet doté d'un système de notification par événements. Il comporte deux aspects : l'*émetteur* d'événements et le *récepteur*. L'émetteur fournit une interface pour permettre aux autres composants de s'abonner et de se désabonner à la notification de ses événements. Il maintient pour cela une liste des composants à qui il distribue ses notifications. À l'autre extrémité, le récepteur offre une interface permettant aux autres composants de lui communiquer des événements. La Figure 3.1 montre comment les composants (objets de classe `Component`) sont composés de ces deux aspects représentés par les classes `Emitter` et `Receiver`.

Politiques de réception

Plusieurs politiques de réception des événements sont proposées au programmeur. Les événements peuvent être pris en compte de manière purement *synchrone*, dans le même fil d'exécution que leur envoi. Dans ce cas, l'émetteur ne reprend la main qu'une fois que le récepteur a traité l'événement et ce mode

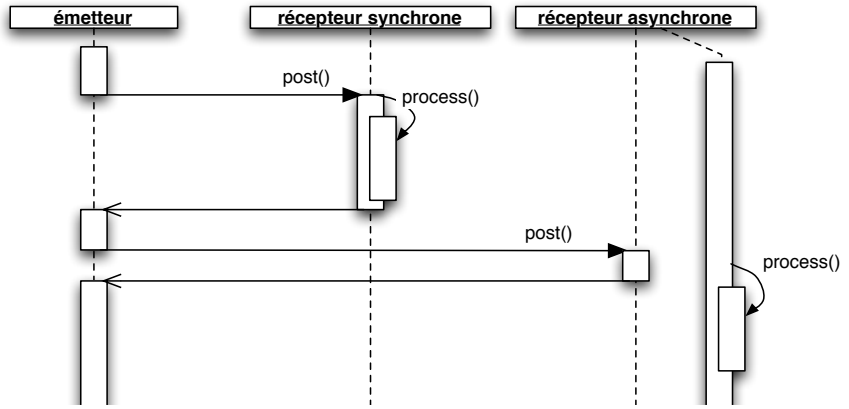


FIG. 3.2 – Séquencement du traitement des événements pour les politiques synchrone et asynchrone

de communication revient à un passage de message ou à l'invocation d'une méthode pour ce qui est de la synchronisation entre les deux parties. Cependant, contrairement à l'appel de méthode, c'est le récepteur qui a l'initiative de l'abonnement, l'émetteur ne le connaît donc pas a priori ; et la notification est à destination d'un ensemble de récepteurs, et non d'un unique objet. La Figure 3.2 illustre cette séquence de traitement de l'événement envoyé par l'émetteur à gauche vers le récepteur synchrone situé au centre.

Les événements peuvent aussi être pris en compte de manière *asynchrone*. Ils sont alors mis par l'émetteur dans une queue propre à chaque récepteur. Un fil d'exécution spécifique au récepteur se charge de scruter cette queue et de traiter chaque événement lorsqu'il se présente. Comme le traitement d'un événement par le récepteur n'appelle pas le retour d'un résultat à l'émetteur celui-ci peut poursuivre son fil d'exécution sans attendre. Cette séquence est illustrée sur la Figure 3.2 par le traitement de l'événement provenant de l'émetteur à gauche à destination du récepteur asynchrone situé à droite.

Enfin, une troisième politique, variante de la queue asynchrone, est proposée. Elle consiste à n'ajouter l'événement à la queue que si celle-ci ne comporte pas déjà en attente de traitement un événement de même type déposé par le même émetteur. Cette politique peut être pratique si l'événement est susceptible de déclencher des traitements coûteux. Un exemple d'utilisation pour lequel un tel mécanisme est utile est la demande de mise à jour du rendu d'une fenêtre. Si une telle demande est déjà présente dans la queue de la fenêtre, cela n'est pas utile d'en insérer une nouvelle.

Ces politiques se combinent avec les composants par simple composition lors de leur définition ou à leur instanciation, chaque récepteur pouvant ainsi être doté d'une politique particulière. Le programmeur peut par ailleurs enrichir à volonté les politiques existantes en les optimisant selon ses besoins. Ces nouvelles politiques peuvent être utilisées aussi pour les composants préexistants. Cette orthogonalité entre les politiques de traitement des événements et les traitements eux-mêmes encourage la réutilisation du code.

Traitement des événements

Le programmeur doit par ailleurs préciser le traitement à effectuer pour les événements reçus. Comme pour les politiques de gestion des événements, le traitement peut être délégué par composition à un autre objet (typiquement, un objet qui encapsule la notion de fonction de rappel). Plus généralement, il peut aussi être spécifié par dérivation et surcharge de méthode, ce qui permet de modifier l'objet en profondeur.

Le mécanisme de traitement des événements proposé par HsmTk est donc très modulaire et permet par le choix des politiques de traitement et par le choix du mécanisme de traitement lui-même d'obtenir la plupart des modèles proposés par les boîtes à outils existantes. On peut ainsi reproduire le mécanisme des fonctions de rappels *(callbacks)*, mais aussi déléguer la gestion de certains événements à des objets à part entière comme le sont les *listeners* de Java par exemple. Le choix des politiques de réception des événements par composant et leur extensibilité permet de plus de mettre en œuvre des mécanismes plus finement adaptés à chaque besoin.

3.1.2 Composant composite

La notion de composant permet de structurer la logique des applications en offrant un support à leur décomposition en unités formant des ensembles cohérents du point de vue fonctionnel. Cependant, comme l'illustre la plupart des méthodes d'analyse et de conception, cette décomposition est souvent récursive, ce qui permet de descendre dans les détails sans être submergé par la complexité de l'ensemble du système. Pour permettre une structuration des composants plus forte que celle introduite par les relations d'abonnement aux événements, les composants peuvent être regroupés au sein d'un composant de plus haut niveau : le composant composite.

Cette structure est récursive et permet ainsi de créer des arbres de composants de profondeur arbitraire. Au sein de cette structure, chaque composant nœud est abonné automatiquement à l'ensemble de ses fils. La boîte à outils HsmTk met par exemple à disposition du programmeur, comme nous le détaillons à la Section 3.2.1, une représentation structurée de la souris : elle peut être vue comme l'assemblage d'une position, d'un ensemble de boutons, et de la valeur de la molette si celle-ci est présente. Si seule la position de la souris est intéressante pour le programmeur, il peut ne considérer que ce sous-ensemble du composant représentant la souris. Plus finement, si seul le déplacement le long de l'axe droite/gauche est pertinent pour la technique d'interaction qu'il réalise, il peut ne s'intéresser qu'à cette sous-partie de la position et ne recevoir ainsi que les notifications concernant le déplacement choisi.

3.1.3 Valeurs actives

Au niveau d'abstraction le plus simple, la boîte à outils offre des valeurs actives (des grandeurs numériques ou de simples booléens) qui notifient les composants abonnés lorsqu'elles changent de valeur. Il est possible de fixer une résolution aux valeurs numériques, ce qui permet de ne recevoir de notifications que si la modification dépasse un certain seuil en valeur ab-

solue. Ainsi, pour des périphériques dont le signal est bruité, la quantité d'événements produits est réduite en fixant un seuil supérieur au niveau du bruit. Cela est particulièrement utile pour les périphériques liés aux tablettes graphiques dont la résolution est très supérieure à celle de l'écran. C'est le programmeur qui choisit alors un niveau de compromis : en rester à la résolution de l'écran et réduire la quantité d'événements à transmettre et à traiter, ou garder la résolution maximale de la tablette si elle est nécessaire à son application particulière.

Par composition, ces valeurs actives sont regroupées, offrant ainsi la notion d'enregistrement ou de "*record*". Par exemple, on dispose ainsi de points (grandeurs à plusieurs dimensions) et de "boîtes à boutons" (qui permettent de regrouper les boutons de la souris ou l'ensemble des touches d'un clavier). Le principal intérêt de ces compositions, outre qu'elles permettent de structurer logiquement les composants, réside en ce qu'elles permettent de contrôler la granularité des notifications. Ainsi, alors qu'à chaque déplacement de la souris ou presque, ses deux coordonnées sont modifiées, une seule notification sera envoyée aux composants abonnés au niveau de la position de la souris.

3.1.4 Machines à états hiérarchiques

Les machines à états hiérarchiques, à la description détaillée desquelles le Chapitre 4 est consacré, sont mises en œuvre par composition de composants élémentaires spécialisés, et s'appuient sur les mécanismes de ces composants élémentaires : notification et composition hiérarchique.

La notification est utilisée par les machines à états hiérarchiques pour communiquer avec l'extérieur, tant pour recevoir les modifications de ses entrées que pour notifier ses changements d'états. Les machines à états hiérarchiques peuvent ainsi être utilisées pour réaliser des fonctions de traitement et de filtrage sur les événements provenant de leurs entrées. La composition des composants est, elle, naturellement utilisée pour structurer les machines à états de manière hiérarchique et réaliser leur structure d'arbre.

3.2 Périphériques et interaction

Pour échafauder une application interactive, les abstractions élémentaires présentées ci-dessus ne suffisent pas. Par sa nature même, une application interactive répond aux entrées de l'utilisateur, les interprète pour effectuer des actions, et fournit à l'utilisateur un retour reflétant l'état interne de l'application et son évolution. Comme nous l'avons vu à la Section 2.2, les techniques d'interaction post-WIMP sont parfois adaptées à des configurations particulières de dispositifs d'entrée et nécessitent donc une représentation assez fine des entrées au sein de l'application pour être réalisées. De même, elles nécessitent aussi souvent un modèle graphique élaboré pour être mises en œuvre.

Nous présentons ici les abstractions des périphériques d'entrée et de sortie mises à disposition du programmeur par la boîte à outils HsmTk. Ensuite, nous présentons la structure qui est proposée pour établir le lien entre les entrées et les sorties, c'est-à-dire pour supporter l'interaction proprement dite.

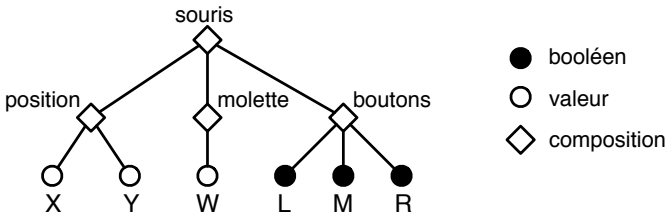


FIG. 3.3 – Représentation d'une souris comportant une molette et trois boutons.

3.2.1 Périphériques d'entrée

Les périphériques d'entrée sont accessibles au sein des applications par l'intermédiaire d'une représentation arborescente. Card propose dans son exploration de l'espace de conception des périphériques d'entrée [Card *et al.*, 1990] trois opérateurs pour structurer leurs descriptions : la composition *(merge),* le positionnement *(layout)* et la connexion *(connect)*. Pour nous l'aspect connexion, qui permet par exemple de relier la position de la souris à celle du curseur, ne fait pas partie du périphérique d'entrée mais de la spécification de l'interaction. Nous décrivons en particulier au Chapitre 6 le pointage sémantique, une technique d'interaction qui joue sur le couplage entre souris et curseur pour faciliter la tâche de pointage

Par ailleurs, nous ne faisons pas la distinction entre composition (réunir le x et le y de la souris en une position) et le positionnement (réunir la position de la souris et ses boutons au sein d'un même périphérique) car, si elle est pertinente pour explorer l'espace de conception des périphériques d'entrée, la composition et le positionnement ont le même effet pour la description des périphériques.

Structure hiérarchique

Les périphériques sont représentés de manière arborescente en assemblant des valeurs actives pour constituer des points, ou des ensembles de touches et de boutons. Ces assemblages peuvent être regroupés pour constituer des ensembles de plus haut niveau (voir par exemple pour une souris comportant une molette et trois boutons la Figure 3.3). Cette structuration utilise directement le mécanisme de composition de la boîte à outils.

Le support offert par la boîte à outils HsmTk pour les périphériques repose sur une architecture modulaire : les modules gérant un type particulier de périphériques sont chargés pendant l'exécution des programmes à leur demande (cette fonctionnalité repose sur le service de bas niveau permettant le chargement dynamique de modules détaillé à la Section 3.3.2). Ainsi, il est aisé d'ajouter le support pour un périphérique particulier : seul un nouveau module a besoin d'être développé et en l'ajoutant à la bibliothèque des modules, toutes les applications peuvent en tirer parti. La boîte à outils offre comme modules standards une représentation du clavier et de la souris par défaut du système. Ainsi, par exemple, pour utiliser la souris du système, il suffit de charger le module correspondant :

```
hsm::Device *mouse = hsm::repository::getDevice("Mouse");
```

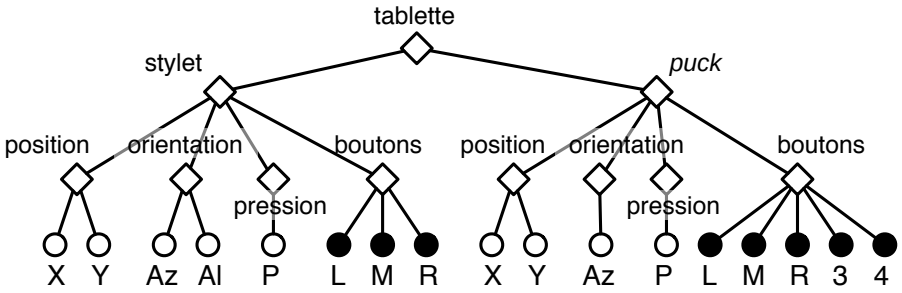


FIG. 3.4 – Tablette supportant deux dispositifs.

Grâce à l'opérateur / on peut alors accéder aux divers éléments de la souris en indiquant leur chemin dans l'arbre des périphériques. Ainsi, pour récupérer l'abscisse de la souris, on peut écrire :

```
float x = *mouse/"position"/hsm::Point::X;
```

Par ailleurs, la boîte à outils permet de gérer en standard les tablettes graphiques Wacom[1] et une partie des périphériques supportant la norme USB/HID[2].

Notification "à chaud"

L'intérêt des périphériques USB/HID est que leurs pilotes permettent aux programmes d'en découvrir les caractéristiques à l'exécution, cette fonction étant ici déléguée à la boîte à outils HsmTk. Par ailleurs, ils permettent aussi d'être notifié de l'ajout et du retrait de périphériques "à chaud". Cette dernière possibilité, si elle est rendue accessible aux applications, leur permet d'adapter les techniques d'interaction qu'elles proposent à la configuration propre de l'utilisateur.

Pour l'instant, la boîte à outils met ce service de notification "à chaud" de l'ajout ou du retrait de périphériques à disposition des applications uniquement pour les différents dispositifs qui peuvent être utilisés sur les tablettes graphiques Wacom : stylet et *puck*. Comme ces dispositifs comportent un identifiant unique, plusieurs d'entre eux peuvent être utilisés avec une même tablette pour des fonctions différentes. On peut ainsi tester des configurations bimanuelles d'interaction : souris ou *trackball* à la main gauche et *puck* ou stylet à la main droite et basculer de l'interaction monomanuelle à l'interaction bimanuelle dès lors qu'un deuxième périphérique devient disponible.

Pour assurer cette notification, chaque tablette est représentée par un composant composite dont les fils sont les dispositifs qui y sont attachés (Figure 3.4). Les composants abonnés à la tablette sont ainsi notifiés de l'apparition et de la disparition de ses fils, et donc des dispositifs de pointage.

[1] Wacom (`http://www.wacom.com/`) fabrique et commercialise les tablettes graphiques les plus répandues sur le marché.

[2] La norme HID propose un vocabulaire qui est utilisé par les périphériques USB pour déclarer les propriétés des éléments qui les constituent. Cette norme est détaillée sur le site `http://www.usb.org/developers/hidpage/`.

FIG. 3.5 – Représentation sous forme d'arbre d'un système de fichiers

3.2.2 Périphérique de sortie

Le périphérique de sortie proposé par HsmTk consiste en une ou plusieurs fenêtres dans lesquelles du dessin vectoriel est affiché. Le choix d'un modèle vectoriel permet de s'abstraire du modèle physique de l'écran composé de pixels. La réalisation d'interfaces zoomables par exemple devient ainsi triviale techniquement puisque le modèle graphique n'a pas d'échelle intrinsèque. De plus, l'interface peut être directement dessinée par des designers graphiques à l'aide de leurs outils usuels, ce qui facilite la réalisation d'interfaces ayant une haute qualité de finition.

Ce modèle graphique a par ailleurs besoin d'être instrumenté pour permettre l'interaction. Il est en particulier nécessaire de pouvoir remonter de la représentation graphique d'un objet à cet objet lui-même pour que les manipulations interactives aient un effet sur celui-ci. Nous détaillons donc comment ce lien entre représentation graphique et sémantique des objets est obtenu.

Modèle graphique

Le modèle graphique utilisé est celui de SVG (*Scalable Vector Graphics*[3]). Il permet ainsi la mise au point des graphismes de l'interface à l'aide d'outils dédiés à la création graphique qui supportent, comme Adobe Illustrator, ce format. Basé sur XML, il peut aussi être manipulé par programme grâce à une API elle aussi normalisée. Son rendu est assuré par la bibliothèque svgl développée par Stéphane Conversy [Conversy et Fekete, 2002].

Outre l'aspect vectoriel, le modèle de SVG permet une structuration des objets graphiques sous forme de graphe dirigé sans cycle (DAG). Pour cela, SVG propose la notion de groupe qui permet de réunir récursivement des primitives géométriques au sein d'un même objet, ainsi que la notion de symbole qui permet de définir un objet graphique et de l'utiliser plusieurs fois dans le graphe. Ainsi, l'arborescence d'un système de fichiers représentée dans la Figure 3.5 correspond au document SVG de la Figure 3.6. Celui-ci utilise en particulier la notion de groupe pour structurer les graphismes de manière arborescente, reflétant ainsi la structure sous-jacente des données représentées (ligne 21 : le groupe représente un répertoire et son contenu ; ligne 28 : le

[3] SVG est une spécification du W3C (`http://www.w3.org/TR/SVG/`).

```
01 <svg xmlns:hsm='http://insitu.lri.fr/~blanch/projects/hsm/'>
02    <!-- définitions -->
03    <defs>
04       <!-- définition des flèches qui permettent de manipuler l'arbre -->
05       <symbol id='closed'>
06          <path d='M 9 3 l 5 5 l -5 5 z' style='fill:gray' />
07       </symbol>
08       <symbol id='opened'>
09          <path d='M 6 6 l 5 5 l 5 -5 z' style='fill:gray' />
10       </symbol>
11
12       <!-- définition d'un gradient -->
13       <linearGradient id='bg' x1='0' y1='0' x2='40' y2='40'>
14          <stop offset='0' style='stop-color:lightgray' />
15          <stop offset='1' style='stop-color:white' />
16       </linearGradient>
17    </defs>
18
19    <!-- arbre -->
20    <g>
21       <g hsm:behaviour='tree'>
22          <!-- étiquette -->
23          <rect width='1000' height='16' style='fill:url(#bg)'/>
24          <use xlink:href='#opened' />
25          <text x='22' y='12.5'>~/test/</text>
26
27          <!-- contenu -->
28          <g transform='translate(16,16)'>
29             <g hsm:behaviour='node'>
30                <rect width='1000' height='16' style='opacity:0'/>
31                <text x='22' y='12.5'>README</text>
32             </g>
33             <g hsm:behaviour='node'>
34                <rect width='1000' height='16' style='opacity:0'/>
35                <text x='22' y='12.5'>INSTALL</text>
36             </g>
37
38             ...
39          </g>
40       </g>
41    </g>
42
43 </svg>
```

FIG. 3.6 – Document SVG (simplifié) correspondant à la Figure 3.5

groupe représente le contenu du même répertoire). Il utilise aussi la notion de symbole pour décrire une fois pour toutes les flèches qui servent à déployer ou à refermer des parties de l'arbre (lignes 5 à 10).

D'autres capacités de SVG sont utilisées sur cet exemple : la possibilité d'utiliser simplement des dégradés, les transformations géométriques, et la transparence. Le fond dégradé permet de distinguer visuellement les fichiers faisant partie d'un même répertoire, renforçant ainsi les indices visuels donnés par l'alignement horizontal. Le dégradé est, comme les symboles, défini une fois pour toutes (lignes 13 à 16) puis utilisé comme remplissage pour les rectangles de fond des répertoires (ligne 23). Une transformation géométrique simple, la translation, permet de positionner le contenu des répertoires par rapport à leurs étiquettes (ligne 28). Enfin, la transparence est utilisée pour ajouter un fond rectangulaire invisible à chaque fichier (lignes 30 et 34). Ce fond est nécessaire pour définir la zone permettant d'interagir avec le fichier, mais pour ne pas occulter le dégradé qui caractérise le répertoire, il est rendu transparent.

Support pour l'interaction graphique

Les mécanismes de la mise en œuvre de l'interaction sont détaillés à la Section 3.2.3 ci-dessous. Cependant, pour pouvoir *in fine* interagir avec les objets graphiques, un lien doit exister entre ceux-ci et les objets du noyau fonctionnel qu'ils représentent. Pour ce faire, HsmTk permet d'attribuer à des groupes SVG une classe à qui est délégué leur comportement interactif. Cette classe est spécifiée par un attribut particulier annotant le groupe au sein du document SVG (par exemple ligne 21 : `hsm:behaviour='tree'`).

Pour permettre de faire le lien inverse et de retrouver un objet à partir de sa représentation, HsmTk propose un mécanisme de *picking*. Celui-ci permet, étant donné un point dans une fenêtre, de rechercher à cet endroit un objet dont le comportement accepte une manipulation donnée. Pour cela, les éléments SVG qui englobent l'élément situé au point choisi sont examinés, en partant du plus profond d'entre eux dans l'arbre SVG, et en remontant vers la racine. Cette remontée s'interrompt lorsqu'un élément dont le comportement convient à ce qui est recherché est trouvé. Cette adéquation est testée grâce aux mécanismes de typage du langage C++.

Ainsi, un clic ayant lieu sur le fichier `README` de la Figure 3.5 déclenche la recherche d'un élément acceptant ce clic. S'il a lieu précisément sur l'une des lettres du texte, la recherche commence alors par l'élément SVG `text`, ligne 31 sur la Figure 3.6. S'il passe au travers des lettres, l'élément situé sous le curseur est le rectangle spécifié ligne 30 qui appartient au même groupe. La recherche débute alors par ce rectangle. Comme ni le rectangle, ni le texte, n'ont de comportement associé, la recherche remonte dans les deux cas à leur parent commun, le groupe de la ligne 29. Ce groupe spécifie un comportement grâce à l'attribut `hsm:behaviour='node'`. S'il accepte le clic, la recherche s'arrête là. Dans le cas contraire, elle se poursuit en remontant au groupe père (ligne 28). Celui-ci n'ayant pas de comportement associé, elle remonte alors au groupe qui l'englobe (ligne 21), et ainsi de suite jusqu'à l'élément racine du document SVG.

Le *picking* peut être paramétré pour examiner non seulement l'élément situé à la position désirée, mais aussi les éléments situés sous lui, qu'il occulte éventuellement. Dans ce cas, la recherche commence par l'élément le plus proche, et remonte comme précédemment à la racine, puis examine l'élément situé juste en-dessous et ses ancêtres, et ainsi de suite. Pour cette recherche avancée, on peut spécifier qu'on ne s'intéresse qu'aux éléments situés sous un élément donné, et exclure ainsi certaines couches superficielles du document SVG. Ce mécanisme permet en pratique de réaliser des outils semi-transparents en allant chercher sous ceux-ci les éléments finalement concernés par l'interaction en "passant à travers" ces outils.

3.2.3 Interaction

L'interaction repose nécessairement sur des conventions : il faut que ses parties prenantes, l'acteur et l'objet de l'interaction, soient d'accord sur les termes qu'ils vont employer et sur le sens qu'ils lui donnent. Nous appelons *protocole* le langage que partagent l'acteur et l'objet de l'interaction. Nous appelons *comportement* l'objet à qui est confié la réalisation des réactions de l'objet de l'interaction à l'invocation des protocoles qu'il comprend. Nous appelons en-

fin *technique d'interaction* le code qui permet d'invoquer un ou plusieurs protocoles en fonction des actions de l'utilisateur. Nous explicitons ici la forme que prennent ces trois aspects de l'interaction dans HsmTk et les outils mis à disposition du programmeur pour les mettre en œuvre.

Protocole d'interaction

Un protocole est défini par la signature d'un ensemble de fonctions qu'un objet doit réaliser (une interface dans la terminologie Java). Le protocole le plus simple possible proposé par HsmTk n'a pas de contenu. Il comporte simplement deux fonctions, l'une signifiant le début (`begin`) de l'interaction, l'autre sa fin (`end`). La fonction de début permet à la cible de l'interaction de signifier si elle accepte ou non la manipulation, et si elle l'accepte, le comportement à qui sera déléguée sa gestion. Ce protocole minimal doit être supporté par tous les autres.

Le protocole *begin/end* permet de mettre en place la gestion de la concurrence de l'interaction, soit en refusant que plusieurs acteurs agissent de concert sur le même objet, soit en acceptant les ordres de plusieurs acteurs de manière indifférenciée (un objet déplacé par deux "mains" subit le déplacement cumulé des deux mains), soit encore en attribuant un rôle différent aux acteurs qui entrent successivement en jeu (on peut alors passer d'un simple déplacement à un déplacement accompagné d'un redimensionnement).

HsmTk propose quelques protocoles prédifinis, dont certains ont trait à l'interaction graphique :

- *enter/leave* peut-être activé quand un point entre ou sort d'un objet graphique ;
- *translate* permet de déplacer un objet ;
- *rotate* permet de pivoter un objet ;
- *zoom* permet de changer l'échelle d'un objet ; et
- *color* permet de changer la couleur d'un objet.

Pour ces protocoles graphiques, une réalisation est fournie sous la forme de comportements par défaut. Ceci est possible indépendamment des applications car la sémantique des objets graphique est connue de HsmTk, c'est celle de SVG et de son API.

D'autres protocoles sont proposés qui n'ont pas une simple sémantique graphique mais qui constituent des manipulations élémentaires :

- *press/release* consiste à enfoncer et relâcher un bouton par exemple ;
- *open/close* consiste à ouvrir et fermer un objet (menu, vue arborescente, *etc.*) ;
- *accept* consiste à vérifier la compatibilité d'un objet avec une cible (il est utile en particulier pour le protocole suivant) ;
- *drag/drop* est une spécialisation de *translate* pour laquelle la compatibilité du lieu de lâché est testée, et qui met les deux objets en relation s'ils sont compatibles.

D'autres protocoles peuvent être ajoutés aisément pour les besoins d'une application particulière, il suffit pour cela de créer une nouvelle classe dérivant de la classe de base `hsm::Protocol` ou d'une de ses spécialisations (la Figure 3.7 donne par exemple la déclaration du protocole *open/close*, alors que la Figure 3.8 montre sa relation avec sa classe de base). Elle héritera de

```
01 namespace hsm {
02    struct OpenClose :
03       Protocol {
04
05       virtual void open() = 0;
06       virtual void close() = 0;
07    };
08 }
```

FIG. 3.7 – Déclaration du protocole *open/close*

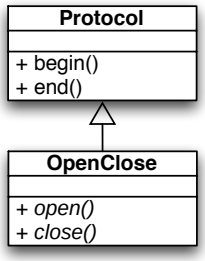


FIG. 3.8 – Le protocole *open/close*

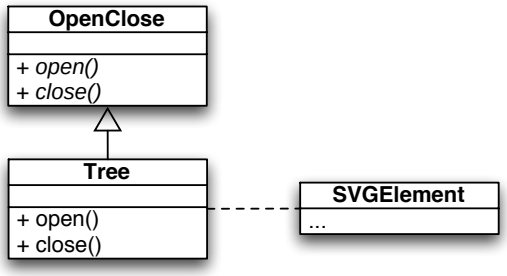


FIG. 3.9 – Le comportement `Tree` réalisant le protocole *open/close*

celle-ci le protocole *begin/end* ainsi que la mécanique qui permet au comportement de connaître le point précis où l'action a commencé, ce qui est nécessaire pour la plupart des interactions graphiques.

Comportement

Lien avec les protocoles. Le comportement d'un objet est la réalisation d'un ou plusieurs protocoles. Un comportement spécifie comment il réagit aux primitives de l'interaction, et en premier lieu aux fonctions `begin` et `end`. Le comportement répond par défaut au début de l'interaction en renvoyant un pointeur sur lui-même, ce qui indique qu'il est prêt à assurer l'interaction. Il peut renvoyer un pointeur *null* pour indiquer qu'il la refuse. Comme précisé plus haut, il peut aussi renvoyer un pointeur sur un autre objet, ce qui permet de lui déléguer la gestion de l'interaction ou d'attribuer des rôles différents à deux acteurs qui tentent simultanément la même interaction.

```
01 class Tree :
02    public hsm::OpenClose {
03
04 protected :
05    svg::SVGUseElement *handle;
06    svg::SVGGElement   *content;
07
08 public :
09    Tree(svg::SVGElement *elem);
10
11    virtual void open();
12    virtual void close();
13 };
```

FIG. 3.10 – Déclaration du comportement d'un arbre

La Figure 3.9 montre le comportement `Tree` qui réalise le protocole `OpenClose`. Dans l'exemple des Figures 3.5 et 3.6, chaque répertoire est décoré d'un attribut qui spécifie son comportement : `hsm:behaviour='tree'` (Figure 3.6, ligne 21 par exemple). HsmTk attache automatiquement une instance de la classe `Tree` aux nœuds ainsi décorés lors du chargement de l'arbre SVG, et permet ainsi de remonter de la représentation au comportement. Cette information est utilisée par le *picking* décrit ci-dessus (Section 3.2.2). On peut rechercher par exemple un objet qui implémente le protocole *open/close* ainsi :

```
hsm::OpenClose *object = window->pick< hsm::OpenClose >(position);
```

La fonction de *picking* s'applique à une fenêtre et prend deux paramètres :
- la *classe* du protocole attendu (ici `hsm::OpenClose`) ; et
- la *position* à laquelle le *picking* doit être effectué.

Si la position correspond à un objet dont le comportement est de la classe `Tree`, cet appel ramènera un pointeur sur ce comportement puisqu'il réalise le protocole attendu *open/close*.

La déclaration[4] de la classe `Tree` correspondante est donnée dans la Figure 3.10. La ligne 2 spécifie que ce comportement réalise le protocole *open/close*. La ligne 9 définit le constructeur utilisé par HsmTk pour construire des instances de ce comportement. Ce constructeur reçoit l'élément SVG auquel le comportement est associé, ce qui maintient le lien inverse à celui du *picking* et permet au comportement de modifier la représentation à laquelle il est attaché au gré de ses propres changements d'état. Les lignes 11 et 12 spécifient que cette classe propose sa propre implémentation des méthodes `open` et `close` réalisant ainsi concrètement le protocole `hsm::OpenClose`.

Lien avec la représentation graphique. La Figure 3.11 montre une réalisation simplifiée du comportement associé à chacun des nœuds de l'arbre. Pour manipuler les objets graphiques auxquels il est attaché, le comportement dispose de l'API de SVG mise à disposition par svgl. C'est ainsi que les lignes 14–15 et 19–20 spécifient les manipulations pour faire apparaître ou masquer les fils d'un nœud.

HsmTk propose deux fonctionnalités de plus haut niveau que la manipulation directe du SVG :

[4] Quelques détails sont omis pour faciliter la lecture de l'exemple.

```
01 Tree::Tree(svg::SVGElement *elem) {
02    // contrat structurel
03    hsm::svgl::Childs(elem).get(handle)
04                           .get(content);
05
06    bool isOpen = false;
07    hsm::svgl::getAttribute(elem, "hsm-arg:isOpen", isOpen);
08    if(not isOpen) {
09       close();
10    }
11 }
12
13 void Tree::open() {
14    handle->setHref("#opened");
15    elem->appendChild(content);
16 }
17
18 void Tree::close() {
19    handle->setHref("#closed");
20    elem->removeChild(content);
21 }
```

FIG. 3.11 – Réalisation du comportement d'un arbre

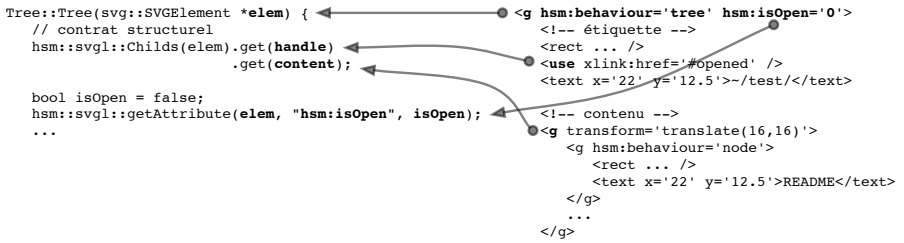


FIG. 3.12 – Liens établis entre le comportement et la représentation SVG

– le moyen pour le comportement d'exprimer un *contrat structurel* que doit remplir un fragment SVG pour pouvoir lui être lié ; et
– le moyen pour les fragments SVG de *passer des paramètres* à la construction du comportement qui leur est associé pour personnaliser ce dernier.

Les lignes 3 et 4 illustrent la première fonctionnalité : pour pouvoir être un `Tree`, le fragment SVG doit posséder au moins deux fils, le premier étant la flèche qui indique visuellement s'il est ouvert, et le second doit être un groupe qui est le contenu de l'arbre. Ce contrat vérifie le type des objets, et interrompt la construction du comportement s'il n'est pas rempli. Il récupère simultanément des pointeurs sur les éléments requis ce qui permet de les manipuler par la suite.

La seconde fonctionnalité de la boîte à outils permet d'ajouter des attributs aux éléments de l'arbre, ceux-ci pouvant être récupérés par les comportements. Par exemple, les lignes 6 à 10 récupèrent l'attribut nommé `hsm-arg:isOpen` et initialisent l'état de l'arbre en fonction de sa valeur. La Figure 3.12 résume les liens ainsi établis entre les variables du comportement à gauche et éléments du fragment SVG correspondant à droite.

```
01 class Tree :
02    public hsm::Behaviour,
03    public hsm::OpenClose {
04
05 private:
06    static const hsm::Behaviour::Factory::Register< Tree > registration;
07    ...
08 };
09
10 const hsm::Behaviour::Factory::Register< Tree > Tree::registration("tree");
```

FIG. 3.13 – Enregistrement d'un comportement

FIG. 3.14 – Différentes représentations d'un même arbre

Répertoire de comportements. Pour que les comportements puissent être instanciés par HsmTk, typiquement lors du chargement d'un document SVG, il faut qu'ils soient répertoriés par la boîte à outils. Celle-ci fournit donc un mécanisme permettant aux classes de comportement de s'enregistrer. Elles doivent pour cela simplement instancier un objet fourni par HsmTk qui réalise toutes les opérations nécessaires, comme cela est illustré par la Figure 3.13. La ligne 6 déclare cet objet, et la ligne 10 l'instancie, en spécifiant la chaîne de caractères qui identifiera ce comportement (ici `"tree"`). C'est cette chaîne qui, lorsqu'elle est utilisée comme valeur pour l'attribut `hsm:behaviour` ajouté au SVG, permet à HsmTk d'attacher le comportement correspondant au fragment de SVG.

Si un comportement qui n'est pas encore enregistré est spécifié par un document SVG, HsmTk recherche alors dans un entrepôt centralisé un module exécutable portant son nom. S'il existe, ce module est chargé, ce qui provoque automatiquement l'enregistrement de la classe de comportement qu'il définit. Ce comportement est alors instancié comme s'il avait été défini par le programme lui-même.

Grâce à ce mécanisme, les comportements standards de HsmTk peuvent être enrichis en fonction des besoins des programmeurs. Ces nouveaux comportements sont alors disponibles pour d'autres applications. Comme ils reposent uniquement sur un contrat structurel avec le fragment de SVG auquel ils sont attachés, un même comportement peut être utilisé pour plusieurs objets indépendamment de leurs formes particulières. La Figure 3.14 montre ainsi des arbres ayant des structures graphiques différentes — présence ou non d'icones, décorations variées — mais qui utilisent exactement le même comportement.

Technique d'interaction

Une technique d'interaction est le mécanisme qui permet de transformer les actions de l'utilisateur en un ensemble d'appels aux fonctions des protocoles d'interaction. Les machines à états hiérarchiques présentées au Chapitre 4 sont utilisées dans HsmTk pour cette traduction. Des exemples de techniques d'interaction seront donnés à la Section 4.3 pour illustrer ce chapitre.

3.3 Services de bas niveau

La programmation d'applications interactives ne requiert pas uniquement la mise en œuvre de techniques algorithmiques. Elle nécessite aussi de gérer finement des tâches de bas niveau qui sont très liées au modèle d'exécution de la machine. Outre les abstractions des périphériques, d'autres abstractions sont nécessaires pour que l'utilisateur puisse interrompre une tâche en cours, ou pour que ses entrées soient prises en compte immédiatement par le système, quelle que soit son activité. Pour réaliser cela, les programmeurs ont par exemple besoin de pouvoir mettre en œuvre de la concurrence dans leurs programmes, et des mécanismes d'interruption et de notification. Ces mécanismes sont en général fournis par le système d'exploitation lui-même ou par une couche logicielle située juste au-dessus, dans des bibliothèques proches du système. Ils proposent des modèles variables d'un système à l'autre, parfois difficiles à appréhender, et donc difficiles à utiliser de manière correcte. Notre boîte à outils offre quelques abstractions pour faciliter ces mises en œuvre. D'autres services indispensables à certaines tâches sont aussi assurés par des bibliothèques dont la conception et les principes sous-jacents sont directement liés aux particularités de la plate-forme et peuvent ainsi différer assez largement d'un système à l'autre. En particulier la possibilité de charger des modules exécutables dynamiquement au cours de l'exécution d'un programme, ou l'accès au système de fenêtrage et au contenu des fenêtres, sont des aspects typiquement dépendants des systèmes d'exploitation.

Ces divers aspects sont généralement peu pris en compte par les diverses boîtes à outils. Elles laissent souvent les programmeurs se "débrouiller" avec les bibliothèques proposées par le système. HsmTk propose des abstractions communes, qui conservent le niveau de détail le plus bas, mais qui suppriment la variabilité des modèles en fonction des systèmes. Les choix faits pour unifier les modèles ont pour but de proposer au programmeur l'ensemble le plus cohérent possible et le plus simple à utiliser.

3.3.1 Concurrence

L'introduction de la concurrence dans un programme entraîne un coût qui doit être justifié. Elle introduit en particulier un surcoût lors de la mise au point du programme qui peut se révéler délicate. Nous pensons néanmoins que l'utilisation d'un modèle dans lequel la concurrence a sa place libère le programmeur de contraintes qui sont elles-aussi pesantes, l'obligeant souvent à simuler lui-même d'une certaine manière la concurrence. Elle lui permet de programmer de manière assez naturelle des comportements qui doivent évoluer simultanément et indépendamment comme on peut en rencontrer

souvent dans des applications interactives. Par exemple, un objet ou sa couleur doit pouvoir être animé pour attirer l'attention de l'utilisateur cependant que celui-ci doit pouvoir continuer à interagir normalement. Il sera alors naturel de confier l'animation à un fil d'exécution distinct de celui du programme qui gère l'interaction. Toutefois, pour permettre ce type de découpage de l'exécution, il faut que la boîte à outils fournisse les mécanismes de communication et de synchronisation adéquats pour que le programme puisse interrompre l'animation au moment opportun, par exemple lorsque l'objet est détruit.

Boucle d'exécution

Nous avons choisi d'intégrer la concurrence dans notre boîte à outils en proposant un modèle simple au programmeur, qui permet de réduire le surcoût lors du développement : les *boucles d'exécution*. Elles consistent en une structure offrant une abstraction à un fil d'exécution partageant la mémoire du programme principal (ou processus léger ou encore *thread*) qui est répandu sur la plupart des systèmes d'exploitation. La boucle d'exécution exécute de manière répétitive un code donné. Contrairement aux abstractions couramment fournies par les bibliothèques proches du système, elle peut être arrêtée puis relancée (cette pause s'effectuant après la fin d'un tour complet de la boucle) depuis un autre fil d'exécution, ce qui permet de facilement mettre en œuvre des collaborations du type de celle de l'animation décrite ci-dessus. Par ailleurs le programmeur peut aussi fixer un délai qui doit s'écouler entre chaque tour de boucle, ce qui permet d'exécuter du code à intervalles donnés.

Le problème de la cohérence de la mémoire propre aux accès concurrents reste cependant posé par ces boucles d'exécution. HsmTk propose un objet de synchronisation simple et bien connu, le *verrou*, qui, associé aux structures de données communes à plusieurs fils, permet d'en sérialiser les accès concurrents par un verrouillage explicite laissé à la charge du programmeur.

Minuterie

La boucle d'exécution est bien adaptée à des comportement particuliers que l'on retrouve dans des programmes interactifs : ceux qui sont répétitifs comme la mise à jour d'un objet animé ou le traitement d'événements présents dans une queue. Cependant, on peut avoir besoin d'exécuter du code après un intervalle de temps donné une fois seulement — l'apparition d'une bulle d'information après que le curseur soit resté un temps donné au dessus d'un élément de l'interface en constitue un exemple. Dans ce cas, l'attente active durant ce délai paralyserait le programme.

HsmTk propose, pour répondre à ce besoin, une *minuterie* qui permet de programmer l'exécution d'un code donné après un délai fixé. Une fois cette tâche programmée, le programme peut continuer son exécution normale. Il peut aussi, si le délai n'a pas expiré, déprogrammer cette tâche. Cette possibilité permet par exemple dans le cas de la bulle d'information d'annuler son apparition lorsque le curseur a bougé avant que le laps de temps d'immobilité requis ne soit complètement écoulé.

Communication entre programmes

Enfin, un programme interactif peut avoir besoin de communiquer avec d'autres programmes et avec le système. Ce type de communication s'effectue en général au travers de canaux tels que des fichiers ou des objets similaires dans lesquels les programmes peuvent lire et écrire des données. Ce mode de communication est asynchrone, et nécessite de pouvoir mettre un fil d'exécution en attente s'il doit lire ou écrire dans un canal et que celui-ci n'est pas prêt pour ces opérations. Pour que cette attente ne soit pas active, la plupart des systèmes fournissent un service qui permet de réveiller des fils d'exécution mis en attente. HsmTk propose une abstraction de ce mécanisme qui permet de traiter uniformément les différents objets de communication usuels. Cette abstraction, le *sélecteur*, permet de suspendre un fil d'exécution dans l'attente du passage dans l'état requis de l'objet de communication. Elle permet aussi de ne pas mettre en attente le fil courant mais de spécifier une fonction qui devra être exécutée lorsque ce changement d'état aura lieu, l'attente se faisant alors dans un fil d'exécution dédié.

Il propose par ailleurs un objet simplifié de communication, l'*alerte*, qui transmet de simples signaux entre fils d'exécution au travers de ce mécanisme de sélecteur.

3.3.2 Chargement dynamique de modules

Pour supporter l'aspect contextuel de l'interaction, il est utile de fournir au programmeur un support à la modularisation de son application. Une application interactive peut au démarrage détecter la présence de périphériques particuliers et reconfigurer les techniques d'interaction qu'elle utilise en fonction de cette présence — par exemple, la présence d'un second périphérique de pointage comme un *trackball* en plus de la souris standard permettra l'utilisation de techniques bimanuelles. Dans ce cas, il est utile de ne charger le code offrant le support de ce périphérique et de cette technique d'interaction que si cette dernière peut être mise en œuvre.

Le support au chargement dynamique de modules — au cours de l'exécution du programme — est très hétérogène parmi les systèmes les plus courants. HsmTk offre donc un système unifié de chargement dynamique de modules. C'est grâce à ce support qu'est modularisée la boîte à outils HsmTk elle-même. La gestion des périphériques d'entrée est par exemple gérée dans des modules qui peuvent être chargés en fonction des besoins, ou de leur disponibilité, lors de l'exécution du programme.

3.3.3 Fenêtres

La dernière abstraction de bas niveau fournie au programmeur est un objet qui abstrait une fenêtre. En effet, bien que la notion de fenêtre soit commune à pratiquement tous les systèmes offrant des applications interactives, le modèle qui sous-tend celle-ci présente beaucoup de variabilité d'un système à l'autre. Dans certains cas, c'est le système qui gère le positionnement et le redimensionnement de la fenêtre en réponse à des actions de l'utilisateur sur les décorations de celle-ci. Dans ce cas, le système notifie ensuite l'application à qui appartient la fenêtre des modifications qui ont eu lieu pour qu'elle prenne

en charge la mise à jour de son contenu. Sur d'autres systèmes, c'est l'application elle-même qui a la responsabilité de prendre en compte les actions de l'utilisateur sur les décorations de ses propres fenêtres pour en modifier la géométrie et mettre à jour leur contenu. Sur ces systèmes, la bibliothèque HsmTk prend en charge cet aspect de la gestion de fenêtres pour fournir un modèle unifié au programmeur, de sorte que l'application n'a alors qu'à gérer le contenu de ses fenêtres.

Modèle graphique de bas niveau

Le modèle graphique de bas niveau proposé pour les fenêtres est celui d'OpenGL[5], la bibliothèque qui permet d'utiliser l'accélération matérielle du rendu graphique par le processeur des cartes dédiées à l'affichage. Le modèle qu'elle propose est celui de primitives géométriques vectorielles (points, segments, polygones) plongées dans un espace à trois dimensions qui est projeté puis échantillonné pour être représenté à l'écran. Il supporte la transparence, les transformations géométriques et fournit par ailleurs un support à diverses opérations réalisables sur le résultat du rendu lui-même. Le niveau d'abstraction qu'il donne est donc plus élevé que celui de la grille de pixels de l'écran. Il ne propose cependant qu'un support très limité pour construire des objets de plus haut niveau que celui des primitives élémentaires, comme pour l'affichage de texte par exemple. C'est pour cela que la bibliothèque svgl [Conversy et Fekete, 2002] est utilisée, puisque cette bibliothèque permet d'afficher du SVG en utilisant les fonctions de base d'OpenGL.

La bibliothèque OpenGL a par ailleurs été choisie car elle offre un ensemble standard de fonctions pour pratiquement tous les systèmes d'exploitation. La partie non portable liée à l'utilisation d'OpenGL — la création d'un contexte graphique permettant l'affichage des résultats des commandes dans une fenêtre particulière — est prise en charge par HsmTk.

Événements du système

Les événements de bas niveau signalant l'activité des périphériques d'entrée sont généralement fournis par le système d'exploitation et le système de fenêtrage. HsmTk permet aux programmeurs d'insérer leurs propres gestionnaires au sein de ceux qui sont proposés pour intercepter ces événements. Ces gestionnaires permettent de créer des abstractions de plus haut niveau que les événements systèmes pour des périphériques qui ne sont pas encore supportés par la boîte à outils. Ils peuvent être réalisés sous la forme de modules qui sont chargés à l'exécution, ce qui facilite leur réutilisation. C'est d'ailleurs en utilisant ce mécanisme que le support des périphériques standards de la boîte à outils est fourni : des modules sont proposés pour le clavier, pour la souris, pour les tablettes graphiques Wacom, et pour les périphériques USB/HID.

3.4 Conclusion

Nous avons présenté dans ce chapitre les divers niveaux d'abstraction qu'offre la boîte à outils HsmTk. Ces abstractions se déploient, pour les entrées comme

[5] La bibliothèque OpenGL est décrite et documentée sur le site : `http://www.opengl.org/`.

pour les sorties, depuis un niveau très bas, comme les événements du système, jusqu'à un niveau plus proche des objets familiers aux utilisateurs, la souris, le texte, les formes géométriques. À chacun de ces niveaux la "boîte est ouverte". Par ouverte, nous entendons que les abstractions proposées, si elles permettent de limiter la complexité de certains aspects de la création d'une application interactive en hiérarchisant les problèmes et en structurant les détails à prendre en compte, ne le font pas en occultant complètement les aspects de bas niveau. Ceux-ci sont toujours accessibles, car nous savons, par notre propre expérience et surtout par celle de nos prédécesseurs, que c'est le réglage des détails les plus subtils qui conditionne la réussite d'une technique d'interaction. Cette ouverture est d'ailleurs le seul gage que nous pouvons apporter lorsque nous affirmons que la boîte à outils est faite pour supporter des développements non-anticipés.

La hiérarchie introduite dans les abstractions a, par ailleurs, une autre vertu. Elle permet de modulariser le code, et cette modularité est favorable à sa réutilisation. Faciliter la réutilisation est un objectif affiché de la boîte à outils. Elle propose pour cela un ensemble de mécanismes, et notamment le chargement dynamique de modules, ce qui permet de réutiliser des pans entiers d'applications tels quels. Si cette approche rappelle l'approche WIMP tant décriée, il faut préciser qu'ici les interacteurs n'ont aucune limitation géométrique, et surtout que de créer de nouveaux interacteurs est aussi facile que d'utiliser ceux qui sont proposés ! Ce support à la modularité permet aussi d'introduire et de réutiliser le support à de nouveaux périphériques. Il permet enfin et surtout de proposer et de réutiliser de nouvelles techniques d'interaction.

Nous avons vu qu'il était facile, grâce à la notion de contrat structurel, d'utiliser un comportement interactif avec des objets graphiques différents, et donc de réduire le couplage entre les interactions et les objets graphiques. De même, une technique d'interaction agissant grâce à l'invocation d'un protocole particulier, elle peut être utilisée sur tous les objets comprenant ce protocole. De plus, de nouvelles techniques pourront être utilisée sur ces mêmes objets pour peu qu'elles respectent ce protocole. Pour faciliter l'innovation et la réutilisation, il nous reste à proposer un support satisfaisant à la réalisation de nouvelles techniques d'interaction. C'est à cet effet que le chapitre suivant présente une abstraction supplémentaire : un langage dédié à l'interaction. Ce langage permet de faire le lien entre périphériques et protocoles d'interaction, au sein de techniques d'interaction, mais aussi de réaliser les comportements interactifs.

Chapitre 4

Les machines à états hiérarchiques

Nous présentons dans ce chapitre le formalisme utilisé pour étendre le langage de programmation et faire des interactions des objets à part entière du vocabulaire du programmeur. Après la présentation informelle du formalisme des machines à états hiérarchiques grâce à un exemple simple, leur syntaxe et leur sémantique sont présentées, puis quelques exemples concrets de leur utilisation pour définir des techniques d'interaction sont donnés.

Un objectif de la boîte à outils HsmTk est de faire en sorte que les interactions deviennent des objets à part entière du vocabulaire du programmeur. Or, les langages de programmation impératifs habituellement utilisés pour construire des interfaces ont été conçus pour des applications purement algorithmiques — appliquer un traitement particulier à chaque élément d'un ensemble de données par exemple. Les structures de contrôle qu'ils proposent sont donc essentiellement adaptées à ces traitements séquentiels et répétitifs de données. Elles sont très proches du modèle d'exécution des ordinateurs qui, comme le note Wegner [1997], « ne peuvent accepter d'entrées pendant qu'ils calculent ; et se coupent alors totalement du monde extérieur ».

Dans ce contexte, le développement de logiciels interactifs ne peut se faire qu'en détournant le modèle algorithmique de programmation de son but initial. Cet écart entre le modèle proposé au programmeur et la tâche qu'il a

à résoudre en l'utilisant (programmer une application interactive), a ainsi introduit un style de programmation reposant sur l'usage de fonctions de rappel (ou *callbacks*). Les problèmes de ce type de programmation sont bien connus : le code produit est difficile à comprendre, à maintenir, et à réutiliser tant la logique du système est distribuée au sein de multiples fonctions interdépendantes — Myers [1991] compare ces fonctions de rappels à « un plat de spaghettis ».

Nous avons donc pris le parti d'utiliser un formalisme adapté à la description et à la spécification de l'interaction pour augmenter le langage de programmation et faire ainsi des interactions et des comportements interactifs des objets à part entière du langage.

4.1 Introduction

Nous présentons ici un ensemble de formalismes qui ont été proposés et utilisés pour décrire l'interaction. Nous présentons ensuite informellement celui que nous avons adopté pour la boîte à outils HsmTk : les machines à états hiérarchiques (HSM pour *Hierachical State Machines*).

4.1.1 Des langages pour l'interaction

Beaucoup de langages particuliers ou de formalismes ont été proposés pour décrire et spécifier l'interaction. Cette prolifération confirme l'inadaptation d'un langage impératif pour décrire un comportement. Les langages proposés se séparent en plusieurs grandes familles : ceux basés sur des modèles réactifs ou ceux basés sur les modèles à états et transitions par exemple. D'autres langages, plus inspirés des formalismes adaptés à la description de la concurrence, ont aussi été proposés.

Modèles réactifs

Dans les modèles réactifs, la métaphore utilisée est celle d'un réseau de câbles au sein desquels se propagent les flux de valeurs de signaux. La propagation des changements de valeurs peut être synchronisée sur les tops d'une horloge. Les "câbles" relient entre-eux un ensemble de "composants", qui reçoivent les signaux en entrée, leur appliquent un traitement, et en émettent de nouveaux en sortie.

Ce formalisme est bien adapté à la description de la cascade de traitements et de filtrages que subissent les valeurs qui caractérisent les périphériques d'entrée avant d'agir sur les éléments de l'interaction. La boîte à outils ICON de Dragicevic et Fekete [2001], présentée brièvement à la Section 2.3.2, page 40, l'utilise ainsi avec succès pour permettre aux programmeurs et aux utilisateurs de configurer l'interaction suivant les périphériques d'entrée dont ils disposent. ICON intègre un éditeur graphique qui permet de programmer cette interaction visuellement, celui-ci étant en effet nécessaire pour pouvoir manipuler la métaphore du réseau facilement.

La nécessité d'un environnement dédié n'est cependant pas l'inconvénient majeur de ce type de modèles. Leur principale limitation est une de leur caractéristique intrinsèque : ces modèles ne peuvent pas, par essence, modifier leur topologie pour refléter l'état du système. Ainsi, dans ICON, si une sortie

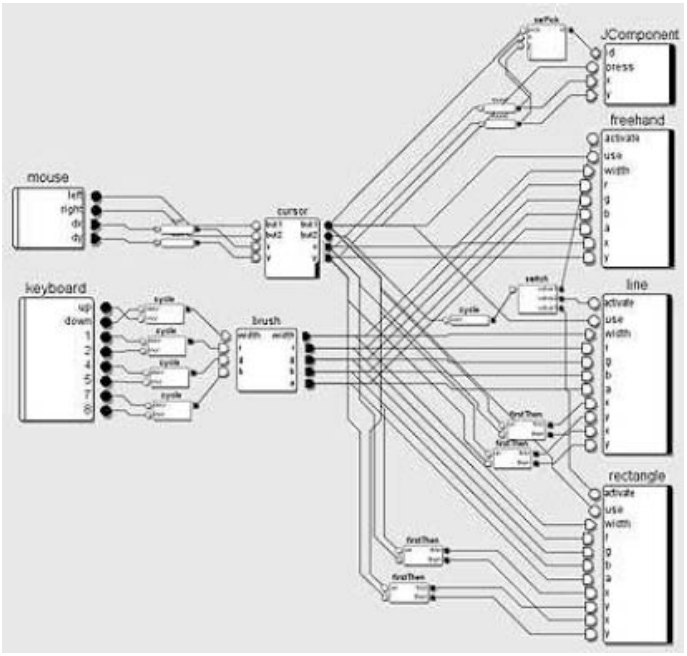


FIG. 4.1 – Multiplication des connexions dans un formalisme réactif
(Illustration extraite de [Dragicevic, 2004])

(la position de la souris) peut être connectée alternativement à différents composants (positions de différents curseurs liés à différents outils), cette sortie sera connectée en permanence à tous les dispositifs possibles, et c'est un paramètre supplémentaire, un bit d'activation, qui permettra de spécifier quel dispositif doit effectivement répondre aux modifications courantes — à la charge alors des autres dispositifs d'ignorer les signaux qui leur parviennent tout de même. La logique de cette activation est alors diffuse au sein du reste du réseau. Ce problème est admis par [Dragicevic, 2004] qui propose quelques aménagements au formalisme visuel pour réduire la complexité visuelle que ce multiplexage crée dans ICON. La Figure 4.1 n'utilise pas ces aménagements, et montre ce que donne dans ICON la configuration de quatre outils (à droite) associés à un seul curseur, et dont le multiplexage est assuré grâce à des touches du clavier. Dragicevic [2004] admet par ailleurs la nécessité d'intégrer une composante basée sur des états à son modèle pour pouvoir plus facilement gérer l'état de l'interaction.

D'autres modèles ont été proposés qui associent à l'approche réactive un niveau de contrôle établissant les différentes connexions durant des périodes données grâce à un modèle de type machine à états. C'est ce que proposent en particulier Jacob *et al.* [1999]. Ils présentent un environnement visuel de programmation, VRED, dans lequel les connexions sont soumises à des conditions dont les valeurs de vérité sont régies par les évolutions d'une machine à états. La Figure 4.2 montre un exemple de cet environnement, avec en haut les connexions du formalisme réactif et en bas la machine à états qui en active les transitions. Sur cet exemple, qui exprime simplement le déplacement d'un

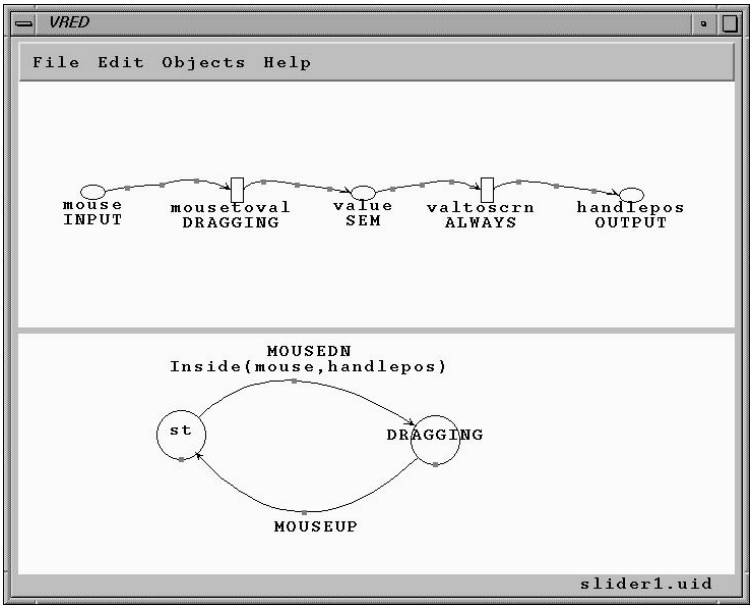


FIG. 4.2 – L'environnement visuel VRED

(Illustration extraite de [Jacob *et al.*, 1999])

objet à l'aide du curseur, le lien entre les deux représentations peut être retrouvé facilement (la condition `DRAGGING` est activée dans l'état de droite de la machine, ce qui autorise la connexion `mousetoval` entre `mouse` et `value` en haut à gauche). Néanmoins, dès que le nombre de conditions augmente, le réseau de connexions et ses liens avec les états de la machine, deviennent, là encore, difficile à identifier et à interpréter visuellement.

Comme les modèles réactifs nécessitent une représentation visuelle pour être manipulés facilement, et qu'ils ne peuvent modéliser facilement à eux seuls tous les aspects de l'interaction, nous nous sommes tourné vers les formalismes à base de machines à états.

Modèles à base d'états et de transitions

Les modèles à base d'états et de transitions proviennent tous de variantes ou de généralisations des automates déterministes finis (DFA). L'automate passe d'état en état en franchissant des transitions activées par la réception d'un événement particulier. Les machines à états permettent d'associer des actions aux transitions (modèle de Mealy [1955]) et/ou à l'entrée et la sortie des états (modèle de Moore [1956]). Elles ont le même pouvoir d'expression que les automates (celui des langages réguliers), mais en pratique les actions peuvent avoir des effets de bord, et suivant la portée de ces derniers, le pouvoir d'expression peut être complètement différent. Cependant, dès lors qu'un système interactif est en jeu, il faut garder à l'esprit que ce sont toujours des effets de bord qui permettent *in fine* l'interaction avec l'utilisateur. On peut par exemple reproduire la propagation des variations de la valeur d'une variable — qui est la base de modèles réactifs — en associant à un état une transition

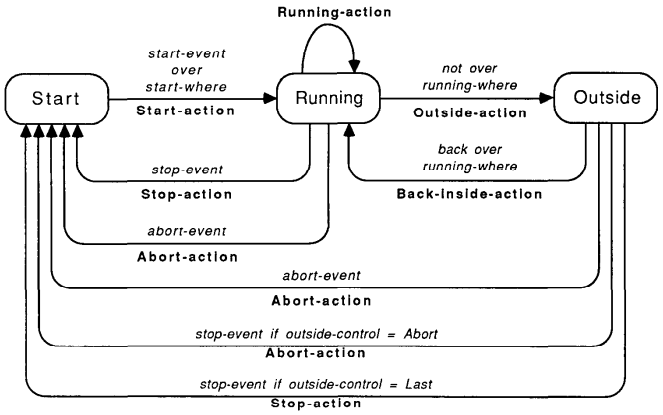


FIG. 4.3 – La machine à états qui spécifie le comportement des interacteurs de Myers

(Illustration extraite de [Myers, 1990])

l'ayant à la fois pour origine et pour cible. Si cette transition est déclenchée par les événements notifiant le changement de l'entrée, et que l'action associée à la transition répercute ces modifications sur une variable cible, la propagation a bien lieu dès que la machine est dans cet état.

Les machines à états sont couramment utilisées pour décrire les comportements interactifs. Green [1986] montre ainsi que l'utilisation des systèmes de transitions augmentés et récursifs (ATN et RTN) peuvent être utilisés pour spécifier le dialogue d'une application interactive. Myers [1989, 1990] a utilisé les machines à états pour décrire le comportement de ses interacteurs (Figure 4.3). Les états sont représentés par les rectangles aux coins arrondis, les transitions par les flèches. Celles-ci sont étiquetées par les événements qui les activent et par les actions qu'elles déclenchent.

D'autres modèles essaient de remédier à certains des défauts des machines à états. L'un de ces problèmes est l'explosion combinatoire qui fait augmenter exponentiellement le nombre de transitions possibles lorsque le nombre d'états augmente. Pour résoudre ce problème, on peut introduire une hiérarchisation des états en permettant d'inclure récursivement des machines à l'intérieur des états. On obtient alors les machines à états hiérarchiques. Elles ont le même pouvoir d'expression que les simples machines à états puisqu'on peut toujours dédoubler les transitions que la hiérarchie permet de factoriser, mais elles permettent de structurer davantage la machine. Les *statecharts* de Harel [1987] proposent d'autres modes de composition des états. Ils permettent notamment d'avoir plusieurs états actifs concouramment. Wellner [1989] les a utilisés pour spécifier l'interaction dans son outil *Statemaster*. Si les *statecharts* ont un pouvoir d'expression très fort, qui leur vaut d'être utilisé comme formalisme par UML pour spécifier les comportements dynamiques, leur sémantique est complexe et parfois difficile à appréhender.

Les réseaux de Petri [1962] permettent d'augmenter l'expressivité des automates en représentant explicitement l'état du système par la position de jetons (marquage) qui évoluent de place en place. Ces jetons ne peuvent fran-

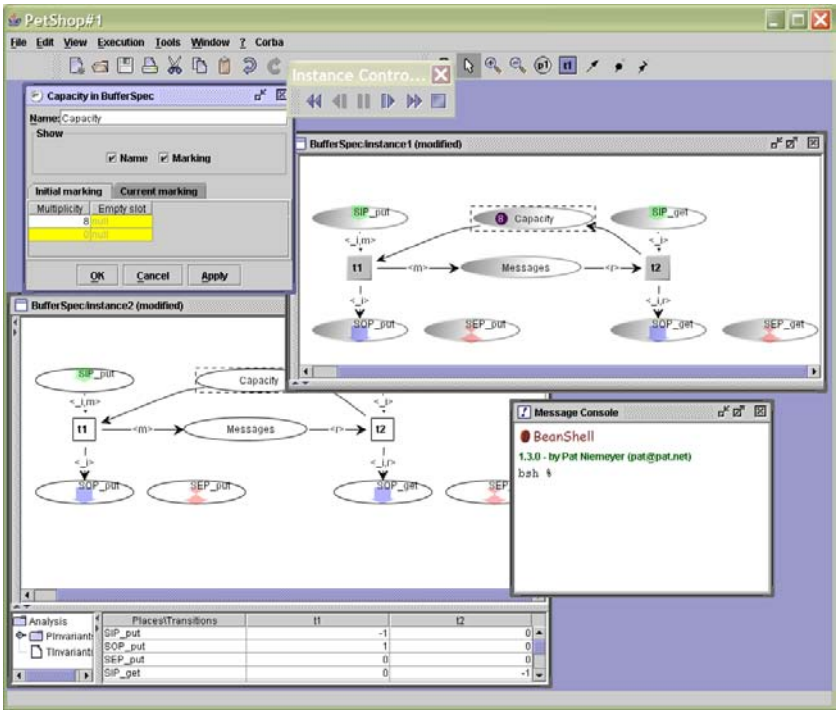


FIG. 4.4 – L'environnement PetShop

chir les transitions qui séparent les places que si celles-ci remplissent certaines conditions (nombre suffisant de jetons en particulier). Les réseaux de Petri restent dans un cadre complètement formalisé, ce qui a l'avantage de permettre de mettre en œuvre des méthodes de vérification de certaines propriétés des systèmes ainsi spécifiés. Cependant, la contrepartie de la puissance d'expression de ce formalisme est, là encore, la nécessité d'un environnement dédié, car seule la programmation visuelle est envisageable avec les réseaux de Petri. Ils sont utilisés par exemple par le formalisme ICO *(Interactive Cooperative Objects)* [Palanque et Bastide, 1993], en conjonction avec une approche par objets, pour décrire les aspects dynamiques de systèmes interactifs. Ce formalisme est intégré à un environnement de programmation visuelle, PetShop (Figure 4.4), qui permet de mettre au point et d'exécuter des systèmes spécifiés à l'aide du formalisme ICO [Bastide *et al.*, 2002].

Langages concurrents et à base de règles

Le langage *squeak* [Cardelli et Pike, 1985], de même que ERL *(Event Response Language)* proposé par Hill [1986], est un des langages à syntaxes textuelles inspiré des langages utilisés pour programmer des programmes concurrents comme CSP [Hoare, 1978]. Olsen [1990] propose aussi une notation à base de règles de productions, qui combine des aspects des langages concurrents, et la définitions d'états, il s'agit de PPS *(Propositional Production Systems)*. Il l'utilise en particulier pour décrire le comportement des interacteurs standards [Olsen, 1998, Chapitre 8]. *Squeak* définit pour sa part des

```
DoubleClick =
  DN? .
     wait[clickTime] UP? .
        wait[doubleClickTime] DN? .
           wait[clickTime] UP? . doubleClick! . DoubleClick
           || click! . down! . UP? . up! . DoubleClick
        || click! . DoubleClick
     || down! . UP? . up! . DoubleClick
```

FIG. 4.5 – Le double-clic exprimé dans le langage *squeak*

(Listing extrait de [Cardelli et Pike, 1985])

canaux permettant de communiquer avec les périphériques : position de la souris, changements d'état des boutons, caractères saisis à l'aide du clavier. Il permet aussi de définir des zones rectangulaires et d'être notifié lorsque le curseur y entre ou en sort. Il permet enfin de fixer des délais d'attente. La Figure 4.5 montre un processus *squeak* qui, à partir des informations de changement d'état d'un bouton (vers le haut `UP` et vers le bas `DN`), génère des événements `doubleClick` s'ils respectent un certain motif temporel contrôlé par des délais ; et qui, sinon, émet les événements `click`, `up` et `down` pour qu'un autre processus puisse les traiter. Ce langage a l'avantage d'avoir une sémantique formellement définie. On peut mettre à son crédit sa concision, mais la mise au point d'interactions complexes paraît délicate pour les non-experts de ce formalisme.

Compromis

Parmi les modèles présentés, nous en avons choisi un sur la base d'un compromis dont les critères principaux sont :

- la *puissance* d'expression ;
- la *simplicité* de la sémantique ; et
- le *degré d'intégration* avec le langage de programmation.

Pour la simplicité, nous avons choisi les machines à états, en utilisant la variante qui limite son principal inconvénient en permettant une meilleure structuration des états : les machines à états hiérarchiques (HSM pour *Hierarchical State Machines*). Ce choix est également motivé par le fait qu'on peut faire de ces machines à états hiérarchiques une représentation textuelle. Nous verrons que cette syntaxe permet de lever implicitement certaines ambiguïtés présentes dans les syntaxes visuelles en introduisant un ordre naturel sur les divers éléments des machines à états hiérarchiques. Cette représentation textuelle permet aussi d'intégrer facilement le formalisme au reste du processus de création d'application, et notamment au langage de programmation lui-même.

4.1.2 Introduction au formalisme des machines à états hiérarchiques

Afin de présenter le formalisme des HSM, nous montrons ici sur un exemple comment programmer le comportement d'un objet interactif simple : celui d'un bouton. Ce bouton se comporte de la manière suivante :

– lorsqu'on enfonce le bouton de la souris, le curseur étant positionné à l'intérieur du bouton, un retour visuel montre qu'il est enfoncé ;
– lorsque l'on relâche le bouton de la souris en étant toujours à l'intérieur du bouton, l'action associée au bouton est déclenchée et l'aspect visuel du bouton revient à l'état haut ;
– si le curseur de la souris sort du bouton alors que son bouton est enfoncé, l'aspect visuel revient à l'état haut sans que l'action ne soit déclenchée ; et enfin
– si le curseur réentre à l'intérieur du bouton sans que le bouton de la souris n'ait été relâché alors que le curseur était à l'extérieur du bouton, celui-ci reprend son aspect enfoncé et il est possible de déclencher l'action en relachant le bouton de la souris à l'intérieur du bouton.

Ce comportement, qui correspond à celui le plus communément admis pour un simple bouton [Olsen, 1998, Chapitre 8], n'est pas trivial à réaliser. En effet, il ne suffit pas de connaître l'état courant (bouton de la souris enfoncé ou non, curseur dans le bouton ou non) pour déterminer si l'action doit être déclenchée ou non, il est aussi nécessaire de connaître son histoire (le curseur est sorti du bouton avec le bouton de la souris enfoncé).

Dans la boîte à outils d'interacteurs Java Swing, qui est couramment utilisée pour construire des interfaces graphiques aux applications écrites en Java, la classe qui gère le comportement du bouton est définie dans un fichier qui compte environ 2900 lignes (`javax/swing/AbstractButton.java`[1]). Ce fichier comporte 1300 lignes de commentaires, et un peu plus de 100 lignes blanches. Il comporte donc environ 1500 lignes de code Java. Un comptage permet d'établir qu'environ une ligne sur dix, soit 150 lignes, sont consacrées à la gestion du comportement décrit ci-dessus. Ces lignes servent d'abord à maintenir l'état du bouton, et à fournir un accès à cet état. Elles sont aussi distribuées au sein de tout le reste du code qui est truffé de tests permettant de choisir les traitements appropriés à effectuer en fonction de l'état courant du bouton. Ces ordres de grandeurs sont similaires pour une bibliothèque d'interacteurs écrite en C++ comme la bibliothèque QT[2].

Première version du bouton

Si on se préoccupe uniquement de gérer correctement l'enfoncement et le relâchement du bouton, celui-ci ne possède alors que deux états qui correspondent à son aspect visuel. Une simple machine à états permet alors de décrire son comportement. La Figure 4.6 montre à gauche les deux états graphiques du bouton et à droite la machine à états qui ajoute à ces états les transitions activées lorsqu'on presse et qu'on relâche le bouton. La Figure 4.7 montre le document SVG correspondant à la représentation graphique précédente.

Pour prendre en compte les subtilités du comportement du bouton liées à l'entrée et à la sortie du curseur du bouton décrites ci-dessus, ajouter des gardes sur les transitions de la machine proposée sur la Figure 4.6 n'est pas suffisant. En effet, l'état visuel du bouton ne correspond plus directement à celui de la souris (le bouton doit apparaître relâché lorsque le curseur en sort, même si le bouton de la souris est toujours maintenu enfoncé), ni même à une simple combinaison logique de l'état du bouton de la souris et de la position

[1] Les sources de Java Swing sont disponibles sur le site de Sun (`http://java.sun.com/`).
[2] QT est distribuée par TrollTech (`http://www.trolltech.com/products/qt/`).

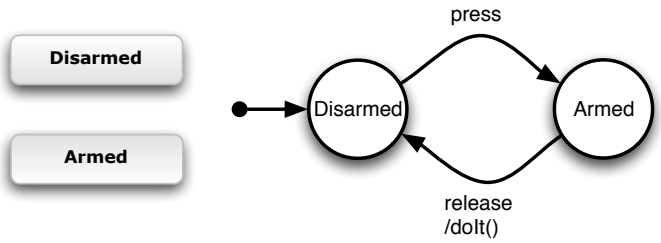


FIG. 4.6 – Aspects graphiques et comportement simplifié d'un bouton

```
01 <svg xmlns:hsm='hsm.insitu.lri.fr'>
02    <defs>
03       <!-- dégradés de fond, détails omis -->
04       <linearGradient id='armed' ... />
05       <linearGradient id='disarmed' ... />
06    </defs>
07
08    <!-- définition du bouton -->
09    <g hsm:behaviour='Button'
10       style='font-size:10; font-weight: bold'>
11
12       <g><!-- version armée -->
13          <rect width='80' height='20' rx='4' ry='4'
14                style='fill:url(#armed); stroke:gray;' />
15          <text x='19' y='13'>Armed</text>
16       </g>
17
18       <g><!-- version desarmée -->
19          <rect width='80' height='20' rx='4' ry='4'
20                style='fill:url(#disarmed); stroke:gray;'/>
21          <text x='19' y='13'>Disarmed</text>
22       </g>
23    </g>
24 </svg>
```

FIG. 4.7 – Le document SVG définissant l'aspect du bouton, annoté pour le lier à son comportement

du curseur (deux états sont possibles quand le curseur est dans le bouton et que le bouton de la souris est enfoncé suivant qu'il l'a été à l'intérieur ou à l'extérieur). Il est donc nécessaire d'ajouter des états à cette machine. Cependant, s'ils sont ajoutés en conservant le formalisme des machines à états, le lien entre l'état visuel du bouton et son état interne devient difficile à percevoir.

Raffinement du comportement à l'aide des machines à états hiérarchiques

En introduisant une hiérarchie d'états, on peut conserver au premier niveau les deux états correspondants aux deux représentations du bouton. Ceux-ci peuvent alors être raffinés en leur ajoutant des sous-états qui prennent en compte les événements *enter/leave* (le curseur entre et sort du bouton) comme les événements *press/release* liés au bouton gauche de la souris. La Figure 4.8 montre une représentation de cette machine à états hiérarchique, en utilisant une notation graphique inspirée de celle des *statecharts*.

Sur la Figure 4.9, le code correspondant à ce comportement est donné en utilisant la syntaxe proposée par HsmTk pour spécifier les machines à états hiérarchiques. Sa structure générale est celle du diagramme précédent. Les

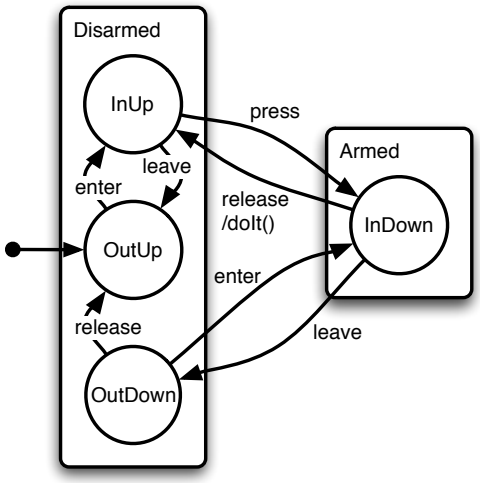


FIG. 4.8 – Comportement raffiné d'un bouton

états sont inclus les uns dans les autres en utilisant des blocs délimités par des accolades comme cela est naturel dans le langage de programmation hôte. La machine `Button` définit donc ses deux sous-états, `Disarmed` (ligne 2 à 32) et `Armed` (ligne 34 à 41). Le premier commence par la définition d'un ensemble de variables (lignes 3 à 6) qui sont liées à l'initialisation de la machine avec le fragment de SVG auquel le comportement est attaché (lignes 8 à 13). Cette initialisation utilise les fonctionnalités de HsmTk décrites à la Section 3.2.3, page 55, pour manipuler le graphe SVG et exprimer le contrat structurel que doit respecter la représentation d'un bouton. Il récupère ainsi les deux groupes SVG qui définissent chacun un aspect visuel du bouton (Figure 4.7, lignes 12 et 18). La synchronisation entre l'état interne du bouton et celui de sa représentation est alors simplement assurée en modifiant le document SVG suivant que l'état `Disarmed` devient actif (action `enter`, ligne 16) on inactif au profit de l'état `Armed` (action `leave`, ligne 17).

Le reste du code spécifie simplement les sous-états et les transitions qui permettent de passer de l'un à l'autre. Par exemple, l'état `OutUp` (lignes 19 à 21) qui est actif lorsque le curseur est à l'extérieur du bouton comporte uniquement une transition (ligne 20). Celle-ci est franchie lorsque le curseur pénètre à l'intérieur du bouton (`enter()`), et l'état actif change en passant à `InUp`, la cible déclarée de cette transition. Une transition particulière est définie ligne 39. Elle est franchie alors que le bouton est armé et que le bouton de la souris est relâché à l'intérieur de celui-ci. Lors de ce franchissement, la machine émet un événement (`broadcast`) signalant qu'il faut déclencher l'action. C'est là un moyen de communiquer avec les autres composants du programme.

4.1.3 Conclusion

Nous avons vu que malgré les propositions nombreuses de langages adaptés à la description de comportements interactifs, peu répondent à nos critères pour

```
01  hsm Button {
02     hsm Disarmed {
03        // variables
04        var svg::SVGElement *elem;
05        var svg::SVGGElement *armedLook = 0;
06        var svg::SVGGElement *disarmedLook = 0;
07
08        // initialisation
09        init {
10           hsm::svgl::Childs(elem).get(armedLook)
11                                  .get(disarmedLook);
12           elem->removeChild(disarmedLook);
13        }
14
15        // maintient de l'état visuel du bouton
16        enter { elem->replaceChild(armedLook, disarmedLook); }
17        leave { elem->replaceChild(disarmedLook, armedLook); }
18
19        hsm OutUp {
20           - enter() > InUp
21        }
22
23        hsm InUp {
24           - leave() > OutUp
25           - press() > Armed::InDown
26        }
27
28        hsm OutDown {
29           - enter()   > Armed::InDown
30           - release() > OutUp
31        }
32     }
33
34     hsm Armed {
35        hsm InDown {
36           - leave() > Disarmed::OutDown
37
38           // invocation de l'action
39           - release() broadcast(DO_IT) > Disarmed::InUp
40        }
41     }
42  }
```

FIG. 4.9 – Le comportement d'un bouton spécifié par une machine à états hiérarchique

programmer ces mêmes comportements. L'exemple des diverses réalisations du bouton nous a montré comme un langage impératif est peu adapté à décrire ce type de comportements.

En encodant l'état, non pas dans la valeur de variables, mais dans le flot de contrôle du programme, l'utilisation d'un formalisme basé sur les machines à états permet de rendre explicites les transitions qui déclenchent ces changements d'état. Par ailleurs, l'intérêt d'introduire la hiérarchie au sein de ce formalisme apparaît déjà avec le simple exemple du bouton présenté. Elle permet en effet de structurer le code et de raffiner le comportement.

4.2 Le formalisme des HSM

Divers formalismes adaptés à la description des comportements dynamiques et à la spécification de l'interaction existent déjà, comme nous l'avons vu. Nous avons choisi d'utiliser les machines à états hiérarchiques pour ajouter une structure de contrôle au langage de programmation impératif utilisé par

les programmeurs utilisant notre boîte à outils. Elles permettent d'inclure une machine à états récursivement à l'intérieur de chaque état de toute machine. Ces machines sont activées uniquement lorsque l'état qui les contient devient actif. Cette décomposition hiérarchique permet en particulier de pallier un défaut bien connu des machines à états, l'explosion combinatoire sur les transitions qui se produit lorsque le nombre d'états augmente, ce qui rend difficile la modification, même minime, d'une telle machine. La hiérarchie introduite par les HSM permet de modulariser le code, d'encourager sa réutilisation et de faciliter les modifications.

Nous avons choisi une syntaxe textuelle aux HSM, ce qui nous permet de les utiliser comme extension du langage de programmation impératif utilisé par la boîte à outils. Cette approche nous permet d'intégrer au cœur même de l'environnement de programmation habituel l'extension des HSM, et maximise ainsi les possibilités de son utilisation. Nous présentons maintenant la syntaxe et la sémantique de cette extension du langage. Comme nous étendons le langage C++, nous utilisons ses définitions des types, des identificateurs et des expressions.

4.2.1 États

Un *état* comporte un nom, il peut définir des variables et des entrées, des actions à effectuer lors de l'initialisation, de l'activation, et de la désactivation. Il peut contenir des sous-états, des transitions, ainsi que d'autres éléments décrits ci-après. La syntaxe permettant de définir un état est :

```
hsm Name { content }
```

où `Name` est un identificateur commençant par une majuscule et `content` est formé d'autant de déclarations décrites ci-dessous que l'on veut, ainsi qu'éventuellement de sous-états.

État initial

Quand plusieurs états sont définis à l'intérieur d'une machine parente[3], le premier état dans l'ordre des définitions est appelé *état initial* de la machine. Quand une machine est activée (nous verrons le mécanisme d'activation plus bas), son état initial devient son état courant, et il est activé à son tour — ce processus étant récursif jusqu'à aboutir sur un état n'ayant pas de sous-états.

Il est cependant possible de spécifier un autre état initial lors de l'activation de la machine, notamment pour pouvoir la mettre dans un état qui soit cohérent avec celui de ses entrées. Pour ce faire, on peut spécifier un ensemble de règles qui indiquent l'état initial de la machine :

```
[condition] : Name
```

où `Name` est le nom d'un sous-état de la machine, et `condition` peut être n'importe quelle expression booléenne comportant des variables ou des entrées qui sont visibles à l'intérieur de la machine.

[3] La notion d'état et de machine est confondue dans ce formalisme puisque la définition d'un état est récursive. Nous utiliserons de manière générale le terme "état", mais parfois le terme "machine" lorsqu'il est fait référence à l'état parent de l'état dont il est question.

Variables

Un état peut déclarer et définir des *variables* de tous types. Les variables ainsi déclarées sont visibles dans toutes les actions de l'état lui-même. Les règles de visibilité des variables dans l'arbre des états sont particulières :

- les variables d'un état ne sont pas visibles de ses sous-états ;
- les variables liées, c'est-à-dire déclarées et définies, d'un état ne sont pas visibles par ses ancêtres ; et
- les variables libres, c'est-à-dire déclarées mais non définies dans un état, sont identifiées à une variable de même nom et de même type — déclarée implicitement si besoin — dans la machine parente, ce processus propageant la variable vers le haut de la machine tant qu'aucun état ne définit cette variable explicitement.

L'instanciation de la machine globale requiert pour finir la définition de toutes les variables non initialisées qu'elle contient.

Pour déclarer une variable, la syntaxe utilisée est la suivante :

```
var type identifier;
```

et pour définir une variable, la syntaxe utilisée est l'une des suivantes :

```
var type identifier = value;
var type identifier(value, ...);
```

où `type` et `identifier` sont respectivement un type et un identificateur valides pour le langage, et `value` une ou des valeurs permettant d'intialiser ou de construire la variable concernée.

La particularité des règles de visibilité des machines à états hiérarchiques en fait un objet hybride entre structure de contrôle — pour lesquelles la portée des variables s'étend à tous les blocs inclus — et structure de données — pour lesquelles les règles de visibilité permettent d'accéder aux membres d'une structure incluse. Les variables liées d'un état sont analogues aux variables locales d'une fonction. Dans cette analogie, les variables libres correspondent aux arguments, passés par référence, puisqu'elles sont spécifiées à l'extérieur de l'état. Elles permettent en fait à la machine englobante de paramétrer l'état, si celui-ci l'autorise.

Entrées

Une *entrée* est une variable particulière qui peut déclencher des transitions au sein de la machine. Elle possède exactement les mêmes caractéristiques de visibilité qu'une variable normale. La spécificité des entrées est que leur type est implicitement défini, il s'agit d'un pointeur sur un composant élémentaire de HsmTk puisque les machines à états hiérarchiques utilisent leur système d'événements pour activer les transitions. Les entrées peuvent donc être des objets instances de toute classe dérivant du composant de base. Une entrée se déclare ou se définit ainsi :

```
in identifier;
in identifier = value;
in identifier(value);
```

Méthodes

Un état peut définir des *méthodes* pour permettre la factorisation du code. Ces méthodes peuvent être appelées partout où il est possible d'utiliser du code dans l'état qui les définit. Ces méthodes sont donc locales et se définissent en apposant le mot clé `local` avant une définition usuelle :

```
local type name(type var, ...) { code }
```

4.2.2 Transitions

Les *transitions* entre états sont déclenchées par les événements qui sont envoyés à la machine. Les transitions sont spécifiées par des règles ordonnées. Une règle de transition se décompose sous la forme suivante :

```
- input.EVENT [condition] { code } broadcast(EVENT) > Target(var = value, ...)
```

où `input` permet de spécifier de quelle entrée on attend des événements. Si elle est omise, les événements venant de n'importe quelle entrée et ayant le type spécifié permettront de franchir la transition. Si l'entrée n'a pas été déclarée dans la machine, elle est ajoutée implicitement, sans définition, à ses entrées.

La donnée d'`EVENT` spécifie le type d'événement qui peut déclencher la transition. Des types d'événements prédéfinis sont donnés pour notifier de changements usuels : création, destruction ou modification d'un composant. Il est possible de capturer tous les types d'événements en utilisant l'étoile (`*`) en lieu et place du type d'événement. On peut aussi omettre le type d'événement attendu, ce qui revient implicitement à attendre le plus courant d'entre-eux : la notification d'une modification. L'événement qui a déclenché la transition est accessible dans le code qui lui est associé grâce à la variable `event`. Celle-ci comporte plusieurs informations dont un pointeur sur le composant qui a émis l'événement, le type de ce dernier, et une valeur (caractère, booléen, entier ou réel) qui est utilisée pour transmettre une information simple. Cette valeur donne, lorsque cela est possible, l'écart entre l'ancien état et le nouvel état atteint lors du changement signalé par l'événement. Si cette information est impossible à transmettre à l'aide d'une simple valeur, le fait d'avoir à disposition un pointeur sur le composant qui a émis l'événement permet de l'interroger directement pour connaître son nouvel état.

Une clause entre crochets spécifie une garde à la transition. Elle est optionnelle. La `condition` booléenne donnée doit être vraie pour que la transition puisse avoir lieu. Cette condition doit pouvoir être évaluée dans le contexte de l'état courant. Plusieurs gardes peuvent être spécifiées, dans ce cas, leur conjonction doit être vraie pour que la transition puisse être franchie.

Le `code` spécifié entre accolades peut être n'importe quelle combinaison d'instructions du langage valide dans le contexte de l'état. Il est exécuté lorsque la transition est franchie. Cette exécution a lieu dans le contexte de l'état courant, avant de le quitter. Comme les gardes, le code est optionnel.

Le mot clé `broadcast` permet de préciser que la machine doit émettre un événement lors du franchissement de la transition. Cet événement est émis à destination des composants abonnés à la machine à états juste après l'exécution du code associé à la transition et juste avant de quitter l'état courant de la machine. Cette partie de la transition est elle aussi optionnelle.

La dernière partie de la transition précise par son nom l'état qui deviendra l'état courant de la machine une fois la transition effectuée. La résolution du nom s'effectue en suivant les règles des espaces de noms du C++. On peut ainsi spécifier un sous-état d'un état frère en qualifiant le nom du sous-état par celui de son père. La Figure 4.9 en fournit plusieurs exemples dont la transition `press` de l'état `InUp` ayant pour cible `Armed::InDown`. Enfin, il est possible de passer des valeurs aux variables de l'état cible en les précisant à la suite de leur nom.

Si la cible d'une transition est l'état courant, celui-ci est quitté puis réentré, le code associé à ces actions étant alors exécuté (on verra ci-dessous comment associer du code à l'entrée et à la sortie d'un état). Par contre, si aucune cible n'est donnée à la transition, la machine reste dans son état courant, sans le quitter puis y revenir, et seul le code associé à la transition est exécuté.

Transitions particulières

Deux types de transitions peuvent être déclenchées autrement que par la réception d'un événement : celles qui le sont par l'appel explicite d'une méthode associée à la machine, et celles déclenchées par l'expiration d'un délai. Pour les premières, il est possible de spécifier des arguments pour la méthode et ces arguments sont alors accessibles à l'intérieur du code de la transition. Pour les secondes, la durée du délai, exprimée en millisecondes, est comptée à partir du moment où l'état est devenu actif. Elles ne peuvent se déclencher que si aucune autre transition provoquant la sortie de cet état n'a eu lieu entre-temps. Dans les deux cas, seule la première partie de la règle de transition est particulière, le reste de la définition pouvant comporter les mêmes éléments que les transitions normales décrites ci-dessus :

```
- method(type var, ...) { code } > ...
- ms > ...
```

4.2.3 Prise en compte des événements

Le traitement des événements commence par la machine située à la racine de la hiérarchie des états. Si cet événement ne peut déclencher de transition à ce niveau, il est passé au sous-état englobant l'état courant de la machine[4]. Cette descente le long du chemin vers l'état courant se poursuit jusqu'à ce qu'une transition puisse être franchie. Si aucune transition ne peut l'être, l'événement est ignoré par la machine. Si une transition est franchie, celle-ci consomme alors l'événement.

Lorsqu'une transition est franchie, l'état cible est activé après un processus se déroulant en trois étapes :

1. D'abord, l'état à partir duquel a eu lieu la transition est désactivé. Ce processus commence par le sous-état courant le plus profond, qui est

[4] Dans les *statecharts*, les événements se propagent au contraire de bas en haut. Ce choix permet de définir les sous-états comme des spécialisations de leurs ancêtres qui raffinent leurs comportements. Nous avons fait le choix inverse dans une optique de modularité, car nous voulons pouvoir réutiliser une machine et altérer son comportement sans forcément disposer de sa définition. Pour nous, une machine n'est donc pas une abstraction de ses sous-états mais plus pragmatiquement un conteneur qui permet le multiplexage temporel de ceux-ci.

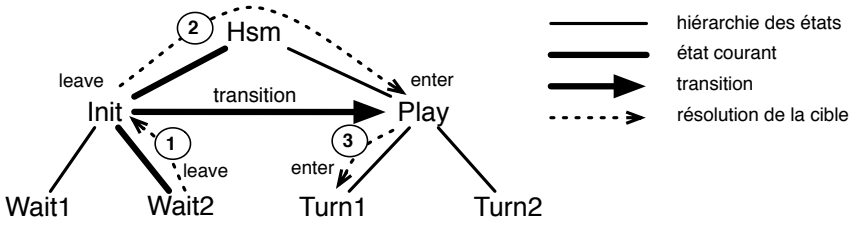


FIG. 4.10 – Mécanisme des transitions

alors désactivé, et remonte récursivement jusqu'à l'état source de la transition, en désactivant au fur-et-à-mesure les états rencontrés lors de cette remontée.

2. Ensuite, s'effectue la résolution de l'état cible. La recherche s'effectue en remontant depuis l'état source vers la machine racine jusqu'à atteindre un ancêtre commun à l'état source et à l'état cible de la transition. Lors de cette remontée, les états courants rencontrés en chemin sont eux aussi désactivés. À partir de cet ancêtre commun, les états situés sur la branche conduisant à l'état cible sont activés récursivement jusqu'à parvenir à l'état cible.
3. Enfin, l'état cible est activé. Son état initial (s'il a des sous-états) est alors activé, et ainsi de suite en descendant récursivement le long de la branche des états initiaux.

Ce processus en trois étapes est illustré sur la Figure 4.10 où les trois étapes — quitter l'état courant, trouver l'état cible, et activer l'état cible — sont étiquetées 1, 2, et 3.

Actions lors de l'activation et de la désactivation

Lors de la transition, du code peut être exécuté à chaque fois qu'un état est activé ou désactivé. C'est ainsi que dans l'exemple du bouton, l'état visuel de celui-ci est modifié en fonction de l'état interne de la machine qui gouverne son comportement (Figure 4.9). Pour chaque état, on spécifie ce code en utilisant les mots clés `enter` et `leave` :

```
enter { code }
leave { code }
```

De manière analogue, il est possible de spécifier du code qui sera effectué un fois pour toutes à la création de la machine en utilisant pour cela le mot clé `init` :

```
init { code }
```

Préconditions

Enfin, une structure permettant la gestion des cas exceptionnels est fournie, il s'agit des *préconditions*. Une précondition s'exprime à l'aide de l'une des syntaxes suivantes (le code étant optionnel) :

```
require (condition) else { code } Target(var = value, ...)
! (condition) : { code } Target(var = value, ...)
```

où l'expression booléenne `condition` est évaluée avant d'entrer dans l'état qui comporte la précondition. Si sa valeur est fausse, le code facultatif est exécuté s'il est présent, et le processus de résolution de la cible reprend avec pour nouvel objectif l'état qui est spécifié comme cible par la précondition.

Les préconditions permettent de renforcer la localité du code. Elles jouent le même rôle que les gardes sur les transitions qui ont pour cible l'état qui les déclarent. Cependant, à l'inverse de ces dernières qui sont évaluées dans le contexte de l'état de départ, les préconditions le sont dans le contexte de l'état cible, et lui permettent ainsi de poser des contraintes que les états qui cherchent à l'atteindre ne peuvent pas forcément vérifier eux-mêmes. Elles fournissent ainsi un mécanisme équivalent à celui des exceptions en détournant le flux de contrôle normal dans certains cas particuliers. Elles permettent généralement de simplifier l'expression des transitions.

4.3 Exemples

Le formalisme des machines à états hiérarchiques permet de fournir un support à certains aspects nécessaires à l'interaction que nous avons dégagés lors de l'état de l'art des techniques d'interactions avancées. Nous présentons ici, à titre d'illustration, quelques exemples de ce qu'il rend possible.

Les différents exemples utilisent les machines à états hiérarchiques pour transformer les événements produits par les actions de l'utilisateur en actions de plus haut niveau, celui des protocoles d'interaction vus au Chapitre 3, Section 3.2.3. Le premier exemple montre simplement comment, à partir d'une position et d'un bouton (ceux de la souris), réaliser le déplacement d'objets dont le comportement réalise le protocole *translate*, et comment raffiner cette technique d'interaction pour contraindre géométriquement ce déplacement. Le second exemple est plus complexe. Il montre comment combiner deux techniques d'interaction, le déplacement et le redimensionnement, en une seule à l'aide d'un multiplexage proposé par un menu circulaire. Enfin, une technique d'interaction permettant d'agrandir et de réduire des objets, et basée sur un usage original du clavier est introduite. Ces exemples montrent qu'avec HsmTk, si la réalisation de techniques simples est facile, les affiner, les réutiliser, ou créer des techniques originales n'est guère plus compliqué.

4.3.1 Le déplacement

Nous présentons la technique d'interaction élémentaire de déplacement d'un objet dont le comportement réalise le protocole *translate*. Ce protocole, outre les méthodes `begin` et `end`, propose une unique méthode, `translate`, qui déplace l'objet concerné d'un vecteur passé en argument.

Principe

La technique d'interaction que nous utilisons est la plus simple de l'interaction graphique : grâce au curseur, lié à la position de la souris, on peut "attraper" les objets déplaçables en enfonçant le bouton gauche de la souris lorsque le curseur les désigne. Les déplacements du curseur translatent alors l'objet concerné tant que le bouton n'est pas relâché.

```
01 hsm Translater {
02    [*button/hsm::Button::cast] : Translating
03
04    hsm Idle {
05       - button > Translating
06    }
07
08    hsm Translating {
09       - button > Idle
10
11       hsm Op {
12          var hsm::SVGLWindow *window;
13          in point;
14
15          var hsm::Translate *op = 0;
16          var hsm::Point origin(2);
17
18          require
19             ((op = window->pick< hsm::Translate >(*point/hsm::Point::cast))
20              != 0)
21          else
22             Nop
23
24          require (op->begin()) else Nop
25
26          enter { origin = *point/hsm::Point::cast; }
27          leave { op->end(); }
28
29          - point {
30             hsm::Point delta(2);
31             delta = *point/hsm::Point::cast;
32             delta -= origin;
33             op->translate(delta);
34             origin += delta;
35          }
36       }
37
38       hsm Nop {}
39    }
40 }
```

FIG. 4.11 – HSM traduisant les actions de la souris en déplacement d'un objet

Réalisation

La Figure 4.11 montre le code qui réalise la technique d'interaction. Il consiste en une quarantaine de lignes. La première ligne déclare la HSM de haut niveau `Translater`. La ligne 2 permet de synchroniser la machine avec ses entrées lors de son initialisation en modifiant son état initial en fonction de l'état de ses entrées. En effet, la machine `Translater` comporte deux états : `Idle` et `Translating` qui sont actifs respectivement quand le bouton de la souris est relâché et quand il est enfoncé. Si, lors de l'initialisation de la machine, le bouton est enfoncé, l'état initial devient alors `Translating`.

Les lignes suivantes (4 à 6) définissent l'état `Idle`. Dans cet état, il ne se passe rien tant qu'aucun événement provenant du bouton n'est reçu. La ligne 5 attend un tel événement et spécifie que la machine passe dans l'état `Translating` lors de sa réception. L'état `Translating` est défini à partir de la ligne 8. Il commence par définir une transition semblable à la précédente, mais ayant pour cible l'état `Idle`. Ainsi, les événements venant du bouton font basculer alternativement de l'état `Idle` à l'état `Translating`. Ces deux transitions, associées à la condition de synchronisation vue ci-dessus, nous assurent ainsi de la cohérence de ces états avec l'état du bouton de la souris.

```
01 hsm::SVGLWindow *window = new hsm::SVGLWindow("test");
02 window->setDocument("test.svg");
03
04 hsm::Device *pointer = hsm::repository::getDevice("Pointer");
05
06 Translater::Hsm translater(window,
07                            *pointer/"buttons"/hsm::ButtonPad::L,
08                            *pointer/"position");
```

FIG. 4.12 – Initialisation de la machine à états

Le sous-état initial de `Translating`, `Op` (ligne 11), réalise l'opération de translation proprement dite. Il commence par définir les variables et l'entrée dont il a besoin (lignes 12 à 16). Il définit ensuite deux préconditions qui doivent être vérifiées préalablement à l'entrée dans cet état :

- pour les lignes 18 à 22, il faut qu'un objet dont le comportement réalise le protocole `Translatable` soit présent dans la fenêtre sous le point initial de l'interaction ; et
- pour la ligne 24, il faut que cet objet accepte l'interaction dont le début lui est signifié à l'aide du protocole élémentaire *begin/end*.

Si l'une de ces conditions n'est pas remplie, la machine passe dans l'état `Nop` (ligne 38) dans lequel il ne se passe plus rien. Les états `Op` et `Nop` permettent donc de rester dans l'état `Translating` en accord avec le bouton de la souris, mais en effectuant ou non l'interaction de translation suivant qu'un objet réceptif à l'interaction est trouvé ou non sous le curseur.

Dans le cas où un tel objet est trouvé, l'état `Op` devient effectivement actif. La ligne 26 permet alors de mémoriser l'origine du déplacement. La ligne 27 permettra de terminer l'interaction lorsque l'état sera quitté. La transition définie ensuite, lignes 29 à 35, est invoquée à chaque modification de la position du curseur. Elle déplace l'objet d'un vecteur correspondant à la différence entre la position actuelle du curseur et sa position à l'étape précédente. Cette transition ne modifie pas l'état courant, puisqu'elle n'a pas d'état cible ; les actions `enter` et `leave` ne sont donc pas activées lors de son franchissement.

On peut remarquer qu'aucune transition ne permet de sortir de l'état `Op`. C'est en fait la transition de `Translating` (ligne 9) qui provoquera la sortie de cet état (ou celle de son frère `Nop`) lorsque le bouton sera relâché. Le code de l'action `leave` appartenant à l'état `Op` (ligne 27) sera alors invoqué et terminera l'interaction avec l'objet actuel.

Utilisation

La machine à états hiérarchique, une fois définie, peut-être utilisée en donnant explicitement ses entrées. La Figure 4.12 comporte le code minimal qui utilise cette machine à états pour interagir avec le contenu d'une fenêtre chargé à partir d'un fichier SVG. Les lignes 1 à 3 créent cette fenêtre et y chargent un document SVG à partir d'un fichier. La ligne 5 récupère le composant du pointeur système (la souris par défaut). Enfin, les lignes 7 à 9 initialisent la machine à états en lui passant ses variables et ses entrées non initialisées. Ces dernières, lignes 8 et 9, spécifient que l'on va utiliser le bouton gauche de la souris et sa position pour activer et contrôler le déplacement.

```
01 hsm ConstrainedTranslater {
02    [*shift/hsm::Button::cast] : Constrained
03
04    in point;
05    in p = point;
06
07    hsm Normal {
08       hsm Translater;
09       - shift > Constrained
10    }
11
12    hsm Constrained {
13       var hsm::Point constr(2);
14       in point = &constr;
15       var hsm::Point origin(2);
16
17       enter { constr = origin = *p/hsm::Point::cast; }
18
19       local void constrain(hsm::Point &c,
20                            const hsm::Point &o,
21                            const hsm::Point &p) { /* détails omis */ }
22
23       - p { constrain(constr, origin, *p/hsm::Point::cast); }
24
25       hsm Translater;
26       - shift > Normal
27    }
28 }
```

FIG. 4.13 – Spécialisation du déplacement en déplacement contraint

Réutilisation

La technique précédente est simple. Elle peut cependant servir de base pour réaliser des variantes plus complexes. Il est par exemple possible de la réutiliser telle quelle en l'englobant au sein d'une autre machine pour altérer son comportement. Ici, le déplacement deviendra contraint selon les axes horizontaux et verticaux dès que la touche *shift* sera enfoncée, reproduisant ainsi une contrainte classiquement proposée dans les éditeurs graphiques.

La Figure 4.13 montre cette extension de la technique précédente. La machine principale `ConstrainedTranslater` comporte deux sous-états, `Normal` et `Constrained`. L'appui de la touche *shift* permet de passer de l'un à l'autre, comme le permettait le bouton dans l'exemple précédent. Chacun des états inclut la machine `Translater` telle quelle (lignes 8 et 25), sans la redéfinir. L'état `Normal` (lignes 7 à 10) se limite à cette inclusion puisqu'il ne modifie pas son comportement.

L'état `Constrained` (lignes 12 à 27) est un peu plus compliqué puisqu'il doit adapter la position qui sert d'entrée à sa version de la machine `Translater` pour la contraindre. L'identification des entrées d'une machine reposant sur leur nom, la position contrainte `constr` (définie ligne 13) doit être identifiée à l'entrée `point` pour être prise en compte par la machine `Translater`. Cette identification est faite ligne 14. Cette position contrainte est alors mise à jour dès que la position d'entrée de la machine `p` est modifiée grâce à la transition de la ligne 23, ce qui alimente la machine `Translater` en événements. La contrainte est exprimée grâce à une méthode locale à l'état qui n'est pas détaillée ici mais dont le prototype est donné ligne 19.

Ainsi, à l'intérieur de la machine `ConstrainedTranslater`, et de l'état `Normal`, l'entrée `point` fait référence à la position du curseur, et `p` est un

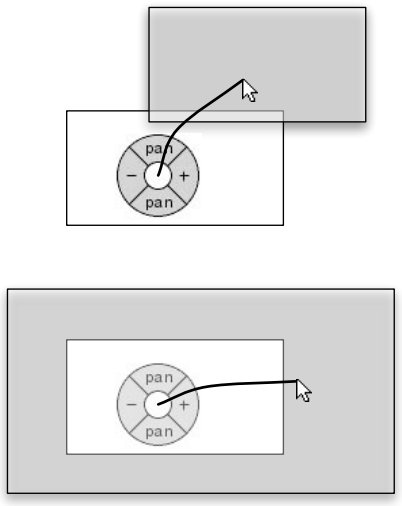


FIG. 4.14 – Déplacement (haut) et redimensionnement (bas) multiplexés à l'aide d'un *control menu*

simple alias de cette entrée défini ligne 5. Dans l'état `Constrained`, cet alias reste visible, ce qui permet de connaître les déplacements réels du curseur, mais l'entrée `point` est volontairement masquée par le point contraint. Grâce à ce masquage, c'est ce point contraint qui est utilisé comme entrée pour le sous-état `Translater`. L'objet est alors manipulé en respectant un déplacement strictement horizontal ou strictement vertical suivant la direction dans laquelle l'amplitude du mouvement du curseur depuis l'origine est la plus importante.

4.3.2 Le déplacement/redimensionnement multiplexé

Nous présentons maintenant une technique d'interaction plus complexe qui combine le déplacement précédent et le redimensionnement. Ces deux interactions sont multiplexées sur un seul périphérique d'entrée en utilisant un *control menu*, technique d'interaction présentée à la Section 2.2.1, page 28.

Principe

Cette technique présente un menu circulaire lorsque le bouton de la souris est enfoncé, et suivant que le mouvement amorcé est orienté verticalement ou horizontalement, l'interaction déclenchée est une translation (Figure 4.14 haut) ou une mise à l'échelle (Figure 4.14 bas). Cette interaction se déclenche dès que l'un des éléments du menu est atteint, sans qu'il soit nécessaire de cliquer sur l'élément. Le menu disparaît alors et l'interaction s'enchaîne de manière fluide dans un seul geste qui spécifie :

- l'*objet* de l'interaction, par son origine ;
- l'*interaction* par la direction de son déplacement initial ; et
- le *paramètre* de cette interaction par la suite du mouvement.

```
01 hsm Controler {
02    [*button/hsm::Button::cast] : Do
03
04    hsm Idle {
05       - button > Do
06    }
07
08    hsm Do {
09       - button > Idle
10
11       hsm Choose {
12          var hsm::SVGLWindow *window;
13          var hsm::Point o(2);
14
15          var hsm::Translate *tl = 0;
16          var hsm::Zoom      *zm = 0;
17
18          var hsm::SVGLCursor *menu = 0;
19
20          enter {
21             o  = *point/hsm::Point::cast;
22             tl = window->pick< hsm::Translate >(o);
23             zm = window->pick< hsm::Zoom      >(o);
24             menu = 0;
25          }
26
27          - 500 { menu = new hsm::SVGLCursor(o, window, getMenuSVG(tl, zm)); }
28
29          leave { if(menu != 0) delete menu; }
30
31          - point [abs(o/hsm::Point::Y - *point/hsm::Point::Y) > 10]
32          > Translate(translate = tl)
33
34          - point [abs(o/hsm::Point::X - *point/hsm::Point::X) > 10]
35          > Zoom(zoom = zm)
36       }

37       hsm Translate { /* détails omis */ }
38       hsm Zoom      { /* détails omis */ }

39       hsm Nop {}
40    }
41 }
```

FIG. 4.15 – HSM définissant un *control menu*

Ce geste s'apprenant facilement, il n'est pas forcément nécessaire de montrer le menu, ce qui permet aux experts d'utiliser cette technique comme une reconnaissance de geste.

Réalisation

La réalisation de cette technique à l'aide des machines à états hiérarchiques et de HsmTk est donnée sur la Figure 4.15. Certains détails sont omis car ils reprennent la technique d'interaction précédente pour la translation et sont très similaires pour le zoom (lignes 37 et 38). En incluant tout le code, cette technique d'interaction nécessite une centaine de lignes pour sa réalisation.

Comme pour le déplacement simple, la machine principale (`Controler`) est subdivisée en deux états, l'un d'attente (`Idle`), et l'autre actif (`Do`). Le passage de l'un à l'autre de ces états est gouverné par l'action du bouton de la souris. Les dix premières lignes servent donc, comme pour la technique précédente, à établir la synchronisation entre ces états et l'état du bouton de la souris, à l'aide d'une condition de synchronisation et de deux transitions.

L'état actif `Do` est, quant à lui, plus complexe, puisqu'il comporte quatre sous-états. Son état initial, `Choose` gère le retour visuel approprié et la détermination de l'interaction entamée. Lorsque cet état devient actif, il commence par mémoriser l'origine de l'interaction, et recherche à cet endroit un objet déplaçable et un objet redimensionnable (lignes 21 à 23). Ces informations permettent de fournir le retour visuel approprié en présentant un menu dans lequel les actions impossibles n'apparaissent que grisées. Le menu est créé uniquement si l'état `Choose` reste actif plus de 500 millisecondes. À l'expiration de ce délai, la transition de la ligne 27 est franchie, ce qui affiche le menu à l'endroit du clic, avec un aspect paramétré par la compatibilité des objets à cet endroit. Le menu est détruit s'il a été créé lors de la sortie de cet état (ligne 29), c'est-à-dire lorsque le choix de l'interaction est effectué.

L'état `Choose` effectue ce choix grâce aux deux transitions qui terminent sa définition (lignes 31 à 35). Celles-ci vérifient à chaque déplacement du curseur que son amplitude depuis l'origine n'excède pas dix pixels horizontalement et verticalement. Dès que cette limite est franchie, l'une des transitions est activée et fait ainsi basculer vers l'état qui réalise le déplacement (`Translate`, ligne 37) ou le redimensionnement (`Zoom`, ligne 38). Au passage, les transitions passent à leur état cible l'objet de l'interaction correspondant, ce qui leur évite de refaire un *picking*.

Les états `Translate` et `Zoom` ont sensiblement la même structure que l'état `Op` de la technique d'interaction précédente, hormis qu'ils n'ont justement pas à réaliser le *picking*. Comme cet état, ils basculent dans l'état `Nop` (ligne 39) s'il n'y a pas d'objet acceptant l'interaction à cet endroit, attendant alors que le bouton soit relâché pour revenir dans l'état d'attente `Idle`.

4.3.3 Le zoom continu au clavier

La plupart des logiciels de création graphique offrent la possibilité de choisir l'échelle à laquelle est représenté le document en cours d'édition. Le nombre de techniques pour réaliser cette tâche est surprenant. Nous en présentons quelques-unes, tirées d'une application réelle, puis nous présentons une technique originale réalisée à l'aide des machines à états hiérarchiques.

Les techniques de zoom

On peut dénombrer dans une application professionnelle de retouche photographique comme Adobe Photoshop au moins sept moyens distincts pour changer l'échelle du document courant :

- la saisie directe de l'échelle désirée (Figure 4.16a et b) ;
- l'incrémentation et la décrémentation par un facteur fixé de l'échelle par l'intermédiaire des commandes "Agrandir" et "Réduire" du menu "Affichage" ;
- l'invocation de ces mêmes commandes à l'aide de raccourcis clavier ;
- l'utilisation de deux boutons accolés à la vue radar permettant de naviguer dans le document (Figure 4.16c et d) ;
- l'utilisation d'un potentiomètre accolé à la vue radar permettant de régler, avec un retour visuel immédiat, le facteur de zoom désiré (Figure 4.16e) ;

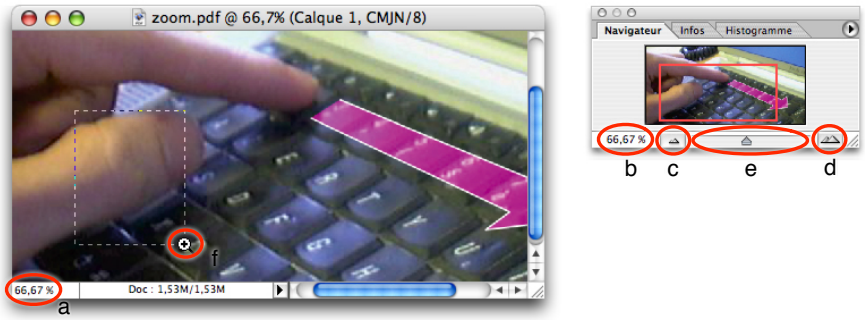


FIG. 4.16 – Différents moyens de zoomer dans Adobe Photoshop

– l'utilisation de l'outil loupe qui permet grâce à ses deux variantes d'incrémenter et de décrémenter de même le facteur de zoom lorsque l'on clique dans la zone de travail ; et enfin
– l'utilisation du même outil loupe pour sélectionner une région de la zone de travail — à l'aide d'un cliquer-tirer — spécifiant soit la partie du document qui doit occuper tout l'espace de travail (et donc sur laquelle on veut zoomer), soit la partie de l'espace de travail qui doit être occupée par la partie visible du document (qui sera ainsi réduite, permettant d'obtenir une vue plus large sur le document) (Figure 4.16f).

Si l'on analyse ces techniques suivant les critères d'indirection, d'intégration et de compatibilité présentés à la Section 2.1.2, on peut remarquer que leur multiplication permet d'obtenir divers compromis. La première technique, la saisie d'une échelle, est celle qui offre la compatibilité avec la tâche la plus faible. Elle ne propose aucun retour intermédiaire et présuppose donc que l'utilisateur sache quel est le bon facteur de zoom. La seconde technique, l'utilisation de commandes d'un menu pour incrémenter l'échelle, impose une indirection spatiale qui la rend peu intéressante. Les trois autres techniques, basées sur l'incrémentation de l'échelle, ne présentent pas cet inconvénient mais partagent le même degré de compatibilité très faible. Le facteur de zoom peut prendre ses valeurs dans un intervalle continu possédant une dimension. Le faire varier à l'aide de commandes discrètes permettant simplement de l'augmenter et de le réduire par pas fixés n'est donc pas directement adapté à la tâche. Les deux techniques restantes, l'usage du potentiomètre et celui de la sélection, sont plus intéressantes et ce pour plusieurs raisons. Elles offrent un retour immédiat permettant d'ajuster directement l'échelle pour obtenir la vue désirée sur le document. Elles ont par ailleurs un degré d'intégration meilleur. Dans le premier cas, une dimension du mouvement du curseur est certes négligée, mais la compatibilité est totale avec l'autre dimension. Cette technique présente quand même l'indirection d'être associée à la vue radar et non au document lui-même. Le *control menu* utilisé pour zoomer offre la même compatibilité mais supprime l'indirection spatiale en utilisant opportunément un menu contextuel. Dans le dernier cas, la tâche est habilement transformée pour désigner une zone de la vue, tâche qui peut être réalisée naturellement à l'aide du curseur.

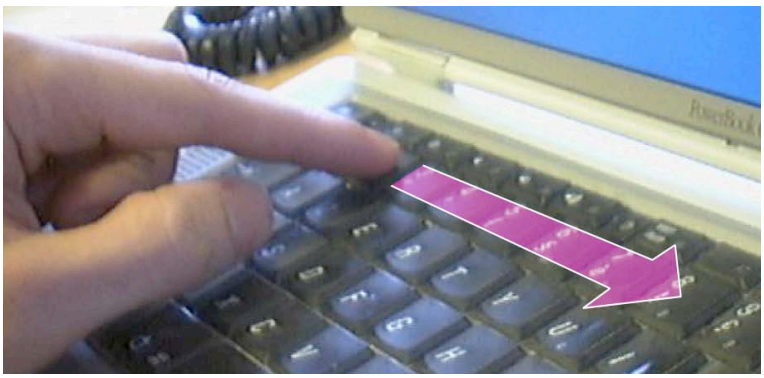


FIG. 4.17 – Le zoom continu au clavier

Principe du zoom continu au clavier

La technique que nous proposons permet de réduire l'indirection spatiale et temporelle sans requérir l'activation d'un mode particulier : elle utilise certaines touches du clavier qui sont en général sous employées dans les applications de création graphique. Elle augmente le degré d'intégration de la technique en utilisant une rangée complète de touches là où les autres techniques utilisant le clavier n'en utilisent que deux. Ainsi, la rangée de touches est utilisée comme un axe sur lequel s'effectue le contrôle du facteur de zoom.

La Figure 4.17 illustre l'utilisation de la technique qui est particulièrement adaptée aux ordinateurs portables, lesquels disposent de touches fines qui peuvent être parcourues en glissant le doigt dessus. En parcourant la rangée de gauche à droite, le document est zoomé alors qu'un parcours dans le sens inverse réduit la taille de la partie affichée. La rangée de touches devient ainsi un capteur de déplacement le long d'un axe à une dimension qui offre un contrôle isotonique sur le facteur de zoom.

Réalisation

Cette technique d'interaction utilise le clavier d'une manière inhabituelle. En effet, ce n'est pas l'appui d'une touche qui est porteur d'information, mais la succession des touches enfoncées qui détermine le sens du mouvement. Une machine à états est bien adaptée pour extraire d'une suite d'événements discrète une information basée sur la temporalité de ces événements. De plus, le modèle hiérarchique des HSM permet de structurer le code en séparant les divers problèmes.

La Figure 4.18 donne le code réalisant le zoom au clavier. La machine à états `KeyboardZoomer` comporte deux sous-états :

– `Idle` (ligne 5), dans lequel le début de l'interaction est attendu ; et
– `Zoom` (ligne 9), dans lequel l'interaction proprement dite a lieu.

Le début de l'interaction est déclenché par l'appui sur une touche du clavier qui déclenche la transition de l'état `Idle` vers l'état `Zoom`. Cette transition est spécifiée par l'état `Idle` ligne 6. Pour que cette transition et les suivantes réagissant aux touches du clavier ne soient effectivement franchies que si les touches en question font partie de la rangée numérique située en haut du cla-

```
01  hsm KeyboardZoomer {
02
03     - keys [(event.data.aInt < '1') || (event.data.aInt > '9')]
04
05     hsm Idle {
06        - keys  > Zoom::Continuous(key = event.data.aInt)
07     }
08
09     hsm Zoom {
10        var hsm::SVGLWindow *window;
11        var hsm::Point &point;
12
13        var hsm::Zoom *zoom = 0;
14
15        ! ((zoom = window->pick< hsm::Zoom >(point)) != 0) : Idle
16        ! (zoom->begin()) : Idle
17
18        leave { zoom->end(); }
19
20        hsm Continuous {
21           var hsm::Zoom *zoom;
22           var int key = 0;
23
24           - 200 > Idle
25           - keys {
26              zoom->zoom((event.data.aInt - key) * 10);
27           } > Continuous(key = event.data.aInt)
28        }
29     }
30  }
```

FIG. 4.18 – Le code réalisant le zoom au clavier

vier (visible sur la Figure 4.17), une transition supplémentaire a été ajoutée au plus haut niveau de la machine (ligne 3). Cette transition n'exécute aucun code et ne fait pas changer d'état. Son rôle est uniquement d'absorber les événements ne provenant pas des touches numériques. Grâce à sa garde qui exclut les événements provenant des touches numériques, seuls ceux-ci atteignent les transitions des sous-états de la machine, simplifiant l'expression de ces dernières.

C'est donc l'appui d'une touche numérique qui permet de franchir la transition de l'état `Idle` (ligne 6), et de commencer l'interaction. La cible de cette transition est `Continuous` contenu dans l'état `Zoom` et, au moment de son franchissement, la transition lui communique quelle touche a été enfoncée en récupérant l'information des données contenues dans l'événement.

Avant la définition de son sous-état, l'état `Zoom` déclare un ensemble de variables utiles (lignes 10 à 13) dont en particulier la fenêtre et la position qui lui serviront à réaliser le *picking*, ainsi que l'objet sur lequel est réalisé le zoom.

Suivent deux préconditions qui doivent être vérifiées pour entrer dans cet état :

- ligne 15, un objet zoomable doit être présent sous le curseur *(picking)* ; et
- ligne 16, cet objet doit accepter l'interaction (protocole élémentaire d'interaction *begin/end*).

Si l'une de ces conditions n'est pas remplie, la machine retourne dans l'état `Idle`. Sinon, l'interaction commence et ne se terminera que lorsque l'état `Zoom` sera quitté, et que l'action associée à ce départ sera effectuée. Celle-ci est définie ligne 18 par l'action `leave` qui notifie l'objet de la fin de l'interaction.

La présence dans l'état `Zoom` veut dire qu'un objet compatible a été trouvé et que l'interaction a commencé. C'est alors la succession des touches qui l'entretient, et la fin de l'interaction est déclenchée par la dernière touche enfoncée. Cette notion de "dernière" n'a de sens qu'en définissant un délai après lequel aucune nouvelle touche n'est attendue, si ce n'est pour commencer une nouvelle interaction. C'est donc une transition déclenchée par l'expiration d'un délai qui interrompt l'interaction et ramène la machine dans l'état `Idle` après que 200 millisecondes aient passé sans qu'une touche ait été enfoncée (ligne 24). Ce délai est compté à partir de l'entrée dans l'état qui contient la transition. Pour ne pas avoir à retrouver un nouvel objet compatible avec l'interaction à chaque fois qu'une nouvelle touche est enfoncée, elle est encapsulée dans un sous-état de `Zoom`.

C'est donc l'état `Continuous` (lignes 20 et suivantes) qui est quitté et réentré durant l'interaction grâce à la transition activée par les événements venant des touches (lignes 25 à 27). À chaque appui de touche, l'objet de l'interaction est grossi ou réduit d'un facteur proportionnel à l'écart entre les touches successives. Pendant ce temps, la machine reste dans l'état `Zoom`. Celui-ci n'est finalement quitté que lorsque la transition déclenchée par l'expiration du délai quitte l'état `Continuous`, puis en remontant, l'état `Zoom`, mettant ainsi fin à l'interaction et remettant enfin la machine dans l'état `Idle` où elle attend le début d'une nouvelle interaction.

4.4 Discussion

Revenons à présent sur les objectifs que nous nous sommes fixé. Ces objectifs sont de permettre conjointement de réutiliser facilement l'existant, et de créer tout aussi facilement de nouvelles techniques d'interaction.

4.4.1 Réutilisabilité

Le principal obstacle à la réutilisation de techniques d'interaction est lié au style de programmation impératif habituellement employé pour les programmer. En effet, une technique d'interaction consiste souvent à faire évoluer l'état de l'application en accord avec les modifications qui ont lieu sur les périphériques d'entrée. Cet état est généralement stocké de manière diffuse au sein de structures de données dont les liens et les invariants sont souvent implicites et rarement exprimés. Dans ces conditions, où la logique est distribuée au sein de fonctions de rappel, il est difficile d'isoler une technique d'interaction, pour pouvoir la réutiliser.

Ce que permet HsmTk, c'est de rendre explicite l'état en le mettant directement au cœur d'une structure de contrôle, les machines à états hiérarchiques. Ainsi, les interactions et les comportements interactifs deviennent des objets à part entière du langage. Leurs liens avec le reste de l'application étant complètement explicites, ils peuvent en être détachés et réutilisés sans peine dans d'autres contextes. L'aspect hiérarchique encourage par ailleurs cette réutilisation puisqu'il permet — comme on l'a vu avec l'exemple du déplacement contraint à la Section 4.3.1, page 82 — d'encapsuler une technique préexistante pour la raffiner ou l'adapter à de nouveaux usages plus particuliers.

Cet aspect hiérarchique permet aussi de limiter l'explosion combinatoire du nombre de transitions possibles avec l'augmentation du nombre d'états d'une machine. Il permet de structurer ces états, de hiérarchiser leurs relations, et ainsi de factoriser certaines de leurs transitions. C'est aussi une forme de réutilisation puisque ces simples transitions sont alors partagées entre plusieurs états, ce qui permet de réduire la duplication de code.

4.4.2 Expressivité

Le formalisme choisi est le fruit d'un compromis entre sa simplicité et son expressivité. Sa syntaxe textuelle permet son intégration aisée dans un processus de développement classique, et ne nécessite pas d'environnement particulier pour être éditée. Les fichiers qui définissent les HSM sont traduits grâce à un préprocesseur mis au point et distribué avec la boîte à outils HsmTk, puis compilés comme les unités de compilation classiques d'un programme. Le code ainsi généré pour le zoom au clavier est donné en exemple en Annexe A. La limitation actuelle de ce processus se trouve lors de la mise au point puisque les erreurs générées à la compilation ont un lien qui peut être difficile à reconstituer avec le fichier source original. Des mécanismes d'annotation du code généré permettant de transmettre des informations aux compilateurs peuvent résoudre ce problème, mais ils ne sont pas encore standards d'un compilateur à l'autre.

Outre la facilité de l'intégrer dans un processus de développement classique, l'utilisation d'un formalisme textuel comporte d'autres avantages. En particulier, le texte possède un ordre strict qui est celui de l'apparition des définitions dans le texte. Le sous-état initial d'une machine est ainsi par défaut toujours le premier sous-état qu'elle définit. Cet ordre permet aussi d'ordonner les transitions et fournit ainsi un moyen simple d'avoir une machine complètement déterministe. Les transitions sont en effet toujours examinées dans l'ordre de leurs définitions et c'est la première transition franchissable qui est franchie.

Il serait certes possible de donner aux machines à états hiérarchiques une syntaxe visuelle, en adaptant par exemple celle des *statecharts* mais le bénéfice d'une telle notation n'est pas évident, car il faut de toute façon associer du code à cette représentation. Par ailleurs, les programmeurs sont habitués à ce type de syntaxe textuelle à base de blocs inclus les uns dans les autres. Le formalisme utilisé ne fait qu'introduire une dizaine de mots clés et de constructions syntaxiques particulières, qui sont aisément assimilées. Ces constructions, tout en restant simples, permettent cependant une concision comme l'illustrent les exemples présentés. L'interaction de déplacement ou le comportement du bouton nécessitent chacun une quarantaine de lignes, et les techniques plus complexes ou originales guère plus. Ces techniques n'ont pas nécessité chacune plus d'une heure de réalisation et de mise au point.

Le formalisme des machines à états hiérarchiques, utilisé de concert avec les abstractions mises à disposition par la boîte à outils HsmTlk, nous permet donc de réaliser l'état de l'art des techniques d'interaction classiques ainsi que celui des techniques avancées, et ce facilement, rapidement, et de manière concise. Il est adapté à des reconnaissances simples de geste comme celle utilisée pour le multiplexage de la translation et du zoom à l'aide du menu circulaire. Il nous a permis aussi de reproduire facilement des interacteurs basés

sur le franchissement. Pour des reconnaissances de gestes plus élaborées, l'intégration de techniques comme celle proposée par Rubine [1991] est possible, étant donné qu'elle repose sur un calcul incrémental de caractéristiques d'un geste, qui peut être effectué dans une transition activée par les mouvements. Cependant, nous n'avons pas réalisé de techniques de ce type.

Par ailleurs, les machines à états hiérarchiques nous ont permis, comme le montre l'exemple du zoom réalisé continûment grâce au clavier, de mettre au point de nouvelles techniques d'interaction avancées en facilitant le prototypage rapide d'idées et leur raffinement progressif vers des versions complètement utilisables. Ces capacités sont illustrées sur des exemples de plus grande ampleur dans la partie suivante de ce travail.

Deuxième partie

Exemples de réalisations utilisant la boîte à outils HsmTk

Chapitre 5

Le projet INDIGO

Le projet INDIGO a pour but de proposer une architecture permettant de réaliser des applications graphiques interactives distribuées. La boîte à outils HsmTk a été utilisée dans ce contexte et a montré son adaptation à la conception de telles applications.

Le projet INDIGO (pour *INteractive DIstributed Graphic Objects* ou objets graphiques interactifs distribués) propose une architecture pour réaliser des applications graphiques interactives distribuées [Blanch *et al.*, 2005]. Celle-ci sépare le noyau fonctionnel de l'application de son interface au sein de processus distincts et éventuellement distants : les serveurs d'objets applicatifs et les serveurs d'interaction et de rendu. Dans cette architecture, le protocole de communication entre les deux types de serveurs est de haut niveau, celui de la sémantique des actions sur les objets applicatifs.

Pour réaliser les serveurs d'interaction et de rendu dans une réalisation prototype, nous avons utilisé la boîte à outils HsmTk. Elle nous a permis de développer plusieurs applications qui sont présentées ici, après avoir donné une vue d'ensemble de l'architecture INDIGO. Nous discuterons ensuite des apports de HsmTk pour ce projet.

5.1 L'architecture d'INDIGO

La séparation entre le noyau fonctionnel et l'interface d'une application est mise en avant depuis longtemps maintenant par les travaux en architecture logicielle des systèmes interactifs. C'est ce précepte qui a permis en particulier l'apparition des boîtes à outils, qui permettent de faciliter le développement

de l'interface indépendamment du noyau fonctionnel. Cette séparation a conduit, comme on l'a vu, à une standardisation a minima de l'ensemble des interacteurs disponibles.

Avec l'apparition de nouvelles plates-formes — des téléphones mobiles aux murs interactifs, en passant par les assistants numériques (PDA), les tablettes numériques, ou les tables interactives — les dispositifs d'entrée et d'affichage sont devenus d'une grande diversité, et les capacités de calcul très variables. Le but d'INDIGO est de permettre de prendre en compte la diversité de ces plates-formes pour pouvoir proposer des interactions adaptées, éventuellement post-WIMP et collecticielles, en tirant partie d'une séparation adaptée du noyau fonctionnel et de l'interface. Pour cela, l'architecture logicielle INDIGO se base sur quatre principes :

- une *architecture répartie* formée de serveurs spécialisés dans la gestion d'objets de l'application d'une part, et dans l'interaction et le rendu graphique d'autre part ;
- la *transformation des objets* de l'application en un graphe de scène dont le rendu est contrôlé en fonction de la plate-forme ;
- l'*interprétation des actions* de l'utilisateur en commandes de haut niveau sur les objets du domaine ; et
- le *contrôle de la cohérence* permettant à plusieurs serveurs d'interaction et de rendu de représenter les mêmes objets.

5.1.1 Architectures existantes

Une architecture répartie entre plusieurs processus est très répandue pour les applications graphiques notamment depuis l'apparition du serveur graphique X Window sur les systèmes à base Unix. Celui-ci a consacré la terminologie suivante : le *serveur* est le processus local qui gère le rendu et l'interaction de l'utilisateur, et le *client* est l'application proprement dite, éventuellement située sur une machine distante. Un bon moyen pour caractériser ces architectures est d'observer le niveau d'abstraction du protocole utilisé par le noyau fonctionnel et l'interface pour communiquer.

Au niveau le plus bas, on trouve des systèmes pour lesquels le dessin complet de l'interface, ainsi que la gestion de l'interaction sont assurés du côté du client, le protocole se contentant de transmettre l'image, au niveau des pixels, vers le serveur. Ce niveau d'abstraction est celui que propose VNC [Richardson *et al.*, 1998] qui transmet, compressées, des copies successives du contenu d'une fenêtre au serveur. Celui-ci envoie dans l'autre sens les événements de bas niveau provenant de la souris et du clavier. Ce modèle a le mérite d'être très simple, il revient à filmer la fenêtre d'une application sur le système où elle tourne, à projeter ce film à distance, et dans l'autre sens à simuler clavier et souris d'après les actions distantes de l'utilisateur. Il permet d'utiliser des applications complètement monolithiques, puisque celles-ci n'ont pas besoin d'être conçues pour une quelconque architecture répartie. Il est cependant difficile pour nous de considérer ce point comme un avantage. D'autres limitations du modèle sont aussi évidentes. Tout d'abord, la bande passante nécessaire pour transmettre les images est importante, surtout pour des applications hautement interactives pour lesquelles la cohérence temporelle de l'affichage est faible. Pour une interface zoomable par exemple, deux images successives ont peu de chance de présenter des similitudes locales au

niveau des pixels, ce qui est nécessaire pour obtenir des bons taux de compression sans perte. Si ce modèle est adapté à des applications très peu interactives, il ne l'est pas pour nous. Par ailleurs, le modèle des dispositifs d'entrée est réduit au plus petit dénominateur commun : un clavier et une souris. Il est impossible de l'étendre, et dès lors, l'application ne peut s'adapter à la configuration du serveur, qui lui est là encore complètement inconnu. Pire encore, la boucle action/perception fait constamment l'aller et retour entre le serveur et le client, ce qui peut prendre un temps important, incompatible avec l'interaction, sur des réseaux encombrés.

Le système X Window [Nye, 1988] propose des primitives graphiques plus élaborées (segments, rectangles, arcs d'ellipse, couleur, texte) que les pixels pour que les clients mettent à jour leur affichage, ce qui permet de meilleures performances. Il conserve par contre le même type de description des périphériques d'entrée que VNC, bien qu'il ait été rendu extensible. Des bibliothèques utilisées du côté du client permettent d'utiliser des abstractions plus élevées que les primitives géométriques, comme celles des interacteurs standards. Cependant, l'interaction reste encore complètement gérée du côté de l'application.

Display PostScript d'Adobe [1993] propose un langage graphique encore plus expressif puisqu'il permet au client de télécharger des programmes PostScript dans le serveur. Mais là encore, l'interaction n'est pas réalisée du côté du serveur. NeWS [Gosling *et al.*, 1989] utilise ce même modèle graphique, mais étend le langage PostScript pour supporter l'interaction. Avec ce mécanisme, l'interaction est bien gérée du côté du serveur, mais l'architecture incite à y placer aussi la logique de l'application. Le client se contente alors souvent de télécharger un programme complet dans le serveur, ce qui revient à nouveau aux applications monolithiques.

D'autres systèmes comme Fresco [Linton et Price, 1993] ont choisi de partager des objets (au sens des langages à objets) entre client et serveur grâce à un logiciel d'intermédiation *(middleware).* Ces systèmes ont aujourd'hui été abandonnés, sans doute en partie à cause de la complexité du protocole de communication qui, s'il permettait un haut niveau d'abstraction, pénalisait les performances de l'interaction.

Dans le domaine du Web, et notamment avec la possibilité de mieux en mieux supportée de manipuler l'arbre du document HTML et de faire des requêtes HTTP depuis le langage Javascript sans recharger la page, des applications plus élaborées que le simple enchaînement de formulaires voient le jour[1]. Pour celles-ci, les informations transitant entre le client et le serveur sont définies par les applications, et peuvent être de haut niveau. Cependant, les modèles graphique (celui des rectangles emboîtés), et surtout d'interaction (le clic sur les hyperliens) proposés par cette plate-forme HTML/Javascript sont très succincts[2].

[1] Les applications de ce type ont été popularisées récemment par le succès de l'interface Web au service de courrier électronique proposé par Google. L'acronyme AJAX *(Asynchronous JavaScript and XML)* s'est ensuite imposé pour désigner les technologies utilisée par ces applications (`http://en.wikipedia.org/wiki/AJAX`).

[2] Il faut cependant noter que le support croissant de SVG par les navigateurs laisse entrevoir des pistes prometteuses pour ce type d'applications.

5.1.2 INDIGO

Si l'on se réfère au modèle de l'Arch [1992] présenté à la Section 2.1.1, page 13, les architectures présentées ci-dessus séparent leurs éléments de part et d'autre du réseau à différents niveaux. Les premiers les placent tous du côté du client, alors que NeWS permet de les placer tous du côté du serveur. Nous avons opté, avec INDIGO, pour une solution plus nuancée qui repose sur deux types de serveurs :

- les *serveurs d'objets* (SERVO) qui réalisent le noyau fonctionnel de l'application, en utilisant un vocabulaire propre aux objets manipulés et à leur domaine d'application ; et
- les *serveurs d'interaction et de rendu* (SERVIR) qui mettent en œuvre les représentations et les techniques d'interaction associées, en fonction des capacités de leur plate-forme.

La gestion des objets du domaine, jusqu'au contrôle du dialogue inclus, fait donc partie du SERVO, alors que la présentation et l'interaction sont confiées au SERVIR.

Graphe d'objets conceptuels et perceptuels

La charnière qui permet d'articuler SERVO et SERVIR est une représentation des objets de l'application. Dans la terminologie de l'Arch, il s'agit des objets de présentation. Pour INDIGO, c'est le *modèle conceptuel* de l'application. Il s'agit en effet de la collection des objets ayant un sens pour les utilisateurs et des méthodes qui leur sont applicables. Ils fournissent ensemble le modèle que se fait l'utilisateur de l'application. Ils sont stockés au sein d'un graphe, le *graphe d'objets conceptuels* (COG) réalisé par un arbre XML. Chaque SERVO gère un tel COG et maintient sa cohérence en réponse aux demandes de mise à jour qu'il reçoit des SERVIR. Les SERVIR disposent d'une copie synchronisée du COG (le SCOG) et sont chargés de la transformer dans une forme perceptible et manipulable par les utilisateurs.

Cette dernière est appelée *graphe d'objets perceptuels* (POG), et utilise le format SVG comme support concret dans le cas particulier des SERVIR présentés ici. Cependant, chaque SERVIR peut utiliser les mécanismes qu'il souhaite pour rendre perceptibles et manipulables les objets du POG. Les communications entre SERVIR et SERVO se font en effet au niveau d'abstraction du COG, ce qui permet de rester indépendant de la forme concrète des objets d'interaction et de limiter le volume d'information transitant entre les serveurs, puisque toute l'interaction est déléguée au SERVIR. La Figure 5.1 illustre le fonctionnement général d'INDIGO.

Concrétisation

La transformation du SCOG en POG s'appelle la *concrétisation* et doit vérifier les propriétés suivantes, du moins pour l'interaction graphique :

- la *réversibilité*, c'est-à-dire que le lien bidirectionnel entre un objet du POG et l'objet dont il provient dans le COG doit être maintenu ; et
- l'*incrémentalité*, c'est-à-dire qu'une modification du COG (insertion, suppression d'un élément, modification d'un attribut) ne doit pas nécessiter le recalcul complet du POG.

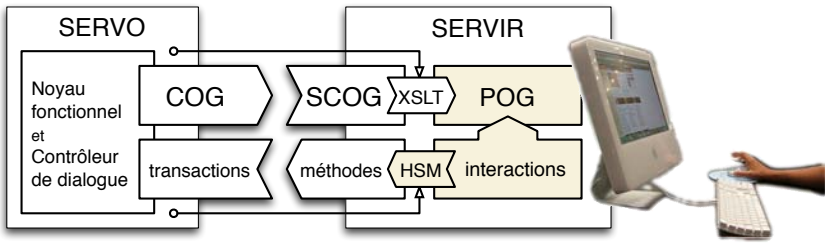


FIG. 5.1 – Fonctionnement général d'INDIGO

(Les éléments colorés sont directement fournis par HsmTk)

Ces propriétés sont nécessaires pour permettre l'interaction : l'incrémentalité pour des questions de performance ; la réversibilité pour permettre la manipulation directe.

La concrétisation a lieu dans le SERVIR et dans le cas présenté ici, elle utilise un programme XSLT[3] qui produit un document SVG annoté par les comportements des objets graphiques à l'aide des mécanismes proposés par HsmTk. Pour produire des transformations qui peuvent être incrémentales, nous avons posé des restrictions au langage XSLT. En particulier, nous imposons que les éléments du COG soient transformés en groupes SVG, et que les images des descendants d'un élément soient incluses dans les sous-arbres SVG images de l'élément. Ce type de restrictions nous a permis de réaliser simplement le moteur de transformation XSLT incrémental nécessaire à notre prototype, mais des travaux récents montrent qu'elles ne sont pas nécessaires pour y parvenir [Onizuka *et al.*, 2005].

La transformation produit également des annotations qui incluent les informations nécessaires pour établir le lien bidirectionnel entre les objets conceptuels et leurs représentations. Pour cela, à chaque groupe SVG issu de la transformation d'un objet conceptuel est associé un identifiant. Le moteur de concrétisation maintient une table qui permet de retrouver toutes les images d'un objet particulier du COG et ainsi de gérer incrémentalement les notifications du SERVO (changement de valeur d'un attribut, ajout ou suppression d'un élément) en remplaçant, insérant ou supprimant des éléments de l'arbre SVG. À l'inverse, chaque groupe SVG contient un attribut identifiant l'objet qu'il représente sous la forme d'un chemin XPATH dans le COG. Cet attribut permet au comportement associé au groupe de traduire les manipulations interactives en appels de méthodes de l'objet conceptuel. Ces appels de méthode sont relayés au SERVO de manière transactionnée. Les modifications éventuelles des objets conceptuels qui en résultent sont notifiées à l'ensemble des SERVIR connectés, provoquant en bout de chaîne la mise à jour des parties du POG correspondant aux objets affectés.

La Figure 5.2 montre en haut un graphe conceptuel simplifié représentant une arborescence de fichiers, tel qu'il est publié par un SERVO gestionnaire de fichiers, et la table des identifiants donnant les images de chaque élément dans le POG. Cette table est calculée lors de la génération du POG montré en bas. Celui-ci a la structure d'arbre que nous connaissons enrichie pour chaque groupe correspondant à un élément du COG des annotations per-

[3] XLST est un langage de transformation d'arbres XML (`http://www.w3.org/TR/xslt`).

graphe conceptuel

```
<Folder type='Folder'>
   <Name type='string'>~/test</Name>
   <Entry type='File'>
      <Name type='string'>README</Name>
   </Entry>
   <Entry type='File'>
      <Name type='string'>INSTALL</Name>
   </Entry>
   ...
```

table des identifiants

```
d0e07
d0e12
d0e15
d0e17

d0e24
d0e26
```

graphe perceptuel

```
<!-- arbre -->
<g hsm:behaviour='tree' indigo:path='/' id='d0e07'>
   <rect width='1000' height='16' style='fill:url(#bgd)'/>
   <use xlink:href='#closed'/>
   <text indigo:path='/Name' id='d0e12' x='22' y='12.5'>~/test</text>

   <!-- contenu -->
   <g transform='translate(16,16)'>
      <g hsm:behaviour='node' indigo:path='/Entry[0]' id='d0e15'>
         <rect width='1000' height='16' style='fill:white; opacity:0'/>
         <text indigo:path='/Entry[0]/Name' id='d0e17'
               x='22' y='12'>README</text>
      </g>
      <g hsm:behaviour='node' indigo:path='/Entry[1]' id='d0e24'>
         <rect width='1000' height='16' style='fill:white; opacity:0'/>
         <text indigo:path='/Entry[1]/Name' id='d0e26'
               x='22' y='12'>INSTALL</text>
      </g>
      ...
```

FIG. 5.2 – COG et POG représentant un système de fichiers

mettant de remonter à celui-ci (attribut `indigo:path`) et d'être retrouvé à partir de la table des identifiants (attribut `id`). L'arbre est par ailleurs annoté pour spécifier les comportements interactifs des divers éléments (attributs `hsm:behaviour`). Le programme XSLT permettant de transformer ce COG en POG SVG est donné en Annexe B.

Serveurs d'objets

Les serveurs d'objets ont la charge de maintenir la cohérence des données. Ils peuvent pour cela mettre en œuvre les moyens de leur choix. Les seules contraintes qui leur sont fixées sont le standard de publication pour les services qu'ils proposent et le protocole de communication qu'ils utilisent. Ceux-ci utilisent des standards du Web afin de permettre au protocole INDIGO de fonctionner sur le réseau internet, par delà les passerelles coupe-feu. Ces standards sont ceux proposés par le W3C :

- WSDL (*Web Services Description Language* ou langage de description des services Web[4]) pour publier le modèle de données des objets conceptuels et les fonctions du SERVO ; et
- SOAP (*Simple Object Access Protocol* ou protocole simple d'accès aux objets[5]) pour permettre aux SERVIR d'invoquer ces fonctions.

[4] La spécification de WSDL se trouve à l'adresse : `http://www.w3.org/TR/wsdl/`.

[5] La spécification de SOAP se trouve à l'adresse : `http://www.w3.org/TR/soap/`.

Par cet intermédiaire, un SERVO, outre son modèle conceptuel, doit exposer trois fonctions :

– *get* qui retourne le COG géré par le SERVO ;
– *post* qui demande l'exécution d'une requête au SERVO ; et
– *listen* qui retourne les modifications effectuées sur le COG.

La fonction *get* permet de ne charger qu'un sous-ensemble du COG, limité en profondeur ou en largeur grâce à un filtre passé en paramètre. Cela autorise le chargement paresseux de la structure de donnée lorsqu'elle est volumineuse. La fonction *post* permet de demander la création ou la suppression d'un élément du COG, ou la modification d'un de ses attributs. Elle permet aussi d'appeler une méthode sur un de ses objets. Elle ne fait que demander ces modifications, elle ne permet pas de savoir si l'opération a réussi ou échoué : c'est une demande de transaction.

C'est la fonction *listen* qui permet de savoir si une telle transaction s'est bien déroulée. Elle rapporte en effet les modifications qu'a subies le COG depuis la version dont on lui passe le numéro en paramètre. Elle permet ainsi de maintenir la version répliquée du COG (le SCOG) gérée par le SERVIR et synchronisée avec celle du SERVO. Elle permet aussi de mettre en œuvre avec SOAP un mécanisme de notification à l'initiative du SERVO. En effet, seul le client SOAP (ici le SERVIR) peut appeler des fonctions du serveur ce qui interdit en théorie au SERVO d'envoyer des notifications. La notification fonctionne donc en *pull-wait-push* : c'est le SERVIR qui contacte le SERVO, mais si aucune modification n'est à signaler, ce dernier ne répond pas et le SERVIR est alors en attente passive sur la connexion, ce qui ne consomme pas de ressource. Dès qu'une modification du COG intervient, le SERVO répond au SERVIR en attente qui est alors notifié instantanément des changements. Au-delà d'un certain délai (1 à 5 secondes) sans modification, le SERVO répond tout de même, signalant qu'il ne s'est rien passé, pour éviter que les connexions ne soient considérées comme perdues par les couches plus basses du protocole réseau. Il suffit alors au SERVIR d'appeler à nouveau la fonction *listen* pour rester à l'écoute des futures modifications.

Serveurs d'interaction et de rendu

Plusieurs types de SERVIR ont été réalisés. Le premier a consisté en un prototypage à l'aide d'un serveur Web utilisant un document HTML comme graphe perceptuel. Le modèle graphique des boîtes imbriquées de HTML nous a permis de réaliser un explorateur de fichiers arborescent montré sur la Figure 5.3. Si ce modèle graphique est satisfaisant pour le prototypage, il est limité pour réaliser des interfaces réellement graphiques. Par ailleurs, le modèle d'interaction proposé par HTML et le langage de script Javascript qui lui est associé est très limité. Ce prototype montre néanmoins que des SERVIR sont réalisables sur cette plate-forme minimale HTML/Javascript, surtout à présent qu'il devient possible de faire des requêtes HTTP (et donc SOAP), et des transformations XSLT depuis Javascript dans les navigateurs.

Un SERVIR plus élaboré a été mis au point en utilisant HsmTk. Celui-ci utilise directement les capacités de la boîte à outils pour afficher le document SVG et le manipuler, ainsi que pour gérer l'interaction. Les techniques d'interaction sont ainsi disponibles en fonction des capacités de la plate-forme. Les comportements interactifs sont quant à eux instanciés dynamiquement

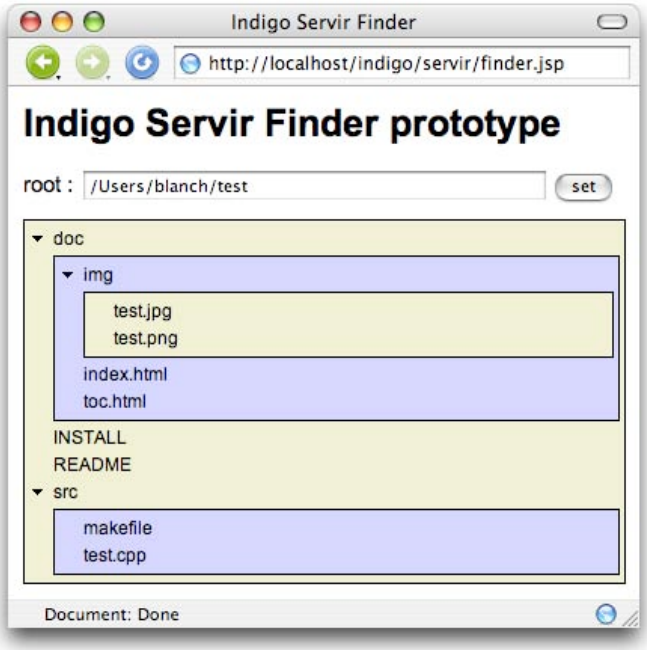


FIG. 5.3 – Prototype Web du SERVIR pour la gestion de fichiers

lors du chargement du SVG. Le support à la modularisation fourni par les bibliothèques dynamiques est ici étendu : si un comportement est inconnu du SERVIR, celui-ci le cherche d'abord dans un entrepôt local, comme c'est le cas par défaut avec HsmTk, puis, s'il ne le trouve pas, il le réclame au SERVO qui lui retourne sous la forme d'une archive de sources. Celles-ci sont alors compilées en un comportement qui est ajouté à l'entrepôt local du SERVIR. Ce mécanisme permet de fournir une bibliothèque de comportements réutilisables, et de l'étendre facilement sans qu'il ne soit besoin de modifier le SERVIR. Celui-ci est donc un serveur générique qui peut être enrichi de nouvelles techniques d'interaction suivant les besoins spécifiques des applications réalisées par les SERVO.

5.2 Applications

En utilisant ce SERVIR graphique basé sur HsmTk nous avons réalisé quelques applications qui permettent de faire la démonstration de la validité de l'architecture INDIGO.

5.2.1 Explorateur de fichiers

La première application a consisté en un explorateur de système de fichiers arborescent. Du côté du SERVIR, l'essentiel du travail pour la création de cette application a consisté en l'écriture de la transformation XSLT donnée en Annexe B. En effet, le comportement associé à l'arbre était préexistant, et il a suffit que le programme XSLT annote correctement le document SVG produit

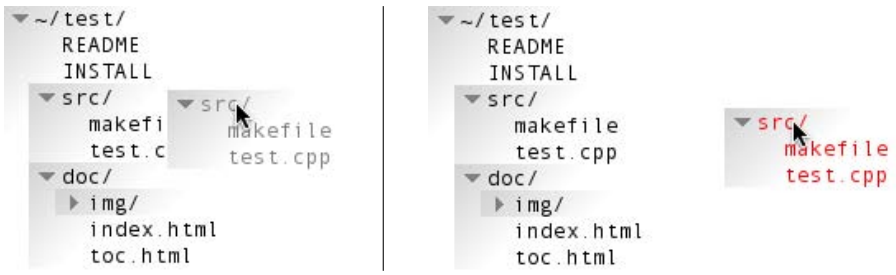


FIG. 5.4 – Interaction de suppression d'un fichier

pour qu'il ait le comportement attendu d'un arbre. Une interaction a ensuite été ajoutée pour permettre de supprimer des fichiers (cette fonction étant exposée par le SERVO). Elle a consisté à ajouter un comportement aux éléments de l'arbre. Celui-ci est une spécialisation du protocole *translate* qui, au lieu de déplacer l'élément visé, en crée une image qu'il déplace et dont il altère le rendu suivant la distance à laquelle elle se trouve de l'original (Figure 5.4). Tant que cette image reste proche (à gauche), le texte est grisé et si l'image est relâchée, elle disparaît simplement. Si elle est plus éloignée de sa position originale (à droite), le texte devient rouge et si elle est relâchée à cet endroit, une requête de suppression de l'objet est envoyée au SERVO. Cette requête est construite en récupérant l'identifiant du groupe SVG, ce qui permet, via la table maintenue par le SERVIR, de retrouver l'élément du COG correspondant. Si cette transaction réussit, le fichier est supprimé du COG par le SERVO, cette modification étant alors diffusée au SERVIR, et, grâce à la mise à jour incrémentale du SVG, les éléments correspondants sont supprimés de l'arbre.

5.2.2 Jeu interactif

Une autre application développée est un jeu de société dans lequel deux joueurs laissent tomber tour à tour des jetons jaunes pour l'un et rouges pour l'autre au sein d'une grille verticale comportant sept colonnes pouvant accueillir chacune six pions (Figure 5.5). Le gagnant est le premier joueur qui arrive à aligner quatre de ses pions suivant une ligne, une colonne ou une diagonale de la grille. Le SERVO de ce jeu se contente de maintenir l'état du tableau de pions et de les ajouter, quand c'est possible, dans la colonne qui a été sélectionnée. Les colonnes ont pour cela un comportement dérivé de celui du bouton qui invoque la fonction adéquate du SERVO, en lui donnant comme paramètre l'indice qui permet de les distinguer. Le comportement du bouton est légèrement modifié pour que la colonne courante soit mise en valeur par un halo dont la couleur rappelle au joueur la couleur de ses pions. Les pions ajoutés au COG sont transformés en éléments SVG circulaires auxquels un comportement interactif est associé. Celui-ci déclenche simplement une animation lorsque le pion est créé pour le faire "tomber" dans sa colonne. Le bouton *reset* reprend quant à lui tel quel le comportement du bouton standard et invoque la transaction du SERVO qui permet de vider complètement le tableau. Le contrôle du dialogue (le respect des règles du jeu) est assuré par le SERVO qui refuse aux joueurs la possibilité de jouer deux fois de suite.

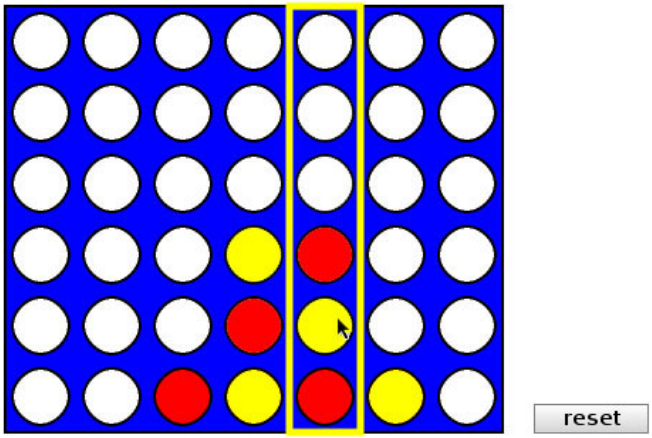


FIG. 5.5 – Grille du jeu

5.3 Discussion

La mise au point des applications présentées ci-dessus nous a permis d'utiliser notre boîte à outils dans un contexte réaliste. Outre la pertinence du modèle proposé par INDIGO pour bâtir des applications graphiques interactives distribuées, quelques enseignements concernant HsmTk peuvent être tirés de ces travaux. En premier lieu, ces développements ont montré que proposer l'accès à différents niveaux d'abstraction s'est révélé particulièrement adapté à la mise au point du SERVIR. Par ailleurs, le modèle d'interaction proposé par HsmTk a permis de mettre en place un cycle de développement particulièrement intéressant pour le prototypage d'applications interactives.

5.3.1 Adaptation des abstractions

À différents nivaux, les abstractions proposées par la boîte à outils se sont révélées être adaptées à nos objectifs.

Support de bas niveau

Au plus bas niveau il a fallu gérer le protocole de communication avec le SERVO. Celui-ci est complètement asynchrone puisque les résultats des opérations demandées au noyau fonctionnel sont communiqués par les notifications transmises par la fonction *listen*. Pour que l'interaction se déroule de manière fluide pendant que le SERVIR est en attente de ces notifications, un fil d'exécution particulier est dédié à cette attente. Le reste de la bibliothèque étant conçu pour supporter la concurrence, ce fil d'exécution peut manipuler le POG SVG en utilisant les primitives de synchronisation de HsmTk. Il peut ainsi effectuer les modifications provoquées par la mise à jour du POG sans interférer avec le reste du programme. Ainsi, l'interaction peut continuer pendant ce temps dans le fil d'exécution principal.

De même, ces fils d'exécution permettent de créer facilement des animations comme celle des pions qui tombent dans l'exemple du jeu en modi-

fiant leurs positions à intervalles réguliers. Là encore, ces modifications n'interfèrent pas avec le reste du programme.

Par ailleurs, le support au chargement dynamique de modules nous a permis d'ajouter un mécanisme permettant de télécharger des comportements et des techniques d'interaction dans le SERVIR depuis le SERVO. HsmTk n'offre cependant pas de garantie de sécurité qu'il serait nécessaire de mettre en œuvre pour exploiter ce mécanisme en situation réelle, mais elle permet, en tout cas dans notre contexte de recherche, d'explorer les perspectives qu'il ouvre.

Support à l'interaction

Nous avons pu constater que HsmTk a permis de faciliter la mise en place de techniques d'interaction intéressantes de plusieurs manières. En facilitant leur réutilisation tout d'abord, les comportements interactifs étant faciles à réutiliser puisqu'ils sont encapsulés dans des objets à part entière. Cette encapsulation est en grande partie rendue possible par le langage des machines à états hiérarchiques dont le support est fourni par la boîte à outils. Le découplage que celle-ci permet entre représentation graphique et comportement est aussi très favorable à cette réutilisation qui peut ainsi être faite dans des contextes différents. La réutilisation du comportement du bouton pour le jeu dans sa version originale et dans une version légèrement modifiée démontre cette capacité.

Nous avons vu aussi que ce découplage permet d'ajouter facilement de nouveaux comportements à des objets existants. L'ajout de l'interaction de suppression dans le gestionnaire de fichiers n'a en effet demandé aucune modification, ni du SERVO, ni du SERVIR, mais le simple ajout d'un attribut à certains éléments du POG grâce à la transformation XSLT, et la mise au point du comportement correspondant. Il faut noter qu'une fois cette technique reconnue intéressante, HsmTk permettrait d'en faire une technique d'interaction gérée par le SERVIR, et non un comportement associé par un SERVO à ses objets particuliers. Cette technique pourrait alors s'appliquer à tous les objets dont la description fournie par le modèle conceptuel stipule qu'ils peuvent être supprimés, et ce quel que soit le SERVO dont ils proviennent.

5.3.2 Processus de développement

Le découplage entre représentation et comportements nous a permis de mettre en place naturellement un processus de développement très itératif. Celui-ci permet de passer de manière continue de la maquette à un prototype fonctionnel, puis finalement à une application de haute qualité graphique.

Maquettage et prototypage

Le choix des techniques d'interaction, et de l'allure de l'interface passe en général par des esquisses et des croquis. Dans notre cas, ils ont été rapidement et naturellement réalisés en SVG. C'est ainsi que le plateau du jeu de société ou des versions préliminaires du gestionnaire de fichiers ont été réalisés comme de simples dessins. Ceux-ci ont été retouchés pour structurer logiquement les

éléments au sein de groupes, préparant ainsi le terrain pour mettre au point le modèle des objets conceptuels proposé par le SERVO.

À partir de ces maquettes, plusieurs activités ont pu être menées en parallèle. D'autres partenaires du projet ont réalisé le développement des SERVO, en raffinant le modèle conceptuel exposé par leur application. Une importante partie de leur travail a aussi consisté à mettre au point le maintien de la cohérence de leur application grâce à un protocole transactionnel [Zhao *et al.*, 2002, Blanch *et al.*, 2005]. Dans le même temps, et indépendamment, des comportements ont pu être ajoutés aux maquettes, et testés localement un utilisant un SERVIR simplifié ne gérant pas de SCOG, et utilisant un simple document SVG comme POG, sans le générer lui-même. Les interactions ont ainsi pu être testées sans être reliées au noyau fonctionnel de l'application. Ce travail a permis d'ajuster de manière cohérente la structure du document SVG et les comportements interactifs qui lui sont associés. Cette phase a conduit à la production de prototypes adaptés à la démonstration.

Intégration fonctionnelle

Pour pouvoir établir la connexion entre le SERVO et le SERVIR, l'étape suivante a consisté à mettre au point la transformation XSLT permettant de produire un POG à partir du COG mis en place par les développeurs du SERVO. Pour cela, le document SVG réalisant l'interface du prototype est utilisé comme trame. Pour l'explorateur de fichier, on peut reconnaître facilement dans la transformation XSLT donnée en Annexe B les éléments du document utilisés pour le prototype donné sur la Figure 3.6, page 51. À partir d'un exemple de COG (celui donné en haut de la Figure 5.2), il est alors facile de tester la conformité structurelle du POG généré par cette transformation (donné en bas de la Figure 5.2). Pour cela, on peut utiliser le même SERVIR simplifié et utiliser le document SVG généré en lieu et place de celui qui a été produit à la main pour le prototype.

Une fois ces étapes franchies, les pièces de l'application peuvent être assemblées. Il faut pour cela ajouter l'appel effectif des méthodes du SERVO aux comportements qui jusqu'à maintenant ne faisaient que manipuler le graphe SVG. Dans le cas du jeu par exemple, dans la phase de prototypage, sélectionner une colonne ajoutait directement un pion au document SVG comme le montre la Figure 5.6 en haut. Ce comportement a été simplement légèrement modifié pour communiquer avec le SERVO, comme le montre la Figure 5.6 en bas, et ce sans avoir besoin de modifier le SERVIR lui-même.

Raffinement

En parallèle de cette intégration fonctionnelle, l'interface peut continuer à être raffinée. Les designers graphiques peuvent entrer en jeu pour améliorer l'aspect visuel de l'interface. Ils peuvent pour cela travailler en se basant sur le document SVG issu du prototypage. Ce travail peut ne pas se limiter à de simples ajustements, puisque les seules contraintes à respecter sont celles exprimées par les contrats structurels requis par les comportements interactifs. Des éléments peuvent donc être ajoutés sans remettre en cause l'interaction. L'intégration de ces raffinements doit par contre passer par leur transposition dans la transformation XSLT. Là encore cependant, seule cette partie de l'ap-

```
void Column::press() {
   // manipulation du document SVG
   window->lock();
   svg::SVGCircleElement *coin = new svg::SVGCircleElement(window->svgDocument);
   hsm::svgl::setAttribute(coin, "hsm:behaviour", "coin");
   hsm::svgl::setAttribute(coin, "hsm-arg:col", id);
   window->svgContent->appendChild(coin);
   window->instanciateBehaviour(coin);
   window->unlock();
}
```

```
void Column::press() {
   // interaction avec le SERVO par l'intermédiaire du SERVIR
   Servir *servir;
   hsm::svgl::getAttribute(window->svgContent, "indigo:servir", servir);
   servir->postMethodToSERVO(id, "play");
}
```

FIG. 5.6 – Modification du comportement de la colonne pour permettre l'interaction avec le SERVO

plication a besoin d'être mise à jour, les modifications n'affectant pas les autres pièces de l'interaction. De même, les comportements peuvent être eux aussi raffinés sans remettre en cause le reste des choix, et sans nécessité de mise à jour complète de l'application.

5.3.3 Conclusion

L'utilisation de notre boîte à outils HsmTk dans le contexte du projet INDIGO a permis de vérifier ses qualités. En particulier, le support à la modularité qu'elle offre grâce par exemple aux machines à états hiérarchiques encourage la réutilisation du code et sa factorisation. Le découplage entre le modèle graphique et les interactions facilite, lui, la mise au point incrémentale et le prototypage rapide. La facilité avec laquelle les techniques d'interaction sont ainsi mises en œuvre ouvre un espace de conception qu'il est facile d'explorer. Ainsi, s'il est facile de réutiliser l'existant, il est toujours tentant de mettre au point des solutions adaptées finement aux problèmes particuliers rencontrés.

Chapitre 6

Le pointage sémantique

Nous présentons ici le pointage sémantique, une technique d'interaction originale qui facilite la tâche de sélection d'objets graphiques à l'aide d'un pointeur. Nous montrons ensuite comment cette technique d'interaction, qui met en jeu des aspects à la fois de bas niveau (accélération de la souris) et de haut niveau (sémantique des objets graphiques) de l'interaction, peut être mise en œuvre grâce à la boîte à outils HsmTk.

Le pointage sémantique est une technique d'interaction facilitant la tâche de sélection à l'aide d'un pointeur dans les interfaces graphiques. Cette technique repose sur l'adaptation du facteur qui lie le déplacement du pointeur à celui du périphérique de pointage auquel il est lié, cette adaptation étant fonction de la sémantique des objets présents à l'écran. Mettre en œuvre effectivement cette technique nécessite donc d'une part de maîtriser le lien existant entre le pointeur et le périphérique de pointage, et d'autre part de pouvoir accéder aux objets graphiques présents à l'écran et à leur sémantique. Ces deux aspects rendent la réalisation du pointage sémantique impossible pour des applications courantes et délicate avec les boîtes à outils existantes. Cependant, grâce à la boîte à outils HsmTk, nous avons été en mesure de vérifier expérimentalement l'intérêt du pointage sémantique, et de prototyper des interfaces l'utilisant.

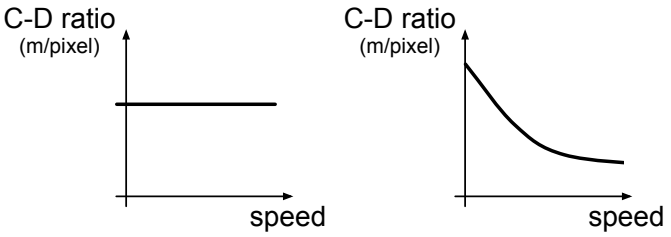


FIG. 6.1 – Le CD ratio fonction de la vitesse de la souris

Pour commencer, nous présentons dans ce chapitre le pointage sémantique lui-même. Nous exposerons ensuite comment HsmTk nous a permis de mettre en œuvre cette technique d'interaction avancée.

6.1 Présentation du pointage sémantique

La sélection de cibles par le pointage est l'une des tâches fondamentales des interfaces graphiques actuelles. À mesure que les applications permettent d'effectuer des tâches de plus en plus complexes, le nombre de cibles potentielles représentant des objets d'intérêts ou des instruments d'interaction augmente obligeant à un compromis sur leurs tailles, et rendant celles-ci difficiles à atteindre. Dans ce contexte, beaucoup de techniques d'interaction ont été proposées pour faciliter le pointage.

En se basant sur le modèle du pointage de Fitts [1954] présenté à la Section 2.1.3, nous savons que pour faciliter le pointage d'une cible, il faut la rendre plus grande et/ou plus proche. Cette idée simple est à la base de nombreuses techniques d'aide au pointage qui approchent les cibles [Callahan *et al.*, 1988, Baudisch *et al.*, 2003, Cockburn et Firth, 2003] ou les dilatent [McGuffin et Balakrishnan, 2002, Zhai *et al.*, 2003, Grossman et Balakrishnan, 2005] dont nous avons présenté quelques exemples à la Section 2.2.1.

Une autre voie a été explorée par ailleurs, elle consiste à modifier le ratio qui lie les déplacements du curseur à ceux de la souris (CD ratio pour *Control to Display ratio* [MacKenzie et Riddersma, 1994]). Une telle technique est utilisée dans pratiquement tous les systèmes graphiques depuis longtemps, il s'agit de l'accélération de la souris. Elle consiste à modifier le CD ratio pour que le déplacement du curseur ne soit pas simplement proportionnel à celui de la souris, mais pour que lorsque celle-ci est déplacée rapidement, le curseur se déplace encore plus rapidement. Le système tient ainsi compte du fait que les déplacements effectués à grande vitesse le sont pour atteindre des cibles a priori distantes. En amplifiant les déplacements, il permet alors de les atteindre plus rapidement. La Figure 6.1 montre à gauche un ratio constant quelle que soit la vitesse du curseur, et à droite un ratio qui décroît avec celle-ci, comme dans le cas classique de l'accélération de la souris. Le ratio exprime ici l'amplitude du déplacement de la souris (en mètres par exemple) nécessaire pour que le curseur effectue un déplacement donné (en pixels). Un ratio faible indique donc qu'il n'y a pas besoin de déplacer beaucoup la souris pour effectuer des mouvements de grande amplitude avec le curseur.

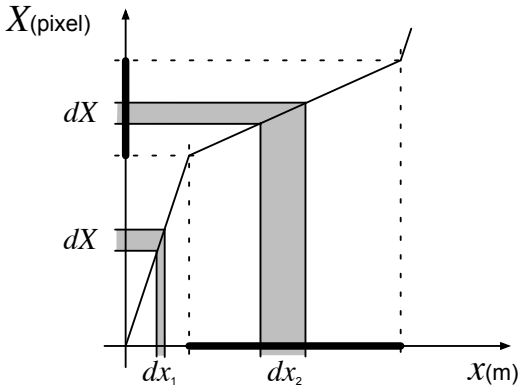


FIG. 6.2 – Adaptation du C-D ratio à l'intérieur d'une cible

6.1.1 Principe du pointage sémantique

Le principe du pointage sémantique, qui formalise et unifie plusieurs travaux antérieurs [Keyson, 1997, Worden *et al.*, 1997, Cockburn et Firth, 2003], est de rendre le ratio dépendant, non plus de la vitesse du périphérique de pointage, mais de la position du curseur par rapport aux objets de l'écran. Ainsi, augmenter le ratio à l'intérieur d'une cible y rend le curseur plus lent à vitesse de déplacement constante de la souris : celle-ci doit en effet alors couvrir une distance plus importante pour traverser un même nombre de pixels. La Figure 6.2 illustre cette idée en montrant la correspondance entre le monde de la souris en abscisse et le monde du curseur en ordonnée. Le monde de la souris est celui de la table sur laquelle elle est posée, et nous utiliserons des lettres minuscules (x) pour y faire référence, l'unité choisie pour exprimer les longueurs dans ce monde étant le mètre. Le monde du curseur est l'écran, et nous utiliserons des lettres majuscules (X) et les pixels pour exprimer les grandeurs qui y sont associées. Choisir un ratio plus important à l'intérieur de la cible (matérialisée par un trait épais) veut dire que pour parcourir la même distance à l'écran (dX), l'amplitude du déplacement de la souris nécessaire est moindre hors de la cible (dx_1) qu'à l'intérieur de celle-ci (dx_2). Le ratio est donc sur cette figure l'inverse de la pente de la courbe qui permet de passer d'un monde à l'autre.

L'interprétation classique du gain apporté par cette famille de techniques est que la lenteur du curseur à l'intérieur des cibles facilite la sélection de celles-ci. Inversement, un faible ratio entre les cibles permet de franchir rapidement les espaces vides qui les séparent. C'est souvent en termes de retour que l'effet est commenté : Worden *et al.* [1997] parlent d'icones "collants", Lécuyer *et al.* [2000] de retour pseudo-haptique. Nous avons pour notre part choisi d'interpréter l'adaptation du CD ratio par l'impact qu'il a sur le mouvement de pointage lui même, et donc de considérer les modifications qu'il apporte au mouvement réalisé en entrée [Blanch *et al.*, 2004].

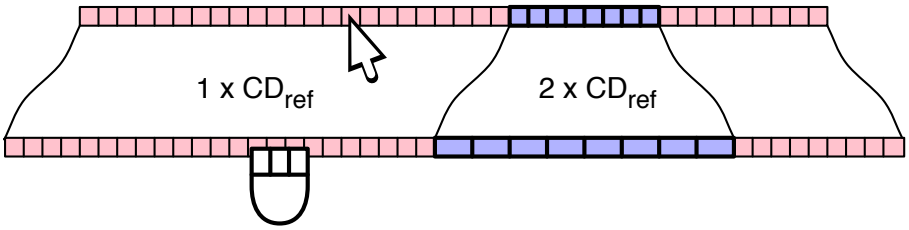


FIG. 6.3 – Le CD ratio comme échelle de l'espace moteur

6.1.2 Le CD ratio comme échelle de l'espace moteur

Comme on l'a noté précédemment, le CD ratio, s'il est uniquement fonction de la position du curseur, peut être interprété comme le rapport entre les tailles des objets dans les espaces visuels (l'écran) et moteur (la table). La Figure 6.3 montre ainsi en haut le curseur et une ligne de pixels à l'écran, au sein de laquelle une cible est symbolisée par un contour plus épais pour les pixels qui en font partie. En bas, la souris et l'image de la ligne de pixels sur la table sont représentées. À chaque pixel de l'écran correspond une zone de l'espace moteur dans lequel la souris doit se trouver pour que le curseur se situe dans le pixel correspondant à l'écran. Ici, avec un CD ratio deux fois plus élevé à l'intérieur de la cible qu'à l'extérieur, la largeur de l'image des pixels de la cible est double de celle des pixels situés en dehors. Le CD ratio est donc bien l'échelle relative de l'espace moteur par rapport à l'espace visuel. Augmenter cette échelle à l'intérieur des cibles revient à les dilater par rapport aux zones vides dans l'espace moteur.

Ainsi, à basse échelle, le mouvement pour atteindre une cible est réduit, mais la taille de la cible est petite aussi. À grande échelle, la distance de la cible est plus importante, mais la précision requise pour l'atteindre est plus faible puisqu'elle est aussi dilatée. On retrouve là le fait que l'indice de difficulté dans le modèle de Fitts est insensible à l'échelle de la tâche (voir la Section 2.1.3) et que le choix d'un CD ratio particulier n'affecte pas cette propriété ([MacKenzie et Riddersma, 1994] montre que même si on note une influence de ce choix sur la performance, il affecte de manière inverse le taux d'erreur). Cependant, en adaptant le CD ratio dynamiquement en fonction de la position du curseur, on peut choisir une échelle faible, appropriée à l'amplitude de la tâche lorsque la cible est distante, puis une échelle plus grande, adaptée à la sélection de la cible lorsque le curseur s'en approche, et ce sans altérer la représentation à l'écran. La cible peut alors être rendue à la fois plus proche et plus grosse dans l'espace moteur.

La précision des mouvements peut être adaptée en modifiant l'échelle des différentes parties de l'écran suivant qu'elles constituent une cible potentielle (par exemple un bouton, ou un icone) ou non (le fond du bureau par exemple). Nous avons mis en évidence [Guiard *et al.*, 2004] qu'il y avait une différence importante entre une tâche de sélection par pointage et la tâche abstraite qui est en fait réalisée. Par exemple, la sélection d'un icone sur un bureau consiste à pointer l'un des 48×48 pixels constituant la cible parmi les 1600×1200 qui constituent l'écran. En terme d'information transmise, on peut estimer que cette tâche revient à fournir environ $\log_2(\frac{1600 \times 1200}{48 \times 48}) \sim 10$ bits au système.

Cependant, si le bureau comporte 64 icones, le choix de l'un d'entre-eux ne représente en fait que $\log_2(64) = 6$ bits. La différence observée — 4 bits, soit tout de même 40 % de la tâche — est en fait due à toute la partie préliminaire du mouvement ayant lieu dans le vide qui indique au système que les pixels du fond de l'écran ne sont pas intéressants, cette information lui étant pourtant a priori connue.

Le but du pointage sémantique est donc de réconcilier ces deux tâches en permettant au système de prendre en compte des informations dont il dispose a priori, pour ainsi faciliter la tâche des utilisateurs. En adaptant l'échelle au contexte du pointeur, c'est-à-dire à la sémantique des pixels, le système adapte le compromis vitesse/précision des mouvements pour qu'il soit facile d'ignorer des zones vides et que l'essentiel des mouvements soit effectivement consacré à la sélection de cible. Une version extrême du pointage sémantique, le pointage d'objet [Guiard *et al.*, 2004], saute même l'espace vide pour déplacer directement le curseur sur la cible la plus proche dans la direction du mouvement.

6.1.3 Modèle

L'hypothèse que nous avons formulée pour expliquer les améliorations observées pour les techniques adaptant le CD ratio en fonction de la position du curseur est que la performance du pointage pour ce type de tâche est liée aux caractéristiques de la tâche dans l'espace moteur (où la cible est grossie) et non à celles de la tâche dans l'espace visuel. Si cette hypothèse est vraie, on peut dériver du modèle de Fitts une prédiction de l'amélioration des performances que ces techniques sont en mesure d'apporter. Nous avons ainsi exprimé la difficulté au sens de Fitts (Équation 2.2, page 20) dans l'espace moteur (id) en fonction de celle dans l'espace visuel (ID) traditionnellement utilisée. Pour cela, il nous faut expliciter la manière dont l'échelle est adaptée.

Formulation de l'échelle

La fonction la plus simple qui puisse être utilisée pour exprimer l'échelle en fonction de la position est une fonction constante par morceau (Figure 6.4). Elle peut être définie en utilisant la fonction rectangle Π définie de la façon suivante[1] :

$$\Pi(u) = \begin{cases} 1 & pour\ \|u\| \leq \frac{1}{2} \\ 0 & sinon \end{cases} . \tag{6.1}$$

Pour une cible de largeur W située à l'abscisse D, et d'échelle S, l'échelle s'exprime alors ainsi en fonction de la position du curseur X[2] :

$$\text{scale}(X) = \left(1 - \Pi\left(\frac{\|X-D\|}{W}\right)\right) + S \times \Pi\left(\frac{\|X-D\|}{W}\right). \tag{6.2}$$

[1] La fonction réellement utilisée pour le pointage sémantique est en fait plus complexe mais les résultats auxquels elle mène sont les mêmes que ceux présentés ci-après. Cette fonction est détaillée en Annexe C.

[2] $\|X - D\|$ est la distance euclidienne du curseur (X) à la cible (D). Les formules sont valides quelle que soit la dimension.

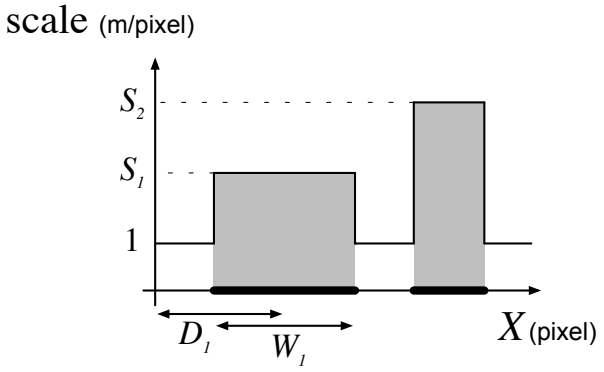


FIG. 6.4 – L'échelle comme fonction constante par intervalles

Le premier terme de la somme vaut en effet 1 à l'extérieur de la cible et 0 à l'intérieur de celle-ci. Le second terme vaut, lui, 0 à l'extérieur de la cible et fixe l'échelle à S à l'intérieur de celle-ci.

L'Équation 6.2 se généralise pour un nombre quelconque de cibles de la manière suivante :

$$\text{scale}(X) = \left(1 - \sum_i \Pi\left(\frac{\|X - D_i\|}{W_i}\right)\right) + \sum_i S_i \times \Pi\left(\frac{\|X - D_i\|}{W_i}\right) \tag{6.3}$$

où D_n, W_n et S_n sont les positions, tailles et échelles de la cible n (Figure 6.4).

Calcul de l'indice de difficulté dans l'espace moteur

Dans le cas d'un monde à une dimension, il est facile de calculer les distances dans l'espace moteur puisqu'un seul chemin est possible pour aller d'un point à un autre. Il suffit donc d'intégrer la fonction d'échelle entre les points considérés pour connaître la distance qui les sépare. Pour connaître la taille de la cible dans l'espace moteur w, on a donc :

$$\begin{aligned} w &= \int_{D-\frac{W}{2}}^{D+\frac{W}{2}} \text{scale}(X)\, dX \\ &= S \times W. \end{aligned} \tag{6.4}$$

On peut remarquer que la taille dans l'espace moteur w est exactement l'aire des zones grisées de la Figure 6.4. Dans le cas d'une cible unique qui est celui de l'expérimentation[3], on peut calculer de la même manière la distance de la cible à l'origine dans l'espace moteur d :

$$\begin{aligned} d &= \int_0^D \text{scale}(X)\, dX \\ &= \left(D - \tfrac{W}{2}\right) + S \times \tfrac{W}{2}. \end{aligned} \tag{6.5}$$

[3] L'impact des distracteurs n'a pas été étudié dans notre expérimentation.

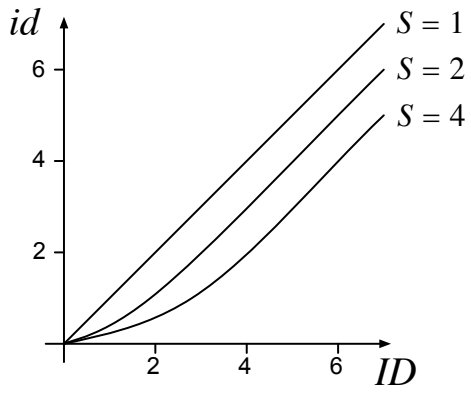


FIG. 6.5 – Indice de difficulté dans l'espace moteur en fonction de celui de l'espace visuel

Le premier terme de la somme correspond à la partie du mouvement séparant l'origine du bord de la cible, et qui se situe donc en dehors de la cible. L'échelle n'intervient alors pas sur cette partie. Le second terme correspond à la partie du mouvement qui a lieu à l'intérieur de la cible, du bord de celle-ci à son centre, et dont l'amplitude correspond bien à la moitié de sa largeur multipliée par son échelle. Si D est petite, le premier terme devient négligeable puisque la partie de D qui ne se trouve pas dans la cible tend alors vers 0. Par ailleurs, si D est grand devant $W/2$, c'est alors le second terme qui devient négligeable devant le premier.

En partant des Équations 6.4 et 6.5, on peut exprimer l'indice de difficulté de la tâche[4] dans l'espace moteur *id* en fonction de l'indice de difficulté habituel ID et de l'échelle de la cible S :

$$\begin{aligned} id \quad &= \quad \log_2\left(\frac{d}{w/2}\right) \\ &\overset{D\to W/2}{\longrightarrow} \quad \log_2\left(\frac{S\times D}{S\times W/2}\right) = ID \\ &\overset{D\gg W/2}{\longrightarrow} \quad \log_2\left(\frac{D}{S\times W/2}\right) = ID - \log_2(S). \end{aligned} \tag{6.6}$$

La Figure 6.5 montre le lien entre ces deux indices de difficulté pour des échelles de l'espace moteur valant 1 (cas standard), 2 et 4. On constate que, pour les ID importants, la difficulté dans l'espace moteur est réduite d'un bit à chaque fois que l'échelle double. Sachant que les indices de difficulté rencontrés sur un bureau standard dépassent rarement 10 (ce qui correspond à une cible plus de mille fois plus petite que la distance à laquelle elle est située), gagner ne serait-ce qu'un bit est significatif.

[4] Nous avons opté pour la formulation originale de Fitts pour l'ID $\left(\log_2(\frac{D}{W/2})\right)$, plutôt que celle basée sur la théorie de l'information de Shannon $\left(\log_2(\frac{D}{W}+1)\right)$ [MacKenzie, 1989], pour faciliter les calculs analytiques. Il faut cependant noter que ces formules sont équivalentes asymptotiquement et que les résultats numériques sont pratiquement identiques quelle que soit la formulation utilisée.

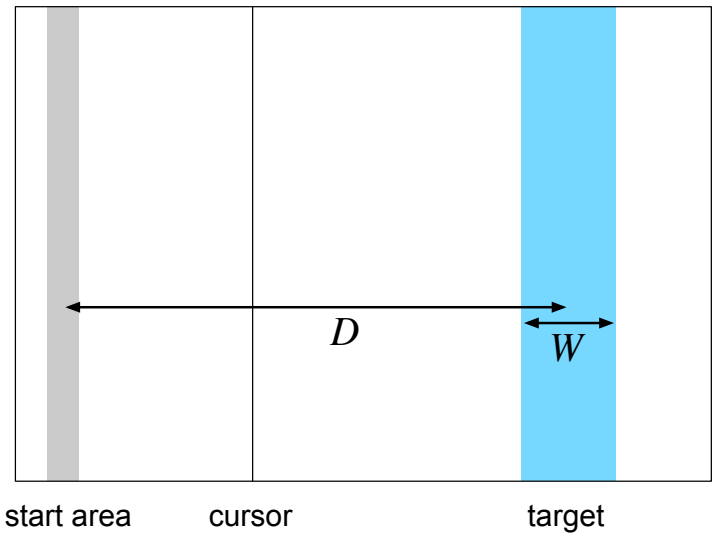


FIG. 6.6 – Écran du protocole expérimental

6.2 Expérimentation

Pour corroborer notre hypothèse, nous avons réalisé une expérimentation contrôlée. Nous détaillons ici son protocole, et analysons ses résultats.

6.2.1 Protocole expérimental

Tâche

La tâche consistait à effectuer une succession de pointages discrets, une seule dimension étant considérée. Les participants devaient positionner le curseur, représenté par une ligne verticale noire d'un pixel d'épaisseur, à l'intérieur d'une zone de départ située à gauche de l'écran et matérialisée par un rectangle gris. Après que le curseur soit resté complètement immobile à l'intérieur de cette zone pendant une demi-seconde, une cible constituée d'une zone bleue apparaissait à sa droite, et le sujet devait l'atteindre et cliquer à l'intérieur de celle-ci (Figure 6.6).

Après chaque clic, un retour visuel indiquait au sujet s'il avait ou non atteint la cible. Il était alors invité à repositionner le curseur dans la zone de départ pour un nouvel essai. Après chaque série, un taux d'erreur était indiqué aux participants et ils étaient encouragés par un message à accélérer si celui-ci était inférieur à 4 % et à être plus précis dans le cas inverse.

Sujets et Matériel

Les participants, au nombre de 12 (11 hommes et une femme), étaient âgés de 27.2 ans en moyenne ($SD = 6.1$ ans).

L'écran utilisé pour l'expérimentation avait une diagonale de 22 pouces pour une résolution de 1600×1200 pixels. Le dispositif de pointage était un

puck sur une tablette graphique Wacom Intuos de taille similaire à celle de l'écran (12×18 pouces). Le CD ratio servant de référence était la résolution de l'écran (1 cm à l'écran correspondait donc exactement à 1 cm sur la tablette), et l'accélération de la souris était désactivée.

Conditions

Trois conditions d'échelle ont été utilisées :
- la première, *contrôle*, conservait l'échelle constante, les tâches visuelle et motrice correspondant alors complètement ;
- la seconde, *double*, adaptait l'échelle à l'intérieur de la cible pour que celle-ci double de taille dans l'espace moteur ; et
- la dernière, *quadruple*, adaptait l'échelle de manière à ce que la taille de la cible soit multipliée par quatre.

Une description précise de la fonction d'échelle utilisée est donnée en Annexe C.

Cinq indices de difficulté pour l'espace visuel ont été testés : $ID = 4$, 5, 6, 7 et 8, ces valeurs balayant l'ensemble du spectre des tâches de pointage typiques réalisées sur un bureau classique. Deux tailles pour la tâche ont été retenues : $D = 512$ ou 1024 pixel. En croisant ces deux variables, l'expérimentation comportait donc 10 tâches différentes.

Séries

Une série pseudo-aléatoire de 100 tâches, comportant chacune des tâches possibles 10 fois a été construite. Celle-ci neutralisait les effets d'ordre en équilibrant l'ordre moyen de passage de chaque tâche, et en comportant une et une fois seulement toutes les paires successives de tâches possibles. Cette série était répétée trois fois par participant, qui ont donc produit chacun 300 échantillons.

Ces 300 tâches ont été séparées en 6 séries de 50 essais (5 fois chaque tâche), les séries 1 et 4 étant réalisées avec la même condition d'échelle, ainsi que les séries 2 et 5, et les séries 3 et 6. Six permutations des trois conditions d'échelle étant possibles, elles ont été distribuées chacune à deux sujets.

Enfin, chaque série était précédée de 10 tâches choisies au hasard et réalisées dans la même condition que la série qui les suivait. Ces essais n'étaient pas enregistrés, et servaient aux sujets à prendre leurs marques. En effet, une étude pilote avait montré qu'après une dizaine d'essais, le temps de mouvement se stabilisait.

6.2.2 Résultats

Les effets du pointage sémantique ont été observés en analysant les trois variables dépendantes suivantes :
- le temps de réaction (RT) qui caractérise le temps qui sépare l'apparition de la cible du début du mouvement ;
- le temps de mouvement (MT) qui caractérise le temps qui sépare le début du mouvement de l'instant du clic ; et
- le taux d'erreur (ER) qui comptabilise la proportion de cibles manquées.

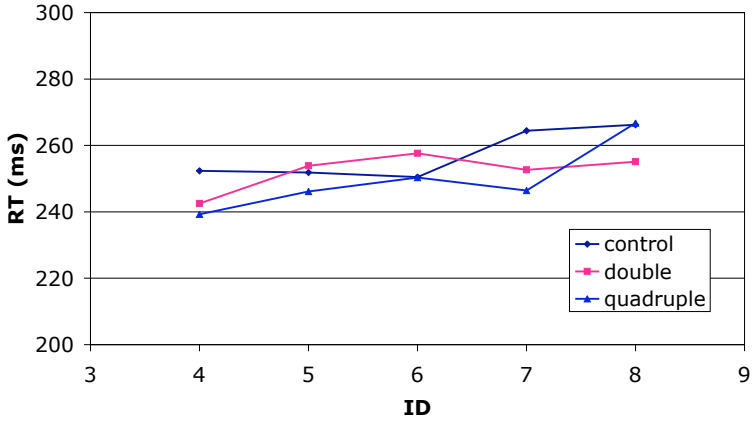


FIG. 6.7 – Temps de réaction en fonction de l'indice de difficulté

Une analyse de la variance de ces trois variables a permis de mettre en évidence les effets des trois facteurs considérés (3 conditions d'échelle, 5 indices de difficulté, et 2 tailles).

Effets non significatifs

Effet de la taille de la tâche. Aucun effet statistiquement significatif de la taille de la tâche (D) n'a été noté sur les trois variables. Ce résultat est en complet accord avec le modèle de Fitts qui affirme que seule la difficulté de la tâche, et non sa taille, la caractérise. Pour le reste de l'analyse, nous avons donc réduit les variables aux seules conditions d'échelle et d'indices de difficulté.

Effet sur le temps de réaction. Le temps de réaction (RT) a été de 253 ms en moyenne avec de faibles variations ($SD = 75.76$ ms). RT augmente légèrement avec la difficulté de la tâche mais cet effet n'est pas statistiquement significatif. Aucune différence significative n'a pu non plus être mise en évidence entre les trois conditions d'échelle (Figure 6.7).

Effet du pointage sémantique sur le temps de mouvement

Le temps de mouvement moyen (MT) en fonction de l'indice de difficulté dans l'espace visuel (ID) est montré dans la Figure 6.8 pour les trois conditions d'échelle. Un effet significatif de la condition d'échelle ($F_{2,33} = 5.35$, $p = .0097$) ainsi que de l'ID ($F_{4,55} = 30.04$, $p < .0001$) est observé sur MT mais aucune interaction significative entre ces deux effets n'a été mise en évidence.

La Figure 6.8 est à comparer avec la Figure 6.5 qui montre la difficulté dans l'espace moteur en fonction de celle dans l'espace visuel. Les courbes ayant la même allure, on a là la première confirmation que le temps de pointage évolue bien conformément à l'indice de difficulté dans l'espace moteur (id). Comme prévu, le bénéfice du pointage sémantique commence par croître avec l'ID avant de devenir pratiquement constant pour les tâches difficiles. Le maximum de ce bénéfice relatif (temps économisé par rapport au temps sans aide) est obtenu pour $ID = 6$ mais dès que $ID \geq 5$, la réduction de MT est d'au

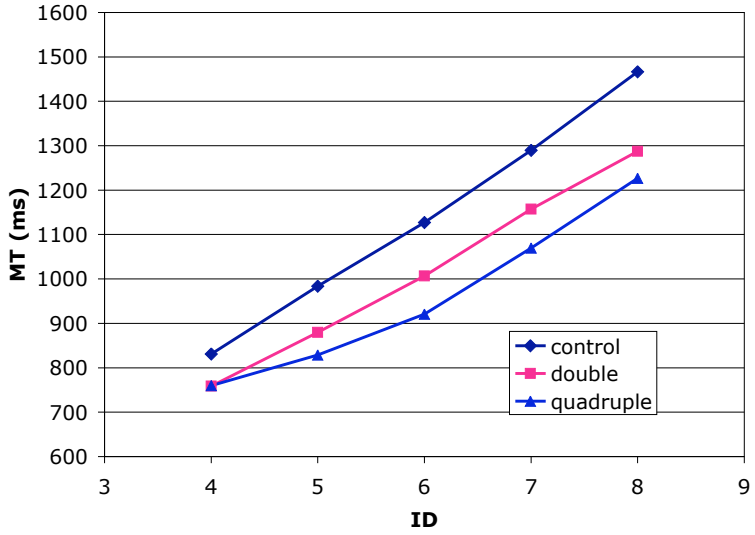


FIG. 6.8 – Temps de mouvement en fonction de l'indice de difficulté

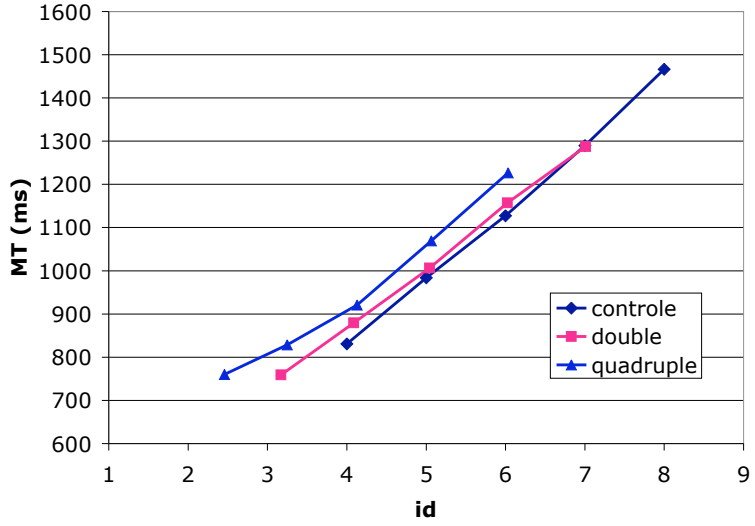


FIG. 6.9 – Temps de mouvement en fonction de l'indice de difficulté exprimé dans l'espace moteur

moins 10 % (10.9 % en moyenne) pour la condition *double* et d'au moins 15 % (16.9 % en moyenne) pour la condition *quadruple*.

La Figure 6.9 montre le temps de mouvement MT en fonction de l'indice de difficulté de la tâche exprimé dans l'espace moteur (*id*). Si notre hypothèse supposant que la performance de la tâche de pointage est régie par l'indice de difficulté dans l'espace moteur est valide, ces courbes utilisant *id* devraient être en meilleur accord avec le modèle de Fitts que les courbes précédentes utilisant ID (Figure 6.8). Le Tableau 6.1 donne le coefficient de détermination

(r^2) et le terme d'erreur quadratique moyenne ($RMSE$) lorsqu'on explique les variations de MT par ID et par *id* dans le modèle de Fitts. On constate qu'utiliser *id* comme variable pour expliquer MT donne de meilleurs résultats puisqu'il explique près de 97 % des variations de MT contre 85 % en utilisant ID. Ce résultat conforte donc notre hypothèse.

	ID	*id*
r^2	.849461	**.96829**
$RMSE$	86.897	**39.882**

TAB. 6.1 – r^2 et $RMSE$ des régressions linéaires de MT

Cependant, on peut observer sur la Figure 6.9 que pour la condition *quadruple*, le bénéfice apporté par le pointage sémantique est moins important que prévu. En effet, si les sujets avaient pleinement tiré parti de la facilitation qu'il est censé introduire, les trois courbes devraient alors se superposer. L'étude du taux d'erreur va nous permettre de mieux comprendre cette écart aux résultats attendus.

Effet du pointage sémantique sur le taux d'erreur

Les participants à l'expérimentation ont reçu comme instruction de se conformer à un taux nominal d'erreur de 4 %. La moyenne du taux d'erreur (ER) se situe en fait à 4.26 % et les différences entre les trois conditions d'échelle sont significatives (Figure 6.10). En moyenne, ER a été de 6.2 % pour la condition *contrôle*, de 4.25 % pour la condition *double* et de seulement 2.35 % pour la condition *quadruple*. Quel que soit l'ID, les mouvements ont été plus précis pour les conditions *double* et *quadruple* que pour la condition *contrôle*, et pour les ID supérieurs à 4, la condition *quadruple* a toujours obtenu des meilleurs taux d'erreur que les deux autres conditions. On peut donc estimer que si le pointage sémantique n'a pas été complétement exploité pour réduire le temps de pointage, c'est parce qu'il a aussi permis d'améliorer la précision des mouvements.

Variations inter-sujets

Cette dernière analyse est confirmée par une analyse individuelle des performances. Les moyennes reportées jusqu'à maintenant sont représentatives de la plupart des individus. Cependant différentes stratégies ont été observées : certains individus ont essentiellement tiré parti du pointage sémantique en réduisant leur taux d'erreur, alors que d'autres ont respecté scrupuleusement le taux nominal de 4 %. Pour ces derniers, on constate que le temps de pointage est strictement gouverné par l'indice de difficulté de la tâche dans l'espace moteur.

Ce résultat confirme que le pointage sémantique facilite indubitablement le pointage, cette facilitation profitant dans des proportions variables au temps de pointage et à la précision du mouvement. Il faut enfin noter que les sujets n'ont remarqué aucun comportement inhabituel et ne se sont pas aperçus des différences de performances, pourtant significatives, entre les

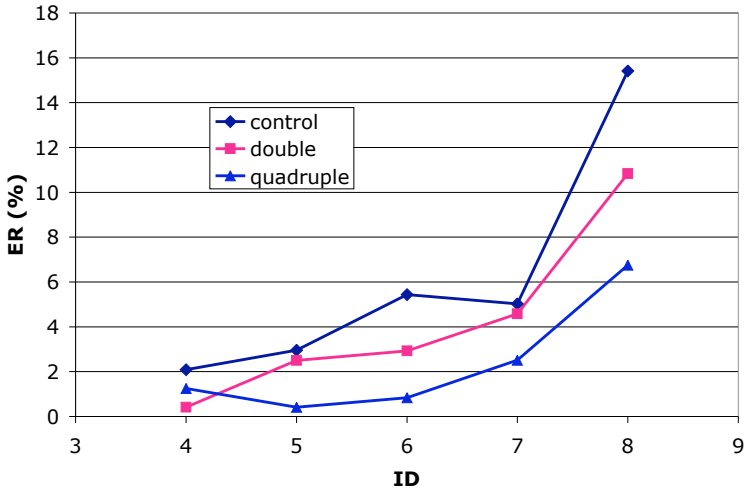


FIG. 6.10 – Le taux d'erreur en fonction de l'indice de difficulté

différentes conditions. Ceci montre que la technique est transparente, à l'instar de "l'accélération" de la souris.

6.3 Implications pour la conception d'interfaces

Si nous avons montré que le pointage sémantique permettait de rendre des cibles plus grandes dans l'espace moteur, le degré de liberté introduit par le choix d'une échelle locale permet d'aller plus loin. Nous explorons ici l'espace de conception pour les interfaces graphiques ouvert par le pointage sémantique.

6.3.1 Deux tailles pour un même objet

Nous avons vu que l'adaptation du CD ratio permet de changer l'échelle de l'espace moteur localement. Elle permet donc de découpler complètement la taille des objets à l'écran de celle qu'ils ont pour la manipulation. En effet, à taille visuelle fixée, on peut choisir arbitrairement la taille motrice d'un objet en utilisant une échelle déterminée par le rapport des deux tailles choisies. Dans les interfaces graphiques usuelles la taille d'un objet est fixée par un ensemble de contraintes : il doit être suffisamment étendu pour permettre de présenter l'information qu'il recèle, suffisamment étendu aussi pour être manipulé s'il y a lieu, mais il doit aussi être le plus petit possible pour permettre de présenter globalement le plus d'information possible dans l'espace limité de l'écran. Ainsi, la taille d'un objet qui présente peu d'information, comme un simple bouton ou une barre de défilement, est principalement déterminée par la contrainte de manipulation, ce qui gaspille de l'espace à l'écran qui pourrait être utilisé à meilleur escient. Inversement, quand beaucoup d'information doit être présentée, comme sur une page Web, les éléments manipulables, comme les hyperliens, peuvent devenir très petits et l'interaction devient alors difficile.

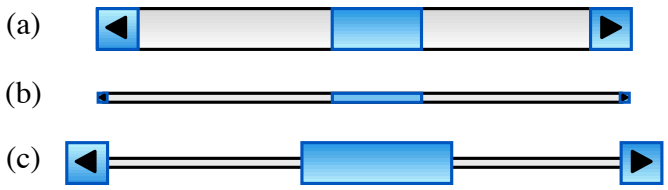


FIG. 6.11 – Reconception de la barre de défilement

(a) version originale ;
nouvelle version : (b) dans l'espace visuel ; et
(c) dans l'espace moteur

Le degré de liberté introduit par le pointage sémantique permet alors de choisir les deux tailles, motrice et visuelle, des objets indépendamment, et de faire des compromis différents dans les deux espaces. La taille visuelle peut ainsi être choisie simplement en fonction de l'information qu'un objet doit présenter, libérant alors des espaces consacrés à rendre les objets manipulables. Inversement, la taille dans l'espace moteur peut être choisie seulement en fonction de l'intérêt des objets pour la manipulation.

6.3.2 Reconception d'interacteurs traditionnels

Nous montrons le potentiel du pointage sémantique en examinant quelques interacteurs usuels, et en montrant qu'ils peuvent facilement être adaptés soit pour réduire leur emprise à l'écran tout en conservant leur maniabilité, soit encore pour faciliter leur manipulation tout en préservant leur aspect. Pour conduire cette reconception, nous nous sommes posé les questions suivantes pour chaque objet :

- Quelle quantité d'information apporte cet objet ?
- Qu'elle est l'importance de cet objet pour la manipulation ?

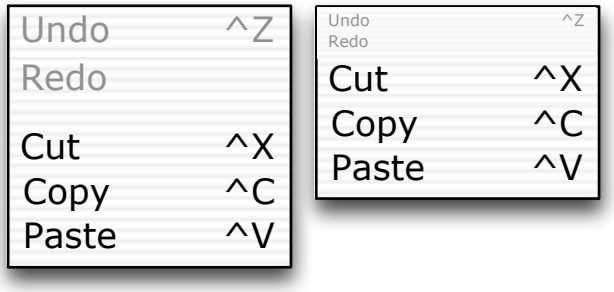
Barre de défilement

Une barre de défilement présente assez peu d'information aux utilisateurs : elle précise une position dans un document et, si la taille de l'ascenseur s'adapte, la proportion du document visible à l'intérieur de la vue actuelle. Cependant, elle occupe typiquement une bande de 15 pixels de large sur la totalité des fenêtres (Figure 6.11a). Elle pourrait donner la même information en utilisant une bande bien plus fine, de 3 pixels par exemple (Figure 6.11b). Pour préserver l'utilisabilité de la barre, il est alors nécessaire de choisir une échelle de 5 pour l'ascenseur et les flèches, ce qui rend à ces parties leur taille originale dans l'espace moteur (Figure 6.11c[5]).

Menus

La contrainte visuelle des menus est que les étiquettes des entrées qui les composent doivent être lisibles. Leur taille visuelle ne peut donc pas être mo-

[5] La déformation de l'espace moteur créée par le pointage sémantique ne peut se projeter exactement en géométrie euclidienne. La représentation qui en est donnée ici n'est donc pas exacte et n'est donnée qu'à titre d'illustration.



(a) (b)

FIG. 6.12 – Reconception d'un menu

(a) version originale conservée dans l'espace visuel ;
(b) nouvelle version dans l'espace moteur

(a)

(b)

FIG. 6.13 – Reconception d'une boîte de dialogue

(a) version originale conservée dans l'espace visuel ;
(b) nouvelle version dans l'espace moteur

difiée (Figure 6.12a). Cependant, l'importance relative des différents items du menu pour la manipulation est variable. Par exemple, les éléments désactivés et les séparateurs qui ne sont pas intéressants pour la manipulation peuvent être rendus plus petits dans l'espace visuel, réduisant alors la distance qui sépare les éléments intéressants du haut du menu (Figure 6.12b).

Boutons et hyperliens

Comme pour les menus, les boutons ou les étiquettes des boîtes de dialogue doivent rester lisibles, leurs tailles visuelles doivent donc être conservées (Figure 6.13a). Cependant, pour la manipulation, seuls les boutons ont une importance, et le reste de la boîte de dialogue peut donc être réduit.

On peut aller plus loin : en effet, l'importance des différents boutons n'est pas nécessairement la même. Le bouton sélectionné par défaut est le plus important car il est supposé représenter le choix le plus probable de l'utilisateur. Il peut donc être rendu plus grand que les autres pour faciliter l'action la plus fréquente. D'une manière générale, la taille des boutons dans l'espace visuel peut être choisie en fonction de leur probabilité d'être utilisés (Figure 6.13). Les boutons qui déclenchent des actions "dangereuses" peuvent même voir leur taille réduite pour diminuer le risque qu'ils soient utilisés par erreur, comme pour le bouton *"Don't Save"* de la boîte de dialogue présenté sur la figure.

Dans un même esprit, les documents comme les pages Web qui sont souvent conçues graphiquement en fonction de considérations esthétiques, ou d'objectifs de communication visuelle, peuvent bénéficier de ce type d'aménagements ne modifiant pas leur aspect. Pour de tels documents, l'interaction est principalement liée à l'activation d'hyperliens. En les grossissant dans l'espace moteur, on peut faciliter la navigation, tout en préservant la mise en forme visuelle du document.

6.3.3 Prolongements

En l'état, le pointage sémantique peut encore être amélioré. Tout d'abord, d'autres informations peuvent être prises en compte pour adapter l'échelle des objets. Ensuite, la problématique des distracteurs (cibles potentielles présentes sur la trajectoire du pointage) n'a pas été abordée ici.

Modifications dynamiques de l'importance

Les exemples donnés jusqu'à présent considèrent l'importance des objets comme un attribut statique. Cependant, dans l'exemple du menu, le fait qu'un élément soit désactivé ou non change en fonction du contexte. Dans ce cas, son échelle doit varier au cours du temps en fonction de son état. De la même manière, le pointage sémantique pourrait être utilisé pour adapter dynamiquement l'importance de certains objets de l'interface en fonction de leur état. Par exemple, lorsqu'une application qui n'est pas au premier plan nécessite l'intervention de l'utilisateur, son icone clignote dans la barre des tâches de Microsoft Windows, ou s'anime dans le Dock de Mac OS X. Dans ce cas, il est probable que l'utilisateur pointe cet icone pour activer l'application en question. Le système, qui a connaissance de l'état de l'application, pourrait donc augmenter la taille de l'icone concerné pour rendre sa sélection plus facile.

En allant plus loin, le pointage sémantique pourrait utiliser l'état du système, et l'historique de l'interaction pour adapter plus finement l'importance des objets. Les applications de la suite bureautique Microsoft Office peuvent par exemple avoir des menus qui s'adaptent aux utilisateurs en remontant les éléments les plus fréquemment utilisés en tête de menu. Cependant, il a été démontré que rendre ainsi la position des éléments des menus instable est source de confusions pour les utilisateurs [Somberg, 1987, Mitchell et Shneiderman, 1989]. Avec le pointage sémantique, on peut adapter la taille des éléments du menu en fonction de la fréquence de leur utilisation, favorisant ainsi l'usage des plus courants, et ce sans les déplacer.

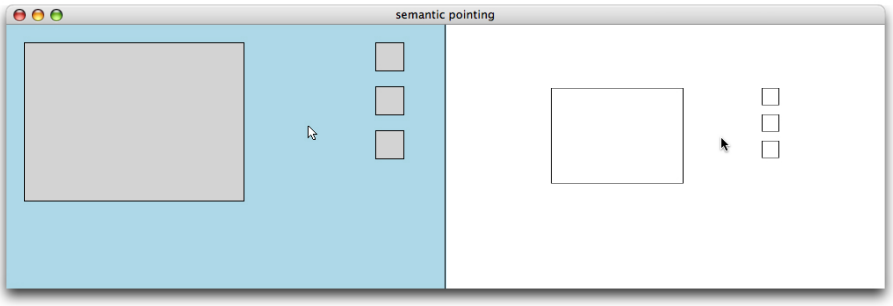


FIG. 6.14 – Premier prototype du pointage sémantique réalisé

Prise en compte des distracteurs

La présence d'une cible potentielle sur le chemin du pointage vers la cible réelle augmente sa distance en utilisant le pointage sémantique tel qu'il est décrit ici. Le bénéfice de la technique devient alors moindre et elle peut même dans certains cas avoir un impact négatif sur la performance. De même, pour certaines applications l'espace est totalement pavé de cibles. C'est le cas par exemple pour un logiciel de retouche photographique pour lequel chaque pixel peut être potentiellement la cible des interactions. Plusieurs pistes peuvent être explorées pour pallier ces problèmes, qui consistent là encore à analyser les informations dont le système a connaissance mais qu'il néglige.

6.4 Utilisation de HsmTk

Nous avons utilisé la boîte à outils HsmTk de plusieurs manières lors de notre recherche sur le pointage sémantique. Elle nous a permis en particulier de développer rapidement un prototype pour tester notre idée et commencer à l'exposer. Par ailleurs, la boîte à outils nous a permis de réaliser l'application ayant servi pour l'expérimentation contrôlée.

6.4.1 Prototypage du pointage sémantique

Le prototypage du pointage sémantique a eu plusieurs utilités. Son premier rôle fut de permettre de tester informellement notre idée. Il a permis aussi d'offrir rapidement un support à la discussion pour permettre d'exposer la notion d'espace moteur qui était nouvelle. Enfin, il a eu un rôle pédagogique plus large en servant de support aux exposés qui ont présenté la technique d'interaction.

Prototype basse fidélité

Le premier prototype réalisé, présenté sur la Figure 6.14, donne une représentation des espaces visuel (à gauche) et moteur (à droite) d'un bureau stylisé sur lequel des rectangles figurent des éléments courants comme une fenêtre d'application et un ensemble d'icones disposés sur le bureau. Nous avons choisi de représenter les deux espaces car, très tôt, nous nous sommes

```
01  <defs>
02     <!-- définition du bureau -->
03     <symbol id='desktop'>
04        <!-- fond du bureau -->
05        <rect width='500' height='300' style='fill:lightblue;' />
06
07        <!-- fenêtre -->
08        <rect x='20'  y='20'  width='250' height='180'
09              hsm:behaviour='sp' hsm-arg:scale='1.2'/>
10
11        <!-- icones -->
12        <rect x='420' y='20'  width='32'  height='32'
13              hsm:behaviour='sp' hsm-arg:scale='1.4'/>
14        <rect x='420' y='70'  ... />
15        <rect x='420' y='120' ... />
16     </symbol>
17  </defs>
18
19  <!-- espace visuel -->
20  <g style='fill:lightgray;' >
21     <g id='visual' >
22        <use xlink:href='#desktop' />
23     </g>
24     <g id='cursor'>
25        <image xlink:href='rCursor.png' />
26     </g>
27  </g>
28
29 <!-- espace moteur -->
30  <g transform='translate(500)' style='fill-opacity:0;'>
31     <g id='motor' hsm:behaviour='visual/Zoomable'>
32        <use xlink:href='#desktop' />
33     </g>
34  </g>
```

FIG. 6.15 – Spécification du prototype de bureau en SVG

rendu compte que considérer le CD ratio comme l'échelle d'un espace moteur qui n'avait pas habituellement de matérialisation était parfois difficile à expliquer sans support concret.

Sur ce prototype, les deux curseurs évoluent simultanément et l'échelle de l'espace moteur est modifiée en fonction de l'échelle associée à l'objet situé à l'emplacement du curseur. Ainsi, sur la Figure 6.14, l'espace moteur situé à gauche est contracté puisque le curseur se situe dans le vide. La spécification des éléments présents sur le bureau et de leurs échelles respectives est donnée par un fichier SVG dont la Figure 6.15 donne un extrait. Les lignes 3 à 15 définissent le bureau et son contenu grâce à un ensemble de rectangles. Ceux qui représentent la fenêtre et les icones sont dotés d'attributs leur assignant un comportement et une échelle (lignes 9 et 13). Le comportement permettra à l'application de récupérer l'échelle associée à l'objet. Le bureau est ensuite représenté au travers de deux vues distinctes qui symbolisent l'espace visuel (lignes 19 à 27) et l'espace moteur (lignes 29 à 34). Cette double représentation est rendue possible grâce à la structure de graphe dirigé sans cycle utilisée par SVG. Le bureau peut ainsi être référencé à plusieurs endroits dans l'arbre (éléments `use` des lignes 22 et 32), et sera donc représenté plusieurs fois. Le mécanisme de style de SVG permet par ailleurs d'altérer ces rendus et ainsi de les différencier en supprimant le remplissage des rectangles dans la représentation de l'espace moteur. Enfin, cette représentation est équipée du comportement standard `Zoomable` (ligne 31) ce qui permet à l'application de la manipuler facilement.

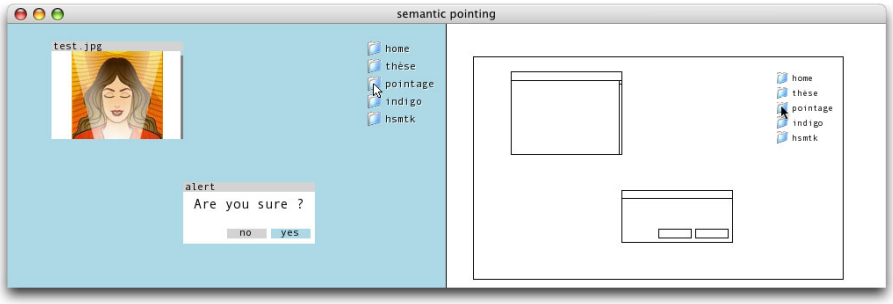


FIG. 6.16 – Prototype raffiné graphiquement du pointage sémantique

L'application elle-même comporte une cinquantaine de lignes de code (commentaires et lignes vides exclus) qui se répartissent ainsi :

- une dizaine de lignes pour initialiser et finaliser le programme (la création de la fenêtre, le chargement du SVG, l'initialisation de l'interaction) ;
- une dizaine de lignes pour définir le comportement des objets actifs pour le pointage sémantique (ce comportement se contente de récupérer l'échelle donnée dans le SVG et de la fournir quand le programme le lui demande) ; et
- une trentaine de lignes pour la machine à états qui met à jour la position du curseur, récupère l'échelle, met à jour l'affichage dès que la souris bouge.

Nous avons réalisé une application équivalente à l'aide du langage Tcl. Ce langage interprété, utilisé en conjonction avec sa boîte à outils graphique standard Tk[6], est particulièrement adapté au prototypage rapide d'application interactive. Le programme réalisé en Tcl, qui charge aussi la configuration du bureau à partir d'un fichier, comporte pour sa part une quarantaine de lignes. Les programmes sont donc d'une taille comparable, avec un léger avantage pour Tcl, mais nous allons voir maintenant jusqu'où nous permet d'aller notre programme.

Raffinement du prototype

La première force de notre prototype est que son modèle graphique est celui de SVG. La représentation du bureau peut donc être affinée et réalisée grâce à des outils dédiés à la création graphique. En éditant le document ainsi généré pour décorer les éléments SVG avec les attributs nécessaires au pointage sémantique, le même programme, sans recompilation, nous permet de tester notre technique d'interaction dans des conditions plus convaincantes (Figure 6.16). Nous sommes ainsi passé d'un prototype basse fidélité à un prototype plus convaincant en déployant très peu d'effort, et surtout sans avoir besoin de toucher au programme lui même.

Ensuite, nous avons encore amélioré ce prototype en ajoutant la possibilité de manipuler les différents éléments du bureau simulé. Cet ajout a, là encore, été pratiquement gratuit puisqu'il a suffi d'inclure des techniques d'interac-

[6] Tcl et Tk sont disponibles à l'adresse : `http://www.tcl.tk/`.

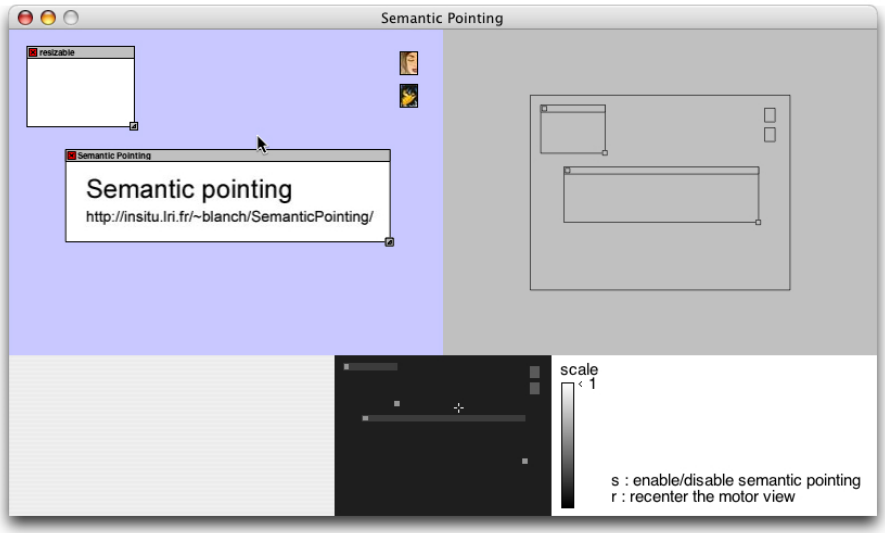


FIG. 6.17 – Prototype réalisé en Java du pointage sémantique

tion préexistantes dans la boîte à outils au sein du programme. Afin de rendre les objets du bureau manipulables, il a alors suffi d'ajouter quelques annotations au document SVG pour qu'on puisse déplacer icones et fenêtres, et pour qu'on puisse redimensionner ces dernières. Le programme comporte à ce stade une cinquantaine de lignes de code supplémentaires. Une telle évolution du prototype en Tcl/Tk n'était simplement pas envisageable, étant données les limitations de son modèle graphique et le faible support à l'interaction qu'il propose.

Nous avons aussi réalisé, à des fins de comparaison et de diffusion, un programme équivalent à l'aide du langage Java[7] (Figure 6.17). Ce programme comporte environ 520 lignes de code. Un peu moins de la moitié de ce code est utilisée pour assurer l'initialisation du programme et pour créer un modèle graphique structuré et des objets interactifs adaptés à nos besoins. Ceux-ci doivent en effet pouvoir être représentés au travers de vues multiples, à des échelles et avec des styles différents. Cette partie utilise comme point de départ le modèle graphique de bas niveau de Java : Java2D. Le reste du programme permet de gérer l'interaction, le calcul de l'échelle et la synchronisation des différentes vues. Il faut noter que finalement, même en ne prenant pas en compte la partie destinée à fournir un niveau d'abstraction graphique satisfaisant, le programme en Java reste deux fois plus long que celui du prototype réalisé avec notre boîte à outils. De plus, à la différence du programme Java, il est facile de modifier le bureau pour tester de nouvelles configurations puisqu'il suffit d'éditer le document SVG correspondant.

[7] Le prototype en Java peut être testé à l'adresse :
`http://insitu.lri.fr/~blanch/projects/SemanticPointing/demo/`.

6.4.2 Réalisation de l'expérimentation contrôlée

Nous avons aussi utilisé notre boîte à outils pour réaliser l'expérimentation contrôlée qui a servi à la validation de notre modèle du pointage sémantique. Elle nous a permis tout d'abord de réaliser simplement l'interface présentée aux participants montrée Figure 6.6. Celle-ci comporte de simples éléments SVG manipulés par le programme pour dimensionner et positionner la cible suivant une succession de tâches paramétrées par un script de configuration, et pour déplacer le curseur en fonction des mouvements de la souris sur la tablette graphique et du calcul de l'échelle de l'espace moteur courante. Deux caractéristiques de HsmTk sont particulièrement utiles pour réaliser cette expérimentation.

Accès aux événements de bas niveau

La première est la possibilité que la boîte à outils offre d'accéder aux événements de bas niveau issus des périphériques. Le dispositif de pointage utilisé pour l'expérience est le *puck* d'une tablette graphique Wacom. Il a été choisi car il est possible de connaître sa position absolue sur la table, sans que l'accélération de la souris du système n'interfère. L'application peut donc calculer elle-même l'échelle de l'espace moteur en fonction de la position du curseur, et le déplacer ainsi en fonction du CD ratio qu'elle a choisi. L'application maîtrise ainsi toute la chaîne de traitement, depuis les données brutes issues de la tablette graphique, jusqu'à la position du curseur à l'écran, ce qui est indispensable pour mettre en place une technique d'interaction comme le pointage sémantique.

Flux de contrôle de l'application

La seconde caractéristique de HsmTk mise à profit pour la réalisation de l'expérimentation est la possibilité offerte par les machines à états hiérarchiques de piloter l'application grâce à deux flux de contrôle entremêlés :

- le contrôle de la succession des tâches présentées aux participants, qui est pilotée par un script de configuration ; et
- le contrôle de chaque tâche, qui est individuellement pilotée par les actions des participants sur la souris.

Ces deux flux de contrôles sont clairement distingués grâce à la hiérarchie introduite dans les états, qui nous permet de décrire le protocole pour chaque tâche au sein d'une machine assez simple, et d'intégrer celle-ci dans une machine qui gère, elle, la succession des tâches. La Figure 6.18 montre une version schématique de cette machine. Dans son premier état (`Wait`), elle signifie à l'interpréteur qu'elle est prête pour la tâche suivante. Celui-ci peut alors invoquer l'une des transitions définies lignes 6 et 7, ce qui donne le contrôle à l'un des sous-états pour permettre l'attente (dans l'état `Sleep`) ou la présentation d'une nouvelle tâche (dans l'état `Task`). Ces différents sous-états reviennent tous dans l'état `Wait` une fois leur exécution terminée. Chaque tâche est ainsi gérée par le passage dans une succession d'états : l'attente de la présence du curseur dans la zone de départ, l'attente de son immobilité complète pendant la demi-seconde requise, le suivi du curseur avec calcul de l'échelle et enregistrement des positions et des temps précis, et enfin la détection au moment

```
01 hsm Experiment {
02    hsm Wait {
03       var Interpreter interpreter;
04       enter { interpreter.next(); }
05
06       - sleep(int ms) > Sleep(delay = ms)
07       - task(float S, float ID, float d) > Task(difficulty = ID,
08                                                distance = d,
09                                                scale = S)
10    }
11
12    hsm Sleep {
13       int delay = 0;
14       - delay > Wait
15    }
16
17    hsm Task {
18       var float difficulty = 5.;
19       var float distance   = 1024;
20       var float scale      = 1.;
21
22       enter { prepareTask(difficulty, distance, scale); }
23
24       hsm Ready {
25          - position [inStartArea(position)] > Steady
26       }
27
28       hsm Steady {
29          - position > Steady
30          - 500 > Go
31       }
32
33       hsm Go {
34          var hsm::Chrono chrono();
35
36          enter { chrono.start(); }
37
38          - position {
39             computeScale(position);
40             updateCursor();
41             log(chrono.date(), position);
42          }
43
44          - button {
45             log(chrono.stop(), passed(position));
46          } > Wait
47       }
48    }
49 }
```

FIG. 6.18 – Machine à états gérant l'expérimentation

du clic de la validité ou non de la tentative. La machine de niveau supérieur est consacrée à l'interprétation du script de configuration, et permet de paramétrer et lancer les tâches, synthétiser et enregistrer les données sur les séries de tâches qui viennent d'être effectuées, et fournir des messages informatifs et des pauses aux participants entre les multiples séries. La Figure 6.19 montre le début d'un tel script de configuration et sa syntaxe.

6.4.3 Conclusion

Les deux applications créées pour mettre au point le pointage sémantique ont permis de mettre en lumière plusieurs caractéristiques de notre boîte à outils qui l'ont rendu particulièrement adaptée à la réalisation de ces programmes de natures pourtant différentes.

```
////////////////////////////////////////////////////////////////////////////////
// File    : plan.0.0.cfg
// Content : template configuration file for dynamicCD
// History : 09-Apr-2003 - [rb] creation
////////////////////////////////////////////////////////////////////////////////

// syntax :
// n <name>    name of the subject
// r <ppp>     # pixels / tablet pitch
// s <scale>   scale in empty space
// S <scale>   scale on the target
// x <ID> <D>  Id and D of the task
// t "<text>"  text to be displayed
// w <ms>      time to wait in ms
// b           write a report for that block
// z           reset stats

// global configuration ////////////////////////////////////////////////////

n test
r 0.1025
s 1

// first block /////////////////////////////////////////////////////////////

t "Première série", w 5000
t ""

S 1

// training
x 5 1024, x 7 1024, ...
z

// real block
x 7 1024, x 7  512, ...
.
.
.

x 5  512, x 5 1024, ...
b
```

FIG. 6.19 – Script de configuration de l'expérimentation (extrait)

Une des premières caractéristiques qu'il convient de citer est la concision des programmes écrits avec la boîte à outils. Cette concision est à mettre en premier lieu au crédit de la variété des niveaux d'abstraction offerts par HsmTk. Ceux-ci permettent de manipuler des objets à un niveau de détail adapté, quelle que soit la granularité de cette manipulation. Les représentations des dispositifs d'entrée permettent ainsi d'en obtenir des visions plus ou moins détaillées selon les besoins, et ce éventuellement jusqu'au niveau le plus bas comme on l'a vu avec le programme ayant permis l'expérimentation du pointage sémantique. De même le modèle graphique peut être manipulé directement à l'aide de l'interface à SVG proposée par la bibliothèque svgl, ou à un niveau plus abstrait en associant aux fragments de SVG des comportements élémentaires préexistants ou en en développant de nouveaux. Enfin, la proposition d'un langage spécifique dédié à la programmation de comportements interactifs facilite leur écriture. Nous avons vu avec le programme de l'expérimentation, que cette structure de contrôle est aussi naturellement adaptée à la description de la logique d'une application interactive. En offrant une structure de contrôle adaptée aux interactions, les machines à états hiérarchiques suppriment les contorsions nécessaires pour

exprimer dans un langage impératif qui n'y est pas adapté les comportements interactifs. Elles permettent aussi de faire de ces interactions des objets facilement réutilisables comme le raffinement du prototype nous l'a montré.

Ce dernier point met en avant la modularité des applications réalisées avec HsmTk. Le langage y contribue comme nous l'avons vu en évitant de diffuser la logique de l'interaction au sein du code. La boîte à outils y contribue aussi par le faible couplage qu'elle impose entre les objets graphiques et les comportements interactifs. Ce couplage, assuré par les contrats structurels définis par les comportements, permet d'imposer les contraintes minimales sur les objets graphiques et ainsi de capturer uniquement les propriétés essentielles qu'ils doivent avoir. Il devient alors aisé de réutiliser les mêmes comportements pour des objets différents, ou de raffiner la représentation d'un objet sans même modifier une ligne de code. Comme le format SVG est manipulable par le biais du programme mais aussi par les outils d'édition graphique spécialisés, ce raffinement peut être réalisé directement par les designers, à l'aide de leurs propres outils. Les documents qu'ils produisent sont utilisés ensuite directement par le programme, ce qui permet d'obtenir rapidement des prototypes plus crédibles.

Chapitre 7

Conclusions et perspectives

7.1 Problématique

Nous avons proposé la boîte à outils HsmTk et ses divers niveaux d'abstraction pour faciliter la mise au point d'applications graphiques aux techniques d'interaction avancées. L'introduction de telles techniques, dites post-WIMP, est en effet une piste prometteuse pour permettre de résoudre la tension grandissante qui existe entre les quantités sans cesse croissantes de données auxquelles les ordinateurs personnels donnent accès, la diversité des dispositifs qu'ils mettent à disposition pour l'interaction, et la faible bande passante entre Hommes et machines que les techniques d'interaction traditionnelles permettent d'atteindre.

Cependant, si les interacteurs WIMP se sont imposés, ce n'est pas simplement du fait qu'ils n'avaient pas encore rencontré les limites auxquelles nous sommes aujourd'hui confrontés. Ils se sont généralisés et standardisés car ils ont été intégrés à des boîtes à outils permettant de simplifier singulièrement l'écriture d'applications interactives. Celles-ci libèrent les programmeurs d'une partie pénible de la mise au point d'un logiciel interactif : les techniques d'interaction elles-mêmes et le réglage de leurs détails.

Pour permettre aux techniques d'interaction post-WIMP de se développer, faire la preuve de leur supériorité ne suffit pas. Beaucoup de techniques performantes ont été proposées durant les dernières décennies, mais très peu sont effectivement utilisées dans des applications qui pourraient pourtant en bénéficier. Nous avons expliqué cette absence par le manque de support à leur mise en œuvre dans les outils de développement actuels et les modèles d'interaction qu'ils véhiculent. C'est donc à ces problèmes que nous nous sommes attaqué.

7.2 Notre contribution

Nous avons défini et développé une boîte à outils dédiée aux techniques d'interaction avancées pour faciliter leur utilisation. Cependant, inclure toutes les techniques d'interaction post-WIMP existantes à l'heure actuelle dans une nouvelle boîte à outils n'est pas suffisant. Nous avons vu que ces techniques tirent leur puissance du fait qu'elles ont été conçues en tenant compte du contexte de leur usage : tâche à laquelle elles sont dédiées, caractéristiques des dispositifs physiques mis en jeu, etc. Comment alors résoudre ce paradoxe et faciliter la réutilisation tout en prônant le sur mesure ?

Pour lever cette contradiction et supporter tant l'utilisation de techniques préexistantes, WIMP ou non, leur personnalisation pour des usages légèrement différents, que la création de nouvelles techniques avancées, notre boîte à outils fournit différents niveaux d'abstractions auxquels la réutilisation peut avoir lieu. Alors qu'avec la plupart des boîtes à outils existantes, la seule alternative lorsque le modèle ne convient pas est de repartir de zéro, HsmTk permet de descendre simplement d'un niveau dans le support proposé et de réutiliser ses composants moins élaborés pour bâtir de nouveaux éléments plus adaptés. La boîte à outils permet par ailleurs facilement de réutiliser ces éléments nouvellement créés, au même titre que ceux fournis originellement.

HsmTk propose donc une pyramide d'abstractions, échafaudées les unes sur les autres, et accessibles aux programmeurs. Au plus bas niveau, elle permet d'accéder aux systèmes d'exploitation et de fenêtrage s'il est besoin de descendre jusque là. Au-dessus de ce niveau, il est possible d'accéder aux détails des événements provenant des périphériques, et d'ajouter le support à des périphériques non pris en compte. À un niveau plus élevé, les périphériques sont structurés logiquement. Il en est de même pour le modèle graphique qui, au dessus du langage des cartes graphiques, propose un modèle vectoriel plus élaboré, éditable par les applications de création graphique, et manipulable par les programmes. Ces différents niveaux reposent sur une architecture à base de composants dont l'ossature est fournie par la boîte à outils.

La boîte à outils offre enfin des moyens avancés pour programmer l'interaction elle-même. En permettant tout d'abord d'associer facilement des comportements interactifs aux éléments de l'interface. En proposant surtout une extension du langage de programmation dédiée aux comportements dynamiques. Les machines à états hiérarchiques sont ajoutées au langage de programmation, et grâce à une syntaxe textuelle, leur intégration dans le cycle de développement et dans les outils usuels des programmeurs est facilitée.

Les machines à états hiérarchiques, utilisées comme structure de contrôle, et associées aux autres abstractions de la boîtes à outils, font des interactions des objets à part entière du langage. Elles peuvent ainsi être vues comme un tout, et leur logique se trouve enfin localisée au lieu d'être diffuse au sein de fonctions de rappels aux interactions difficiles à percevoir et à documenter. Leur mise au point en est ainsi facilitée. Par ailleurs, comme leurs liens avec le reste des applications devient explicite, elles sont facilement réutilisables.

7.3 Validation

Le formalisme des machines à états hiérarchiques a été choisi car sa sémantique est simple à appréhender tout en gardant un pouvoir d'expression suffisamment élevé pour décrire les interactions complexes. L'introduction de la hiérarchie permet en particulier la modularisation du code, et favorise sa structuration et l'isolation des différentes problématiques. Elle encourage ainsi la réutilisation. Par ailleurs l'adéquation du formalisme avec les interactions rend celles-ci concises à exprimer et faciles à prototyper.

Nous avons utilisé notre boîte à outils avec succès dans divers contextes. Dans le cadre du projet INDIGO, elle a montré ses capacités à supporter une architecture élaborée d'applications graphiques interactives distribuées. Elle a en particulier montré l'utilité d'exposer des fonctionnalités de bas niveau, qui ont permis l'utilisation d'un protocole de communication complètement asynchrone, ou l'ajout dynamique de code pour étendre les capacité des applications pendant leur exécution, et ce sans difficultés majeures. Durant ce projet, un cycle très itératif de mise au point a pu être mis en œuvre grâce à l'architecture de l'interaction proposée par HsmTk. Ainsi, le passage du stade de simple maquette, à celui de prototype auquel les fonctionnalités peuvent être ajoutées graduellement, pendant que l'interface est raffinée graphiquement favorise l'expérimentation de nouvelles techniques et la génération d'idées.

Quelques exemples de réalisations de techniques d'interaction montrent en effet que si les techniques d'interaction usuelles sont faciles à reproduire, les techniques post-WIMP sont tout aussi faciles à programmer. La concision du langage des machines à états hiérarchiques et leur simplicité permet ainsi de tester en quelques dizaines de lignes des techniques d'interaction avancées et d'en créer des variantes comme l'ont montré les divers exemples que nous avons présentés. Comme elles rendent facile et rapide le prototypage, et qu'elles proposent un idiome adapté à la description des comportements interactifs, elles encouragent la créativité, et permettent ainsi la création de nouvelles techniques.

La mise au point du pointage sémantique illustre ces capacités. HsmTk nous a tout d'abord permis de prototyper rapidement la technique d'interaction en couplant l'utilisation de fonctionnalités de bas niveau (le lien entre la souris et le curseur) et de haut niveau (l'attribution d'une sémantique aux objets graphiques). Là encore, le prototype a pu être facilement raffiné incrémentalement pour atteindre un niveau de réalisme satisfaisant. Il a ainsi pu servir à illustrer facilement l'idée sous-jacente du pointage sémantique : découpler la taille visuelle et motrice des objets pour adapter la première à la quantité d'information qu'ils offrent aux utilisateurs, et la seconde aux contraintes posées par l'interaction. Ce principe nous a permis de réviser les interacteurs classiques et de proposer de nouvelles versions optimisées en fonction de ces deux critères.

Par ailleurs, la validité du modèle que nous avons proposé pour expliquer les gains de performance observés a été testée grâce à une expérimentation contrôlée mise elle aussi au point en utilisant notre boîte à outils. La réalisation de cette expérimentation a bénéficié de la possibilité de contrôler facilement toute la chaîne de traitement qui part des événements de bas niveau en provenance des périphériques, et de la manipulation facile des éléments graphiques de l'interface. Enfin, les machines à états hiérarchiques nous ont per-

mis d'intégrer facilement plusieurs flux de contrôle pour l'interaction : celui de l'interaction elle-même et celui du script gérant la succession des tâches présentées aux participants à l'expérience.

Au travers de ces différentes applications, les machines à états hiérarchiques, en encodant l'état de l'application et de l'interaction à son flux de contrôle, ont donc fait la preuve de la modularité qu'elles apportent. Les différents niveaux d'abstraction proposés par la boîte à outils ont aussi permis de ne pas laisser le programmeur dans le désert de la programmation bas niveau dès que les éléments de haut niveau ne sont pas tout à fait adaptés aux besoins de l'interaction.

7.4 Perspectives

Plusieurs perspectives peuvent être ouvertes par notre travail. Tout d'abord, notre boîte à outils a montré son aptitude à prototyper et évaluer les techniques d'interaction avancées. Elle nous encourage donc à améliorer les techniques originales que nous avons introduites. Le pointage sémantique tel qu'il a été présenté ne permet pas par exemple d'éviter les problèmes posés par la présence de distracteurs sur la trajectoire d'un mouvement de pointage. En utilisant des informations extraites de la dynamique du mouvement, comme le fait sommairement l'accélération de la souris, nous pensons qu'il est possible de déterminer les différentes phases du mouvement, et d'adapter l'échelle de l'espace moteur pour ne le dilater qu'à l'approche de la fin du mouvement qui, elle seule, nécessite de la précision.

Pour ce qui est des techniques d'interaction, nous pensons aussi que notre technique de contrôle continu du zoom au clavier peut servir d'exemple pour explorer de nouveaux usages du clavier. En effet, cette technique utilise une reconnaissance de geste primitive. D'autres usages liés à l'exploitation de motifs temporels peuvent être explorés, comme par exemple décrire le motif de pointillés que l'on veut appliquer à une ligne dans un éditeur graphique en transposant ce rythme dans le domaine temporel. Dans l'interaction graphique, pour laquelle le clavier est sous-employé, nous pensons que d'autres formes d'usages de celui-ci méritent d'être explorées. Outre les motifs temporels, l'usage de motifs spatiaux peut être exploité. Utiliser des accords permettrait par exemple de réaliser des raccourcis clavier qui ne soient pas liés à une touche particulière, mais à une configuration de touches enfoncées simultanément, et ce à n'importe quel endroit du clavier, remplaçant l'apprentissage d'une lettre mnémonique par celui d'une position de doigts, présumé plus facile. Toutes ces idées peuvent être explorées facilement avec notre boîte à outils.

Concernant la boîte à outils elle-même, des évolutions moins ponctuelles sont à considérer. Elles ont trait aux limitations que nous avons constatées à notre approche. Si nous avons vanté le découplage entre la structure exacte des objets graphiques et des comportements qui peuvent leur être associés, ce découplage a l'inconvénient de rendre lâches les relations entre ces deux éléments. Dans l'état actuel, seul le test explicite permet de vérifier qu'un fragment graphique est conforme au contrat structurel défini par le comportement qui lui est associé. L'évolution de l'un ou l'autre des éléments de cette association requiert donc une vérification de la non remise en cause de ce contrat.

Cette vérification pourrait être assurée par des outils automatisés étant donné que le contrat est explicitement formulé. De même, de tels outils pourraient permettre, à partir des contrats, de générer des squelettes de documents utilisables comme structure de base à enrichir ensuite. De tels outils réduiraient certains problèmes de mise au point.

De même, nous avons volontairement minimisé les contraintes sur les machines à états hiérarchiques, en adoptant une démarche pragmatique pour la mise au point de leur sémantique. La formalisation de leur sémantique permettrait la mise au point d'outils de vérification de certaines propriétés, utiles pour la production d'un code robuste. En l'état actuel, des problèmes pourraient être simplement détectés par des analyses statiques. La vérification que tous les états peuvent être atteints permettrait de détecter certains problèmes. De même, vérifier que tous les chemins d'exécution qui commencent un protocole d'interaction le concluent effectivement est aussi envisageable. De tels outils pourraient aussi permettre de faciliter la combinaison de techniques existantes en détectant leurs incompatibilités et en proposant des mécanismes pour les résoudre (ajouter des hystérésis spatiaux ou temporels, multiplexer les comportements, etc.)

L'ajout de tels outils pour supporter la mise au point des techniques d'interaction avancées renforcerait encore l'intérêt de notre boîte à outils, et permettrait de dépasser le stade du prototypage pour bâtir effectivement des systèmes interactifs complexes. La question de savoir si notre boîte à outils "passe à l'échelle" obtiendra alors une réponse concrète.

À plus long terme, il sera intéressant d'étendre notre boîte à outils pour s'attaquer à des problématiques d'interaction plus complexes, comme par exemple le support à l'édition collecticielle de documents, et aux interfaces distribuées et multi-supports qui nécessitent une infrastructure élaborée pour mettre en œuvre des interactions distribuées.

Annexes

Annexe A

Traduction des machines à états hiérarchiques en C++

Le code des machines à états hiérarchiques est généré par un compilateur écrit en Tcl. Celui-ci construit dans une première passe un arbre syntaxique abstrait à partir des fichiers définissant les machines à états hiérarchiques. Cet arbre est traversé dans une seconde passe pour générer le code C++ correspondant. Cette génération prend un temps négligeable par rapport à la compilation du code lui-même, car la grammaire définissant le langage est très simple — elle ne nécessite pas de lire plus d'un élément en avance pour déterminer la règle courante ce qui la rend LL(1).

Le code généré utilise plusieurs mécanismes du C++ pour supporter la sémantique des machines à états hiérarchiques. En premier lieu, la hiérarchie des machines à états est traduite en une hiérarchie d'espaces de noms, ce qui permet d'utiliser leurs règles de portée pour la résolution des états lors des transitions. Ensuite, le mécanisme des modèles *(templates)* est utilisé pour agréger la mécanique de la machine fournie par la boîte à outils, les données encapsulées dans une structure permettant de leur donner les règles de visibilité décrites à la Section 4.2.1, page 75, et le code particulier à chaque état. Ce mécanisme produit du code qui permet au compilateur de générer du code efficace.

Un extrait du code généré par la machine à états hiérarchiques de la technique de zoom continu au clavier présentée à la Section 4.3.3, Figure 4.18, page 88, est donné dans les pages suivantes pour donner une indication de sa structure.

Fichier de déclaration

```
////////////////////////////////////////////////////////////////////////////
// File    : keyboardZoomer.h
//           generated from keybooardZoomer.hsm
// Content : header of c++ interactions HSM
// History : 22-JUL-2005 - [hsm2cpp.tcl] automatic creation
////////////////////////////////////////////////////////////////////////////

#if !defined(KEYBOARDZOOMER_H)
#define KEYBOARDZOOMER_H


/* part from original file ********************************************/

#include <hsm/svgl/SVGLWindow.h>
#include <hsm/svgl/SVGLWindowPicking.h>
#include <hsm/interaction/Protocol.h>
#include <hsm/component/Point.h>


/* includes ***********************************************************/

#include <hsm/component/Hsm.h>


/* KeyboardZoomer hsm definition **************************************/

namespace KeyboardZoomer {

   // Super interface
   namespace Events {
      struct Interface : public hsm::Interface {
         enum Id {
            TimeOutEvent = 0
         };
      };
   } // namespace Events

   // Data
   struct Data {
      hsm::SVGLWindow * &window;
      hsm::Point & position;
      hsm::Component *&keys;
      Data(hsm::SVGLWindow * &window_,
           hsm::Point & position_,
           hsm::Component *&keys_) :
         window(window_), position(position_), keys(keys_) {}
   };

   // Abstract Hsm
   class AHsm : public hsm::Hsm, protected virtual Data {
   protected :
      virtual void _init_();
      virtual void registerInputs();
      virtual bool handleEvent(const hsm::Event &event);
   public :
      void setKeys(hsm::Component &c);
   };

   // Interface
   template< class SuperInterface = Events::Interface >
   struct TInterface : public SuperInterface, protected virtual Data {};
   typedef TInterface< > Interface;

   // Hsm
   extern const char name[];
   template< class SuperInterface = Events::Interface >
   struct T {
      typedef hsm::impl::Hsm< name, AHsm,
                              TInterface< SuperInterface >, Data
                            > Hsm;
   };
   typedef T< >::Hsm Hsm;
```

```
/* Idle hsm definition ****************************************************/

   namespace Idle {

      // Data
      struct Data { ... };

      // Abstract Hsm
      class AHsm : public hsm::Hsm, protected virtual Data {
         ...
      };

      // Interface
      template< class SuperInterface = KeyboardZoomer::Interface >
      struct TInterface : public SuperInterface, protected virtual Data {};
      typedef TInterface< > Interface;

      // Hsm
      extern const char name[];
      template< class SuperInterface = KeyboardZoomer::Interface >
      struct T {
         typedef hsm::impl::Hsm< name, AHsm,
                                 TInterface< SuperInterface >, Data
                               > Hsm;
      };
      typedef T< >::Hsm Hsm;

   } // namespace Idle


/* Zoom hsm definition ****************************************************/

   namespace Zoom {

      ...

/* Continuous hsm definition **********************************************/

      namespace Continuous {

         ...

      } // namespace Continuous

   } // namespace Zoom

} // namespace KeyboardZoomer


#endif // !defined(KEYBOARDZOOMER_H)
```

Fichier de définition

```
////////////////////////////////////////////////////////////////////////////
// File    : keyboardZoomer.cpp
//           generated from keyboardZoomer.hsm
// Content : implementation of interactions HSM
// History : 22-JUL-2005 - [hsm2cpp.tcl] automatic creation
////////////////////////////////////////////////////////////////////////////


/* includes ****************************************************************/

#include "keyboardZoomer.h"


/* KeyboardZoomer hsm implementation ***************************************/

namespace KeyboardZoomer {

   void AHsm::_init_() {
      _initial_ = hsmMap[Idle::name]
                = new Idle::T< Interface >::Hsm(this, Idle::Data(keys));
      hsmMap[Zoom::name] =
                  new Zoom::T< Interface >::Hsm(this,
                                                Zoom::Data(window,
                                                           position,
                                                           keys));
   }

   void AHsm::registerInputs() {
      keys->addReceiver(top());
   }

   bool AHsm::handleEvent(const hsm::Event &event) {
      switch(event.message) {
      case hsm::Event::CHANGE :
         if(true
             && (event.sender == keys)
             && ((event.data.aInt < '1') || (event.data.aInt > '9'))) {
            return true;
         }
         break;
      }
      return false;
   }

   void AHsm::setKeys(hsm::Component &c) {
      keys->removeReceiver(top());
      keys = &c;
      keys->addReceiver(top());
   }

   const char name[] = "keyboardZoomer";


/* Idle hsm implementation *************************************************/

   namespace Idle {

      ...

   } // namespace Idle


/* Zoom hsm implementation *************************************************/

   namespace Zoom {

      ...
```

```
/* Continuous hsm implementation ******************************************/

      namespace Continuous {

         ...

         bool AHsm::handleEvent(const hsm::Event &event) {
            switch(event.message) {
            case hsm::Event::CHANGE :
               if(true && (event.sender == keys)) {
                  zoom->zoom((event.data.aInt - key) * 10.);
                  leaveSubHsms();
                  hsm::Hsm *next = _super_->resolve(Continuous::name);
                  Continuous::Hsm *nextHsm = dynamic_cast< Continuous::Hsm * >(next);
                  if(nextHsm != 0) {
                     nextHsm->data.key = (event.data.aInt);
                  }
                  if(next != 0) {
                     next->enterSubHsms();
                  }
                  return true;
               }
               break;
            case hsm::Event::INTERNAL :
               if(true
                   && (event.sender == this)
                   && (event.data.aInt == Interface::TimeOutEvent)) {
                  leaveSubHsms();
                  hsm::Hsm *next = _super_->resolve(Idle::name);
                  if(next != 0) {
                     next->enterSubHsms();
                  }
                  return true;
               }
               break;
            }
            return false;
         }

         ...

      } // namespace Continuous

   } // namespace Zoom

} // namespace KeyboardZoomer
```

Annexe B

Concrétisation d'un graphe conceptuel en graphe perceptuel

Nous donnons ici le programme XSLT qui permet de transformer le COG représentant un système de fichiers donné en exemple au Chapitre 5, Section 5.1.2, en POG SVG.

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:xlink='http://www.w3.org/1999/xlink'
    xmlns:hsm='http://insitu.lri.fr/hsm'
    xmlns:indigo='http://insitu.lri.fr/indigo'
    xmlns:fs='http://insitu.lri.fr/indigo/fs' >

   <xsl:output method='xml' indent='yes'
      doctype-system='http://www.w3.org/Graphics/SVG/SVG-19991203.dtd'
      doctype-public='-//W3C//DTD SVG 03December 1999//EN' />

<!-- document ------------------------------------------------------------->

   <xsl:variable name='rootPath'>/</xsl:variable>
   <xsl:template match='fetchResponse'>
      <svg xml:space='preserve' style='font-size:13; font-family:Andale Mono'>

         <!-- définitons -->
         <defs>
            <linearGradient id='bgd' gradientUnits='userSpaceOnUse'
                            x1='0' y1='0' x2='40' y2='40'>
               <stop offset='0' style='stop-color:lightgray' />
               <stop offset='1' style='stop-color:white' />
            </linearGradient>

            <symbol id='closed'>
               <path d='M 6 3 l 5 5 l -5 5 z' style='fill:gray; stroke:none' />
            </symbol>
            <symbol id='opened'>
               <path d='M 3 6 l 5 5 l 5 -5 z' style='fill:gray; stroke:none' />
            </symbol>
         </defs>

         <!-- arborescence des fichiers -->
         <g>
            <xsl:apply-templates select='fetchReturn'>
               <xsl:with-param name='parentPath'>
                  <xsl:value-of select='$rootPath'/>
               </xsl:with-param>
            </xsl:apply-templates>
         </g>
      </svg>
   </xsl:template>
```

```
<!-- répertoire ------------------------------------------------------------->

   <xsl:template match='*[@xsi:type='fs:Volume']|*[@xsi:type='fs:Folder']'>
      <xsl:param name='parentPath'/>
      <xsl:variable name='path'>
         <xsl:choose>
            <xsl:when test='$parentPath=$rootPath'></xsl:when>
            <xsl:otherwise>
               <xsl:value-of select='$parentPath'
            />/<xsl:value-of select='name()'
            />[<xsl:number value='position() - 1'/>]
            </xsl:otherwise>
         </xsl:choose>
      </xsl:variable>

      <!-- fragment SVG -->
      <g hsm:behaviour='tree' indigo:path='{$path}' id='{generate-id()}'>
         <rect width='1000' height='16' style='fill:url(#bgd)'/>
         <use x='3' xlink:href='#closed' />

         <!-- récupération du nom -->
         <xsl:apply-templates select='Name'>
            <xsl:with-param name='parentPath'>
               <xsl:value-of select='$path'/>
            </xsl:with-param>
         </xsl:apply-templates>

         <!-- traitement des fils -->
         <g transform='translate(16,16)'>
            <xsl:apply-templates
               select='*[@xsi:type='fs:Folder']|*[@xsi:type='fs:File']'>
               <xsl:with-param name='parentPath'>
                  <xsl:value-of select='$path'/>
               </xsl:with-param>
            </xsl:apply-templates>
         </g>
      </g>
   </xsl:template>


<!-- fichier ---------------------------------------------------------------->

   <xsl:template match='*[@xsi:type='fs:File']'>
      <xsl:param name='parentPath'/>
      <xsl:variable name='path'>
         <xsl:value-of select='$parentPath'
      />/<xsl:value-of select='name()'
      />[<xsl:number value='position() - 1'/>]
      </xsl:variable>

      <!-- fragment SVG -->
      <g hsm:behaviour='node' indigo:path='{$path}' id='{generate-id()}'>
         <rect width='1000' height='16' style='fill:white; opacity:0'/>

         <!-- récupération du nom -->
         <xsl:apply-templates select='Name'>
            <xsl:with-param name='parentPath'>
               <xsl:value-of select='$path'/>
            </xsl:with-param>
         </xsl:apply-templates>
      </g>
   </xsl:template>


<!-- nom de fichier --------------------------------------------------------->

   <xsl:template match='Name'>
      <xsl:param name='parentPath'/>
      <text x='22' y='12' indigo:path='{$parentPath}/Name' id='{generate-id()}'>
         <xsl:value-of select='.'/>
      </text>
   </xsl:template>

</xsl:stylesheet>
```

Annexe C

Fonction d'échelle de l'expérimentation du pointage sémantique

Le Chapitre 6 décrit le pointage sémantique. Pour simplifier l'expression analytique de la distance et de la taille de la cible dans l'espace moteur, nous avons utilisé une fonction simplifiée pour exprimer l'échelle en fonction de la position des cibles et du curseur. En pratique, cette fonction introduit une discontinuité dans le CD ratio qui peut être perçue par l'utilisateur. Pour remédier à ce défaut, nous avons utilisé une fonction plus lisse que la fonction rectangle Π donnée par l'Équation 6.1, page 113. Cette fonction Ω a un profil en cloche, et la Figure C.1 permet de comparer Π et Ω.

Tout comme Π, Ω a été choisie pour les propriétés suivantes :

- modifier la taille de la cible en fonction de l'échelle ; et
- décroître rapidement à l'extérieur de la cible pour ne pas dilater l'espace vide.

Une contrainte supplémentaire lui a été imposée : celle d'être continue.

La première contrainte se traduit par le fait que les parties grisées de la Figure C.1 ont la même aire. Cela veut dire que l'intégrale de Ω à l'intérieur

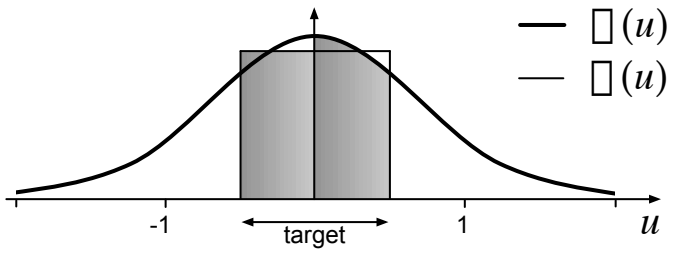


FIG. C.1 – Profil de la variation de l'échelle dans les cibles

Les surfaces grisées ont la même aire.

de la cible doit valoir 1, comme celle de Π, c'est-à-dire :

$$\int_{-1/2}^{1/2} \Omega(u)\, du = 1\,. \tag{C.1}$$

La seconde contrainte s'exprime par exemple par :

$$\int_{1/2}^{\infty} \Omega(u)\, du \leq 1\,. \tag{C.2}$$

Nous avons utilisé la fonction Ω suivante :

$$\Omega(u) = \frac{\ln(3)}{\cosh^2(\ln(3) \times u)} \tag{C.3}$$

car elle remplit les conditions précédentes et que son intégrale peut être déterminée analytiquement, ce qui nous a permis de calculer exactement les distances dans l'espace moteur.

La fonction d'échelle de l'Équation 6.2, page 113 devient alors :

$$\text{scale}(X) = \left(1 - \sum_i \Omega\left(\frac{\|X - D_i\|}{W_i}\right)\right) + \sum_i S_i \times \Omega\left(\frac{\|X - D_i\|}{W_i}\right) \tag{C.4}$$

et la relation entre id et ID qui en découle a exactement les mêmes caractéristiques qu'en l'utilisant la fonction rectangle (Figure 6.5, page 115).

Bibliographie

[Accot et Zhai, 1997] Johnny Accot et Shumin Zhai. Beyond Fitts' law : models for trajectory-based HCI tasks. In *CHI'97 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 295–302. ACM Press, 1997.

[Accot et Zhai, 2002] Johnny Accot et Shumin Zhai. More than dotting the i's — foundations for crossing-based interfaces. In *CHI '02 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 73–80, New York, NY, USA, 2002. ACM Press.

[Adobe Systems Inc., 1993] Adobe Systems Inc. *Programming the Display PostScript System with X*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[Apitz et Guimbretière, 2004] Georg Apitz et François Guimbretière. CrossY : a crossing-based drawing application. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 3–12, New York, NY, USA, 2004. ACM Press.

[Appert *et al.*, 2004] Caroline Appert, Michel Beaudouin-Lafon, et Wendy Mackay. Context matters : Evaluating interaction techniques with the CIS model. In *People and Computers XVIII - Design for Life - Proceedings of HCI 2004*, pages 279–295. Springer Verlag, 2004.

[Arch, 1992] A metamodel for the runtime architecture of an interactive system : the UIMS tool developers workshop. *ACM SIGCHI Bulletin*, 24(1) :32–37, 1992.

[Balakrishnan, 2004] Ravin Balakrishnan. "Beating" Fitts'law : virtual enhancements for pointing facilitation. *International Journal of Human-Computer Studiese*, 61 :857–874, 2004.

[Bastide *et al.*, 2002] Rémi Bastide, David Navarre, et Philippe Palanque. A model-based tool for interactive prototyping of highly interactive applications. In *Proceedings of CHI'02, extended abstracts on Human Factors in Computer Systems*, pages 516–517. ACM Press, 2002.

[Baudisch *et al.*, 2003] Patrick Baudisch, Edward Cutrell, Dan Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson, et Alex Zierlinger. Drag-and-pop and drag-and-pick : techniques for accessing remote screen content on touch- and pen-operated systems. In *Proceedings of Interact 2003*, pages 57–64, 2003.

[Beaudouin-Lafon et Lassen, 2000] Michel Beaudouin-Lafon et Henry Michael Lassen. The architecture and implementation of CPN2000, a post-WIMP graphical application. In *UIST'00 : Proceedings of the 13th annual*

ACM symposium on User interface software and technology, pages 181–190. ACM Press, 2000.

[Beaudouin-Lafon et Mackay, 2000] Michel Beaudouin-Lafon et Wendy E. Mackay. Reification, polymorphism and reuse : three principles for designing visual interfaces. In *AVI '00 : Proceedings of the working conference on Advanced visual interfaces*, pages 102–109, New York, NY, USA, 2000. ACM Press.

[Beaudouin-Lafon, 1997] Michel Beaudouin-Lafon. Interaction instrumentale : de la manipulation directe à la réalité augmentée. In *IHM'97 : Actes des neuvièmes journées francophones sur l'Interaction Homme-Machine*. Cépaduès Éditions, 1997.

[Beaudouin-Lafon, 2000] Michel Beaudouin-Lafon. Instrumental interaction : an interaction model for designing post-WIMP user interfaces. In *CHI'00 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 446–453. ACM Press, 2000.

[Beaudouin-Lafon, 2004] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *AVI'04 : Proceedings of the working conference on Advanced visual interfaces*, pages 15–22. ACM Press, 2004.

[Beaudoux et Beaudouin-Lafon, 2001] Olivier Beaudoux et Michel Beaudouin-Lafon. DPI : A conceptual model based on documents and interaction instruments. In *People and Computer XV - Interaction without frontier (Joint proceedings of HCI 2001 and IHM 2001)*, pages 247–263. Spring Verlag, 2001.

[Beaudoux et Beaudouin-Lafon, 2005] Olivier Beaudoux et Michel Beaudouin-Lafon. OpenDPI : A toolkit for developing document-centered environments. In *ICEIS 2005 : Proceedings of the 7th International Conference on Enterprise Information Systems*, volume 5, pages 39–47. ICEIS Press, 2005.

[Beaudoux, 2004] Olivier Beaudoux. *Espaces de travail interactifs et collaboratifs : Vers un modèle centré sur les documents et les instruments d'interaction*. Thèse de doctorat, LRI & ESEO, 2004.

[Bederson *et al.*, 2000] Benjamin B. Bederson, Jon Meyer, et Lance Good. Jazz : an extensible zoomable user interface graphics toolkit in Java. In *UIST'00 : Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 171–180. ACM Press, 2000.

[Bederson *et al.*, 2004] Benjamin B. Bederson, Jesse Grosjean, et Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, 30(8) :535–546, 2004.

[Bederson et Hollan, 1994] Benjamin B. Bederson et James D. Hollan. Pad++ : a zooming graphical interface for exploring alternate interface physics. In *UIST '94 : Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, New York, NY, USA, 1994. ACM Press.

[Bier *et al.*, 1993] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, et Tony D. DeRose. Toolglass and magic lenses : the see-through interface.

In *SIGGRAPH '93 : Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 73–80, New York, NY, USA, 1993. ACM Press.

[Bier *et al.*, 1994] Eric A. Bier, Maureen C. Stone, Ken Fishkin, William Buxton, et Thomas Baudel. A taxonomy of see-through tools. In *CHI '94 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 358–364, New York, NY, USA, 1994. ACM Press.

[Blanch *et al.*, 2004] Renaud Blanch, Yves Guiard, et Michel Beaudouin-Lafon. Semantic pointing : improving target acquisition with control-display ratio adaptation. In *CHI'04 : Proceedings of the 2004 conference on Human factors in computing systems*, pages 519–526. ACM Press, 2004.

[Blanch *et al.*, 2005] Renaud Blanch, Stéphane Conversy, Thomas Baudel, Yun Peng Zhao, Yannick Jestin, et Michel Beaudouin-Lafon. INDIGO : une architecture pour la conception d'applications graphiques interactives distribuées. In *Actes des dixseptièmes journées francophones sur l'Interaction Homme-Machine (IHM 2005)*, 2005.

[Blickenstorfer, 1995] C. H. Blickenstorfer. Graffiti : Wow ! *Pen Computing Magazine*, pages 30–31, january 1995.

[Buxton et Myers, 1986] W. Buxton et B. Myers. A study in two-handed input. In *CHI '86 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–326, New York, NY, USA, 1986. ACM Press.

[Callahan *et al.*, 1988] J. Callahan, D. Hopkins, M. Weiser, et B. Shneiderman. An empirical comparison of pie vs. linear menus. In *CHI'88 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 95–100. ACM Press, 1988.

[Card *et al.*, 1980] Stuart Card, Thomas P. Morn, et Allen Newell. The keystroke-level model for user performance with interactive systems. *Communications of the ACM*, 26 :396–210, 1980.

[Card *et al.*, 1983] Stuart Card, Thomas P. Morn, et Allen Newell. *The Psychology if Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[Card *et al.*, 1990] Stuart K. Card, Jock D. Mackinlay, et George G. Robertson. The design space of input devices. In *CHI'90 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 117–124, New York, NY, USA, 1990. ACM Press.

[Cardelli et Pike, 1985] Luca Cardelli et Rob Pike. Squeak : a language for communicating with mice. In *SIGGRAPH '85 : Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 199–204, New York, NY, USA, 1985. ACM Press.

[Carroll, 2003] John M. Carroll, éditeur. *HCI Models, Theories, and Frameworks, Toward a Multidisciplinary Science*. The Morgan Kaufmann Series in Interactive Technologies. Elsevier, 2003.

[Chatty *et al.*, 2004] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, et Christophe Mertz. Revisiting visual interface programming : creating GUI tools for designers and programmers. In

UIST'04 : Proceedings of the 17th annual ACM symposium on User interface software and technology, pages 267–276. ACM Press, 2004.

[Cockburn et Firth, 2003] Andy Cockburn et Andrew Firth. Improving the acquisition of small targets. In *Proc. HCI 2003*, pages 181–196, 2003.

[Collomb et Hascoët, 2004] Maxime Collomb et Mountaz Hascoët. Speed and accuracy in throwing models. In *Proceedings of the conference HCI 2004 : Design for life*, volume 2, 2004.

[Conversy et Fekete, 2002] Stéphane Conversy et Jean-Daniel Fekete. The svgl toolkit : enabling fast rendering of rich 2D graphics. Technical Report 02/1/INFO, École des Mines de Nantes, 2002.

[Coutaz, 1987] Joëlle Coutaz. PAC, an object-oriented model for dialog design. In *Proceedings of INTERACT'87 : the IFIP Conference on Human-Computer Interaction*, pages 431–436, 1987.

[Dragicevic et Fekete, 1999] Pierre Dragicevic et Jean-Daniel Fekete. Étude d'une boîte à outils multi-dispositifs. In *Actes des onzièmes journées francophones sur l'Interaction Homme-Machine (IHM 1999)*, pages 55–62, 1999.

[Dragicevic et Fekete, 2001] Pierre Dragicevic et Jean-Daniel Fekete. Input device selection and interaction configuration with ICON. In *joint proceedings of HCI 2001 and IHM 2001*, pages 543–558, 2001.

[Dragicevic, 2004] Pierre Dragicevic. *Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables*. Thèse de doctorat, Université de Nantes, 2004.

[Edwards *et al.*, 1997] W. Keith Edwards, Scott E. Hudson, Joshua Marinacci, Roy Rodenstein, Thomas Rodriguez, et Ian Smith. Systematic output modification in a 2D user interface toolkit. In *UIST '97 : Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 151–158, New York, NY, USA, 1997. ACM Press.

[Engelbart, 1998] Douglas C. Engelbart. Augment, bootstrap communities, the Web : what next ? In *CHI '98 : CHI 98 conference summary on Human factors in computing systems*, pages 15–16, New York, NY, USA, 1998. ACM Press.

[Fekete, 2004] Jean-Daniel Fekete. The InfoVis toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis'04)*, pages 167–174. IEEE Press, 2004.

[Fitts, 1954] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psycology*, 47 :381–391, 1954.

[Goldberg et Richardson, 1993] David Goldberg et Cate Richardson. Touch-typing with a stylus. In *CHI '93 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 80–87, New York, NY, USA, 1993. ACM Press.

[Goldberg et Robson, 1981] A. Goldberg et D. Robson. The Smalltalk-80 system. *Byte Magazine*, 6(8) :36–48, 1981.

[Gosling *et al.*, 1989] James Gosling, David S. H. Rosenthal, et Michele J. Arden. *The NeWS book : an introduction to the network/extensible window system*. Springer-Verlag New York, Inc., 1989.

[Green, 1986] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3) :244–275, 1986.

[Grossman et Balakrishnan, 2005] Tovi Grossman et Ravin Balakrishnan. The bubble cursor : Enhancing target acquisition by dynamic resizing of the cursor's activation area. In *CHI'05 : Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 281–290. ACM Press, 2005.

[Guiard *et al.*, 2004] Yves Guiard, Renaud Blanch, et Michel Beaudouin-Lafon. Object pointing : a complement to bitmap pointing in GUIs. In *GI '04 : Proceedings of the 2004 conference on Graphics Interface*, pages 9–16. Canadian Human-Computer Communications Society, 2004.

[Guiard, 1987] Yves Guiard. Asymmetric division of labor in human skilled bimanual action : The kinematic chain as a model. *Journal of Motor Behavior*, 19(4) :486–517, 1987.

[Guimbretière *et al.*, 2001] François Guimbretière, Maureen C. Stone, et Terry Winograd. Fluid interaction with high-resolution wall-size displays. In *UIST'01 : Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 21–30. ACM Press, 2001.

[Guimbretière et Winograd, 2000] François Guimbretière et Terry Winograd. FlowMenu : combining command, text, and data entry. In *UIST'00 : Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 213–216, New York, NY, USA, 2000. ACM Press.

[Harel, 1987] David Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[Hascoët *et al.*, 2005] Mountaz Hascoët, Maxime Collomb, et Renaud Blanch. Évolution du *drag-and-drop* : du modèle d'interaction classique aux surfaces multi-supports. *Revue Information, Interaction, Intelligence*, 4(2) :9–38, 2005.

[Hascoët, 2003] Mountaz Hascoët. Throwing models for large displays. In *HCI'2003, Designing for Society*, volume 2, pages 73–77. British HCI Group, 2003.

[Heer *et al.*, 2005] Jeffrey Heer, Stuart K. Card, et James A. Landay. prefuse : a toolkit for interactive information visualization. In *CHI '05 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM Press, 2005.

[Henry *et al.*, 1990] Tyson R. Henry, Scott E. Hudson, et Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST '90 : Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 112–122, New York, NY, USA, 1990. ACM Press.

[Hick, 1952] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, (4) :11–26, 1952.

[Hill, 1986] Ralph D. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction — the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3) :179–210, 1986.

[Hill, 1992] Ralph D. Hill. The abstraction-link-view paradigm : using constraints to connect user interfaces to applications. In *CHI '92 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 335–342, New York, NY, USA, 1992. ACM Press.

[Hoare, 1978] C. A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8) :666–677, 1978.

[Hopkins, 1991] Don. Hopkins. The design and implementation of pie menus. *Dr. Dobb's Journal*, 16(12) :16–26, 1991.

[Hudson *et al.*, 1997] Scott E. Hudson, Roy Rodenstein, et Ian Smith. Debugging lenses : a new class of transparent tools for user interface debugging. In *UIST '97 : Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 179–187, New York, NY, USA, 1997. ACM Press.

[Hudson *et al.*, 2005] Scott E. Hudson, Jennifer Mankoff, et Ian Smith. Extensible input handling in the subArctic toolkit. In *CHI '05 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 381–390, New York, NY, USA, 2005. ACM Press.

[Huot *et al.*, 2004] Stéphane Huot, Cédric Dumas, Pierre Dragicevic, Jean-Daniel Fekete, et Gérard Hégron. The MaggLite post-WIMP toolkit : draw it, connect it and run it. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 257–266. ACM Press, 2004.

[Igarashi *et al.*, 1999] Takeo Igarashi, Satoshi Matsuoka, et Hidehiko Tanaka. Teddy : a sketching interface for 3D freeform design. In *SIGGRAPH '99 : Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 409–416, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Jacob *et al.*, 1999] Robert J. K. Jacob, Leonidas Deligiannidis, et Stephen Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1) :1–46, 1999.

[Johnson *et al.*, 1989] Jeff Johnson, Teresa L. Roberts, William Verplank, David C. Smith, Charles H. Irby, Marian Beard, et Kevin Mackey. The Xerox Star : a retrospective. *Computer*, 22(9) :11–26, 28–29, 1989.

[Kabbash *et al.*, 1994] Paul Kabbash, William Buxton, et Abigail Sellen. Two-handed input in a compound task. In *CHI '94 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 417–423, New York, NY, USA, 1994. ACM Press.

[Keyson, 1997] David V. Keyson. Dynamic cursor gain and tactual feedback in the capture of cursor movements. *Ergonomics*, 12 :1287–1298, 1997.

[Krasner et Pope, 1988] G. E. Krasner et S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3) :26–49, 1988.

[Kristensson et Zhai, 2004] Per-Ola Kristensson et Shumin Zhai. Shark2 : a large vocabulary shorthand writing system for pen-based computers. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 43–52, New York, NY, USA, 2004. ACM Press.

[Kurtenbach *et al.*, 1997] Gordon Kurtenbach, George Fitzmaurice, Thomas Baudel, et Bill Buxton. The design of a GUI paradigm based on tablets, two-hands, and transparency. In *CHI '97 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 35–42, New York, NY, USA, 1997. ACM Press.

[Kurtenbach et Buxton, 1993] Gordon Kurtenbach et William Buxton. The limits of expert performance using hierarchic marking menus. In *CHI '93 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 482–487. ACM Press, 1993.

[Kurtenbach, 1993] Gordon Paul Kurtenbach. *The design and evaluation of marking menus*. Thèse de doctorat, University of Toronto, 1993.

[Lecolinet, 2003] Éric Lecolinet. A molecular architecture for creating advanced GUIs. In *UIST '03 : Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 135–144. ACM Press, 2003.

[Lécuyer *et al.*, 2000] Anatole Lécuyer, Sabine Coquillart, Abderrahmane Kheddar, Paul Richard, et Philippe Coiffet. Pseudo-haptic feedback : Can isometric input devices simulate force feedback ? In *Proc. VR 2000*, pages 83–90, 2000.

[Linton et Price, 1993] Mark Linton et Chuck Price. Building distributed user interfaces with fresco. *The X Resource*, (5) :77–87, 1993.

[MacKenzie et Riddersma, 1994] I. Scott MacKenzie et S. Riddersma. Effects of output display and control-display gain on human performance in interactive systems. *Behaviour & Information Technology*, 13 :328–337, 1994.

[MacKenzie, 1989] I. Scott MacKenzie. A note on the information-theoretic basis for Fitts' law. *Journal of Motor Behavior*, 21 :323–330, 1989.

[McGuffin et Balakrishnan, 2002] Michael McGuffin et Ravin Balakrishnan. Acquisition of expanding targets. In *CHI '02 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 57–64. ACM Press, 2002.

[Mealy, 1955] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[Mertz *et al.*, 2000] Christophe Mertz, Stéphane Chatty, et Jean-Luc Vinot. The influence of design techniques on user interfaces : the DigiStrips experiment for air traffic control. In *HCI-Aero 2000 International Conference on Human-Computer Interaction in Aeronautics*, 2000.

[Mitchell et Shneiderman, 1989] J. Mitchell et B. Shneiderman. Dynamic versus static menus : an exploratory comparison. *ACM SIGCHI Bulletin*, 20(4) :33–37, 1989.

[Moore, 1956] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, (34) :129–153, 1956.

[Myers *et al.*, 1990] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, et Philippe Marchal. Garnet : Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11), 1990.

[Myers *et al.*, 1992] Brad A. Myers, Dario A. Giuse, et Brad Vander Zanden. Declarative programming in a prototype-instance system : object-oriented programming without writing methods. In *OOPSLA '92 : conference proceedings on Object-oriented programming systems, languages, and applications*, pages 184–200, New York, NY, USA, 1992. ACM Press.

[Myers *et al.*, 1997] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, et Patrick Doane. The amulet environment : New models for effective user interface software development. *EEE Transactions on Software Engineering*, 23(6) :347–365, 1997.

[Myers et Rosson, 1992] Brad A. Myers et Mary Beth Rosson. Survey on user interface programming. In *CHI '92 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202. ACM Press, 1992.

[Myers, 1989] Brad A. Myers. Encapsulating interactive behaviors. In *CHI '89 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 319–324, New York, NY, USA, 1989. ACM Press.

[Myers, 1990] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3) :289–320, 1990.

[Myers, 1991] B. A. Myers. Separating application code from toolkits : Eliminating the spaghetti of call-backs. In *Proceedings the 4th annual ACM Symposium on User interface software and technology*, pages 211–220. ACM Press, 1991.

[Nigay et Coutaz, 1991] Laurence Nigay et Joëlle Coutaz. Building user interfaces : organizing software agents. In *Proc. ESPRIT'91 Conference*, pages 707–719, 1991.

[Nigay et Coutaz, 1995] Laurence Nigay et Joëlle Coutaz. A generic platform for addressing the multimodal challenge. In *CHI '95 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 98–105, 1995.

[Nye, 1988] Adrian Nye. *XLIB Programming Manual and Reference Manual*. O'Reilly & Associates, Inc., 1988.

[Olsen, 1990] Dan R. Olsen, Jr. Propositional production systems for dialog description. In *CHI '90 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 57–64, New York, NY, USA, 1990. ACM Press.

[Olsen, 1998] Dan R. Olsen, Jr. *Developing user interfaces*. Morgan Kaufmann Publishers Inc., 1998.

[Onizuka *et al.*, 2005] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, et Takashi Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW '05 : Proceedings of the 14th international conference on World Wide Web*, pages 671–681, New York, NY, USA, 2005. ACM Press.

[Palanque et Bastide, 1993] Philippe Palanque et Rémi Bastide. Interactive cooperative objects : an object-oriented formalism based on Petri nets for user interface design. In *Proceedings of the IEEE/System Man and Cybernetics 93*, 1993.

[Petri, 1962] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *Proceedings of the 1962 IFIP Congress*, pages 386–390, 1962.

[Pfaff, 1985] G. E. Pfaff, éditeur. *User Interface Management Systems : Proceedings of the Seeheim Workshop*. Springer-Verlag, 1985.

[Pietriga, 2005] Emmanuel Pietriga. A toolkit for addressing HCI issues in visual language environments. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005.

[Pook *et al.*, 2000] Stuart Pook, Éric Lecolinet, Guy Vaysseix, et Emmanuel Barillot. Control menus : excecution and control in a single interactor. In *Extended abstracts on Human factors in computing systems (CHI'00)*, pages 263–264. ACM Press, 2000.

[Ramos et Balakrishnan, 2003] Gonzalo Ramos et Ravin Balakrishnan. Fluid interaction techniques for the control and annotation of digital video. In *Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 105–114. ACM Press, 2003.

[Richardson *et al.*, 1998] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, et Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1) :33–38, 1998.

[Rubine, 1991] Dean Rubine. Specifying gestures by example. In *SIGGRAPH '91 : Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 329–337, New York, NY, USA, 1991. ACM Press.

[Schmucker, 1986] K. Schmucker. MacApp : an application framework. *Byte Magazine*, 11(8) :189–193, 1986.

[Shneiderman, 1983] Ben Shneiderman. Direct manipulation : a step beyond programming languages. *IEEE Computer*, 16(8) :57–69, 1983.

[Shneiderman, 1998] Ben Shneiderman. *Designing the User Interface*. Addison Wesley Longman, 1998.

[Somberg, 1987] Benjamin L. Somberg. A comparison of rule-based and positionally constant arrangements of computer menu items. In *Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 255–260. ACM Press, 1987.

[Sutherland, 2003] Ivan Edward Sutherland. Sketchpad : A man-machine graphical communication system. Technical Report UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, 2003.

[van Dam, 1997] Andries van Dam. Post-WIMP user interfaces. *Communications of the ACM*, 40(2) :63–67, 1997.

[Wegner, 1997] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5) :80–91, 1997.

[Wellner, 1989] P. D. Wellner. Statemaster : A uims based on statechart for prototyping and target implementation. In *CHI '89 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 177–182, New York, NY, USA, 1989. ACM Press.

[Wobbrock *et al.*, 2003] Jacob O. Wobbrock, Brad A. Myers, et John A. Kembel. Edgewrite : a stylus-based text entry method designed for high accuracy and stability of motion. In *UIST '03 : Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 61–70, New York, NY, USA, 2003. ACM Press.

[Worden *et al.*, 1997] Aileen Worden, Nef Walker, Krishna Bharat, et Scott Hudson. Making computers easier for older adults to use : area cursors and sticky icons. In *CHI '97 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 266–271. ACM Press, 1997.

[Zhai *et al.*, 2000] Shumin Zhai, Michael Hunter, et Barton A. Smith. The metropolis keyboard - an exploration of quantitative techniques for virtual keyboard design. In *UIST '00 : Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 119–128, New York, NY, USA, 2000. ACM Press.

[Zhai *et al.*, 2003] Shumin Zhai, Stéphane Conversy, Michel Beaudouin-Lafon, et Yves Guiard. Human on-line response to target expansion. In *CHI '03 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 177–184. ACM Press, 2003.

[Zhai et Kristensson, 2003] Shumin Zhai et Per-Ola Kristensson. Shorthand writing on stylus keyboard. In *CHI '03 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 97–104, New York, NY, USA, 2003. ACM Press.

[Zhao *et al.*, 2002] Yunpeng Zhao, Thomas Baudel, et Jie Zhou. Objets graphiques transactionnels : une méthode ouverte pour la création d'applications interactives distribuées synchrones. In *IHM '02 : Proceedings of the 14th French-speaking conference on Human-computer interaction (Conférence Francophone sur l'Interaction Homme-Machine)*, pages 183–190, New York, NY, USA, 2002. ACM Press.

[Zhao et Balakrishnan, 2004] Shengdong Zhao et Ravin Balakrishnan. Simple vs. compound mark hierarchical marking menus. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 33–42, New York, NY, USA, 2004. ACM Press.

Architecture logicielle et outils pour les interfaces homme-machine graphiques avancées

Dans cette thèse nous proposons une approche et des outils pour faciliter la mise au point et l'utilisation de techniques d'interaction avancées au sein d'applications graphiques interactives. Nous proposons de résoudre les exigences antithétiques de la réutilisation, nécessaire à la factorisation des efforts, et de l'innovation, nécessaire à l'adaptation à de nouveaux contextes, en fournissant une pyramide d'abstractions de divers niveaux permettant leur recombinaison pour s'adapter finement aux besoins spécifiques à chaque usage. Nous proposons également d'intégrer aux langages impératifs une structure de contrôle basée sur un formalisme de machines à états hiérarchiques pour faciliter la programmation de comportements dynamiques et faire des interactions des objets à part entière du vocabulaire des programmeurs.

Nous montrons par des exemples comme ces éléments permettent la reproduction de l'état de l'art des interactions, tant standards qu'avancées, et la mise au point de techniques d'interaction originales et performantes. Nous présentons en particulier la réalisation d'applications graphiques interactives utilisant une architecture distribuée permettant de localiser l'interaction sur le système local et de reporter le noyau fonctionnel sur une machine distante. Nous présentons enfin une technique d'interaction avancée, le pointage sémantique, qui facilite la tâche élémentaire de sélection par pointage en permettant d'utiliser deux tailles pour les objets de l'interface, l'une choisie en fonction des informations qu'ils présentent, l'autre en fonction de leur importance pour la manipulation.

Software architecture and tools for advanced computer-human graphic interaction

This thesis presents an approach and a set of tools that facilitate the development and use of advanced interaction techniques in interactive graphical applications. We solve the contradictory constraints of reusability, required for factoring, and innovation, required for adapting applications to new contexts of use, by providing a pyramid of levels of abstractions that can be combined in various ways to adapt to the specific needs of each application. We also augment imperative programming languages with a new control structured based on hierarchical state machines. This facilitates the programming of dynamic behaviours by turning interactions into first-class objects of the programming language.

Through a set of examples, we demonstrate how this approach supports the implementation of both classical and state-of-the-art interactions, as well as the implementation of novel interaction techniques. In particular, we describe the implementation of a distributed architecture for developing interactive graphical applications where advanced interaction and rendering is handled on the local machine while the functional core runs on a distant machine. We also describe a novel interaction technique called semantic pointing that facilitates the selection of objects with a pointing device by decoupling the visual size of objects, defined by their presentation requirements, from their size in the motor space, defined by their interaction requirements.

discipline : **informatique**, spécialité : **interaction homme-machine**
mots clés : **architecture logicielle, boîte à outils, interaction graphique avancée, pointage sémantique.**

thèse préparée au **Laboratoire de Recherche en Informatique**,
bâtiment 490, Université Paris-Sud, 91405 Orsay cedex, France.