Orsay

nº d'ordre : 2023

UNIVERSITE DE PARIS-SUD

CENTRE D'ORSAY

THESE

presentee pour obtenir

le grade de Docteur en Science

par

Stephane Chatty

Sujet : La construction d'interfaces Homme-machine animees

soutenue le 25 mars 1992 devant la Commission d'examen composee de

M.	Jean-Pierre JOUANNAUD	President
Mme	Joelle COUTAZ	Rapporteur
M.	Henry LIEBERMAN	Rapporteur
M.	Michel BEAUDOUIN-LAFON	
M.	Michel BIDOIT	
M.	Jean-Marc GAROT	
M.	Jean-Marie HULLOT	

Resume

Cette these porte sur la construction d'interfaces homme-machine animees. A l'heure actuelle, les interfaces graphiques font l'objet de nombreux travaux en ce qui concerne leurs techniques, leur architecture, et les outils pour les mettre en uvre. Le son et la video y sont parfois introduits. Mais l'animation y est rarement utilisee, et quand elle l'est, c'est de maniere marginale. Seuls les systemes d'animation d'algorithmes l'utilisent explicitement, mais ce sont des applications tres speciques, qui sont plus graphiques que vraiment interactives.

L'introduction d'animation dans les interfaces passe par l'etude de nombreux aspects. Quelles utilisations pour l'animation ? Quelles techniques pour concilier animation et interactivite ? Quels formalismes, quels modeles pour decrire des interfaces animees ? Quels outils pour les realiser ?

Whizz est une boîte a outils pour realiser des interfaces animees. Elle est construite comme une extension d'X_{TV}, une boîte a outils pour interfaces a manipulation directe. Le modele propose par Whizz repose sur la notion d'objets communiquant entre eux par des ots de donnees et des evenements. Certains de ces objets provoquent le mouvement, d'autres permettent de le decrire, et enn les objets graphiques utilisent ces informations pour leur evolution. Ce modele permet ainsi de decrire de maniere homogene l'ensemble du comportement dynamique d'une interface, qu'il soit provoque par le temps, l'utilisateur, ou l'evolution des donnees representees. Par ailleurs, ses deux modes de transmission de l'information permettent de rendre compte des phenomenes continus ou ponctuels. Ils offrent aussi un moyen de structurer la communication a l'interieur d'une application interactive.

La principale application de Whizz a ce jour est Witness, un outil de mise au point de programmes. Witness, qui par ailleurs possede les mêmes caracteristiques que les outils de mise au point habituels, permet la representation animee des donnees d'un programme. Avec Witness, un programmeur peut mettre au point un programme grâce a des representations graphiques qu'il a lui-même construites avec un outil de dessin.

Abstract

This thesis concerns the construction of animated user interfaces. Many different aspects of graphical interfaces are presently studied: graphical techniques, software architecture, and tools for building them. Sound and video are sometimes added to such interfaces, but animation is rarely used, and only as a secondary feature. Only algorithm animation systems use it intensively, but they are very specic applications, and are not highly interactive.

Adding animation to user interfaces raises several questions. What are the applications of animation? Which techniques should be used to enable animation and interaction simultaneously? Which formalisms or models could describe animated interfaces? Which tools are required to build them?

Whizz is a toolkit for building animated user interfaces. It is an extension of X_{TV}, a user interface toolkit designed for direct manipulation interfaces. The model introduced with Whizz is based on objects which communicate through data streams and events. Some objects initiate movement by spontaneously emitting information. Other objects are used to describe movement by altering that information. Finally, graphical objects use that information to drive their evolution. This model allows a homogeneous description of the whole dynamic behaviour of an interface, be it driven by time, the user's actions, or the evolution of application data. Moreover, the existence of two ways of transferring information allow the description of continuous phenomena as well as isolated ones. They also offer a means of structuring communications within an interactive application.

The main application of Whizz to date is a debugger named Witness. In addition to traditional debugging capabilities, Witness allows the animated representation of program data. Using Witness, programmers can debug a program with their own graphical representations, built with a direct manipulation drawing tool.

Remerciements

Mes travaux de recherche se sont deroules au Laboratoire de Recherche en Informatique de l'Universite de Paris-Sud, dans le groupe dirige par Michel Beaudouin-Lafon. Michel, outre les idees de depart de cette these, m'a offert un environnement de tres haut niveau, aussi bien materiel que scientique. J'ai beaucoup appris pendant ces annees passees ensemble, et je l'en remercie.

Cette these doit aussi l'existence aux chefs du Centre d'Etudes de la Navigation Aerienne, auquel j'ai appartenu pendant toute la duree de mes travaux. Jean-Marc Garot et Dominique Colin de Verdiere m'ont fait conance durant tout ce temps, et je leur en suis reconnaissant.

Michel Bidoit a ete pour moi un directeur de these admirable. Il m'a laisse entierement libre quand tout allait bien, et a ete tres present dans les moments difciles. Je l'en remercie chaleureusement.

Jean-Pierre Jouannaud me fait l'honneur de presider ce jury, et je lui en exprime ici ma gratitude. Joelle Coutaz et Henry Lieberman ont accepte de participer a ce jury en tant que rapporteurs, et leurs remarques ont contribue a augmenter la qualite de cette these. Je les en remercie. Jean-Marie Hullot a lui aussi accepte de participer a ce jury, et j'apprecie son interêt pour mes travaux.

Cette these est le resultat de plus de trois ans passes pour la plus grande partie au LRI, et a une moindre mesure au CENA. J'ai trouve dans les deux cas un environnement particulierement stimulant et agreable, et je tiens a saluer Thomas Baudel, Francois-Regis Colin, Eric Cournarie, Patrick Dujardin, Jean-Daniel Fekete, Fabrice Mourlin et tous les autres pour leur gentillesse.

Yves Berteaud, Jean Chassaing et Patrick Lecoanet ont pousse la patience jusqu'a relire les premieres versions de cette these, et je les en remercie.

Enn, si j'ai pu mener cette these a bien, c'est aussi grâce a la patience de Claudine, et a la gentillesse de Cecile et Luc.

Table des matieres

Chapitre **1.**

Introduction

On designe sous le nom d'*interfaces homme-machine* ou *interfaces utilisateurs* tout ce qui permet aux utilisateurs d'interagir avec un ordinateur. Cela recouvre bien sûr le materiel, mais surtout le logiciel qui l'exploite. Leur etude ressort de domaines aussi varies que la psychologie cognitive (comment realiser des interfaces agreables et efcaces) ou le genie logiciel (comment realiser des interfaces plus facilement).

Au debut des annees 1990, le domaine des interfaces homme-machine est un domaine particulierement sensible. Le succes commercial du Macintosh, lance par Apple en 1983, a appris aux constructeurs informatiques l'importance des enjeux commerciaux lies aux interfaces. Les interfaces et les outils pour les fabriquer sont donc devenus des arguments de vente. Par ailleurs, l'activite de standardisation du logiciel est intense, et les interfaces n'echappent pas a cette regle : il existe par exemple un projet de standard ISO sur les menus. Ces facteurs conjugues risquent d'accrediter une idee fausse : l'idee que les interfaces homme-machine sont bien connues et leur technique bien maîtrisee.

Tout d'abord, le domaine des interfaces n'est que partiellement explore. De nouveaux types d'interfaces apparaissent regulierement. Ils mettront un certain temps avant d'être assimiles. Par exemple, on commence a peine a connaître les problemes lies aux interfaces multi-utilisateurs. Les interfaces multi-modales et les \realites virtuelles" sont encore connees a un nombre restreint de laboratoires. Il n'est même pas besoin de faire appel a des technologies de pointe pour penetrer dans l'inconnu. Ainsi, en ajoutant l'animation a nos interfaces graphiques si familieres, on peut decouvrir des modes d'interaction inexplores.

Ensuite, les techniques de realisation d'interfaces sont toujours mal maîtrisees. C'est normal pour les domaines nouveaux. Mais c'est encore vrai aussi pour les domaines explores depuis longtemps. Malgre le nombre de boîtes a outils et de generateurs d'interfaces disponibles, realiser une simple interface graphique mono-utilisateur est encore souvent un travail long et difcile. En effet, les outils disponibles offrent surtout des solutions pour ce qu'on retrouve dans toutes les applications : des menus,

des boutons, des boîtes de dialogue. Pour ce qui concerne la representation et la manipulation des donnees speciques a chaque application, il reste beaucoup a faire.

Cette these porte sur l'introduction de l'animation dans les interfaces, et sur les outils pour realiser des interfaces animees. Elle est organisee autour d'un l conducteur : la volonte de realiser un outil de mise au point de programmes permettant la visualisation graphique des donnees, de leur evolution, et des operations effectuees sur elles. Nous verrons que la realisation d'un tel outil fait appel a de nombreux domaines de l'ingenierie des interfaces. En particulier, cela illustre les deux points evoques plus haut. D'une part, introduire des animations ouvre la voie a de nouveaux modes d'interaction. Et d'autre part, l'animation permet de jeter un regard nouveau sur l'architecture des interfaces et des applications interactives, et sur les outils associes. On peut considerer que cette these aborde trois grands themes : l'animation et ses techniques, l'ingenierie des interfaces, et l'animation de programmes.

1.1 La construction d'interfaces

C'est maintenant un fait reconnu : les applications interactives sont difciles et coûteuses a realiser, a entretenir et a faire evoluer. Pour cette raison, de nombreuses recherches portent sur les techniques et les outils qui simplieraient ces tâches. On assiste ainsi a des efforts pour denir l'architecture des applications interactives, de maniere a permettre une separation claire entre la partie interactive et la partie fonctionnelle. En delimitant les relations entre ces deux parties, on simplie la structure des applications, et on permet la reutilisation de logiciel. Le but ultime de ces travaux est la realisation d'environnements de programmation pour les applications interactives : les *User Interface Management Systems* (UIMS). Plus modestement, d'autres travaux portent sur les techniques de realisation des interfaces elle-mêmes. Ainsi, l'apparition des boîtes a outils, qui offrent des techniques et des objets de base pour l'interaction avec l'utilisateur, a permis la realisation d'outils de creation interactive d'interfaces. Ces outils sont connus sous les noms d'editeurs de presentation ou de generateurs d'interfaces.

Aujourd'hui, les generateurs d'interfaces deviennent monnaie courante. Avec le marche commercial ouvert par le systeme de fenêtrage X et les environnements qui en derivent, tels *OpenLook* et *Motif*, on voit proliferer les \editeurs de widgets" et les \langages de description d'interfaces". Mais ces outils ne permettent de creer que des interfaces a base de menus, de boîtes de dialogue et de boutons, c'est-a-dire des interfaces de commande. Ces interfaces de commande, si utiles et importantes soient-elles, ne couvrent pas tout le champ des interfaces. Elles deviennent vite lourdes a utiliser lorsqu'on veut manipuler des donnees.

Dans ce dernier cas, une interface a *manipulation directe*, permettant d'agir sur les representations graphiques des donnees, est souvent plus adaptee. Mais ces interfaces sont encore relativement negligees, et peu d'outils fournissent de l'aide pour les construire. Les raisons a cela sont multiples. Tout d'abord, autant les interacteurs comme les boîtes de dialogue ou les menus sont bien connus, autant les representations de donnees peuvent être variees. Il est donc difcile d'identier les objets de base qui doivent être fournis pour construire de telles interfaces. De la même maniere, les techniques d'interaction sur ces representations sont elles-aussi mal denies. En particulier, les notions de dialogue qu'on trouve dans les interfaces de commande ne s'appliquent pas. Enn, on est toujours a la recherche d'architectures adaptees a la creation d'applications a manipulation directe. Tant que des modeles d'architecture clairs et precis n'auront pas vu le jour, on ne pourra pas disposer d'outils. A l'heure actuelle, les interfaces a manipulation directe sont donc realisees \a la main".

1.2 L'animation

L'animation consiste a faire evoluer un dessin au cours du temps. Elle trouve diverses applications dans les interfaces. On l'utilise par exemple comme transition entre deux etats, pour informer l'utilisateur. On utilise aussi des representations animees telles que sabliers, montres ou thermometres pour visualiser des processus en cours ou simplement pour faire patienter l'utilisateur. On l'utilise encore pour representer des donnees qui varient, sous la forme de cadrans par exemple. Cependant, il n'existe pas d'outils pour realiser de telles animations. En consequence, l'animation n'est employee que de maniere marginale, et est realisee a chaque fois avec des moyens de fortune.

Cette absence d'outils constitue un obstacle a l'etude de l'animation, tout comme l'absence de materiels adequats et d'outils logiciels a retarde le developpement des interfaces. En effet, sans outils il est difcile de realiser des animations, et donc de verier leur utilite. L'etude de leur interêt passe donc par la realisation d'outils. On peut cependant constater que le seul domaine ou l'animation est utilisee de maniere raisonnee, a savoir l'animation d'algorithme, semble tres fertile.

1.3 L'animation d'algorithmes

L'animation de programmes ou d'algorithmes repond a des besoins tres speciques. Les programmes informatiques sont souvent difciles a comprendre dans leur forme textuelle. Un programme presente une vue tres deformee d'un algorithme. En effet, pour ecrire un programme, on commence par enlever tout ce qui ne releve pas

directement de la resolution du probleme. Ces informations de contexte seront au mieux releguees dans des commentaires difcilement lisibles, et au pire, elles disparaîtront totalement. Ensuite, on ajoute toutes les informations inutiles a la comprehension, mais imposees par la syntaxe du langage de programmation. L'abstraction initiale, apres avoir ete amputee, est noyee dans un \bruit" qui nuit a sa comprehension.

En fait, un programme est avant tout destine a être compris par une machine. En permettant une presentation graphique en plus de la version textuelle du programme, on s'adresse a l'homme. En effet, le programmeur a sa propre representation mentale des notions qu'il introduit dans son programme, et cette representation a souvent une forme graphique. Il est d'ailleurs frequent de faire preceder la phase de programmation d'un dessin sur une feuille de papier.

Par ailleurs, a partir du texte d'un programme, il est impossible d'imaginer le comportement de ce programme lors de son execution. En effet, même si l'on comprend l'algorithme mis en uvre, l'execution de l'algorithme va dependre des donnees qui lui sont fournies. En general, le nombre des possibilites est tel qu'il est impossible de les envisager toutes, et que l'execution reelle est le seul moyen de comprendre ce qui se passe. Pour cela, il faut offrir une representation animee de l'execution du programme, montrant autant l'evolution des donnees que la progression de l'algorithme lui-même.

Les preoccupations qui motivent les recherches sur l'animation d'algorithmes ne sont pas les mêmes que pour les interfaces en general. Un systeme d'animation d'algorithmes est une application specique, dont le but principal est de presenter des informations. L'interaction n'y est pas primordiale, et les problemes de genie logiciel y sont moins cruciaux. Neanmoins, on retrouve vite des problemes communs, en particulier avec les interfaces a manipulation directe. Dans les deux cas il faut construire des representations pour des donnees. Il faut aussi formaliser les liens entre ces representations et les donnees du programme a animer, ou du noyau fonctionnel a interfacer.

1.4 Objectifs de cette these

Cette these est consacree a la construction d'interfaces animees. Elle tente de montrer que l'animation peut être consideree comme une composante naturelle d'une application interactive, parfaitement integree avec les mecanismes d'interaction et de representation visuelle de donnees. Elle propose pour cela un modele unicateur, qui permet de realiser des interfaces a manipulation directe animees, tout autant que des representations de donnees pour un systeme d'animation de programmes. Ce faisant, cette these participe au rapprochement entre le domaine de la construction d'interfaces et celui de l'animation d'algorithmes.

Le modele que nous presentons permet de decrire une interface comme un ensemble d'objets representant des donnees, et dont on peut denir le comportement. Il donne de ce comportement une description uniforme, qu'il s'agisse d'animer un objet d'un mouvement propre, de decrire ses reactions aux actions de l'utilisateur, ou sa prise en compte des operations sur les donnees qu'il represente. Ce modele est mis en uvre dans Whizz, une boîte a outils pour la construction d'interfaces animees. Diverses applications de Whizz sont envisagees. En particulier, l'un des exemples montre comment on peut melanger manipulation directe et animation pour obtenir un nouveau mode d'interaction ou les objets manipules prennent vie.

Par ailleurs, cette these montre comment Whizz peut être utilise pour construire des representations animees de donnees, dans le cadre d'un systeme d'animation de programmes. L'outil de mise au point graphique Witness y est decrit, ainsi que la maniere dont Whizz etablit une architecture claire en speciant les communications entre les donnees et leur representation. Le choix d'un outil de mise au point de programmes se justie par la similarite qui existe entre representer les donnees d'un programme en cours de mise au point, et ajouter une interface a un noyau fonctionnel. Par ailleurs, choisir comme application un outil de mise au point permet l'experimentation immediate par des utilisateurs, dans un milieu de programmeurs. L'experimentation de Witness a d'ailleurs permis, en marge des etudes sur l'animation, de montrer l'interêt d'une interface iconique pour manipuler un outil de mise au point. Witness permet ainsi de valider Whizz, tout en etant l'un des premiers outils de mise au point entierement graphiques.

1.5 Organisation

Cette these est organisee selon l'ordre logique des differents outils logiciels qui y sont presentes. La demarche est celle qui mene a la construction d'un outil de mise au point graphique de programmes. Tout d'abord, le chapitre 2 est consacre a une presentation generale des interfaces homme-machine. Nous y aborderons les techniques de dessin et d'interaction, les diverses approches de modelisation des applications interactives, et les outils pour realiser des interfaces. Le chapitre 3 est consacre a la boîte a outils X_{TV}, destinee a la programmation d'interfaces a manipulation directe. X_{TV} sert de base aux travaux presentes par la suite.

Ensuite, nous nous interesserons a l'animation dans les interfaces. Au chapitre 4, nous etudierons les applications possibles de l'animation et les problemes que pose son integration dans les interfaces. Pour cela, nous examinerons des systemes d'animation existants, mais aussi des systemes pour la programmation temps-reel, et des systemes de synthese musicale. Cette etude nous amenera a considerer les animations comme des cas particuliers de comportements dynamiques d'une interface. Au chapitre 5, nous

nous interesserons a Whizz, une boîte a outils pour la denition de ce comportement dynamique. Nous decrirons le modele mis en uvre par Whizz, puis les services qu'il offre, pour terminer sur les problemes les plus interessants poses par sa realisation.

Enn, les deux derniers chapitres sont consacres a l'animation de programmes. Le chapitre 6 presente les techniques et les outils existants pour l'animation d'algorithmes et de programmes. Le chapitre 7, quant a lui, decrit Witness, l'outil de mise au point graphique developpe grâce a Whizz. Nous examinerons son interface de commande, qui utilise l'interaction iconique. Puis nous etudierons son interface de presentation des donnees, qui repose sur un serveur d'animations bâti autour de Whizz.

Chapitre **2.**

Interfaces : techniques, modeles et outils

Dans ce chapitre, nous allons examiner l'etat de l'art dans le domaine des interfaces, an de preciser le contexte dans lequel cette these se situe. Pour cela, il faut d'abord s'interesser a l'environnement technique : quelles interfaces sur quels ordinateurs, pour quels types d'interaction. Ensuite, viennent les considerations logicielles. Celles-ci prennent deux formes. D'une part, on trouve les modeles, qui tentent de formaliser la construction des applications interactives. Le but est d'offrir des bases de travail pour realiser des interfaces plus facilement, et de maniere adequate. D'autre part, on trouve toute une gamme d'outils, qui encouragent ou non l'utilisation d'un modele particulier. Ces outils vont de la simple bibliotheque graphique aux environnements de conception d'interfaces.

2.1 Aspects techniques

2.1.1 Materiel

Pendant quelque temps, l'interaction entre l'ordinateur et l'utilisateur s'est faite grâce a un clavier et un ecran afchant des caracteres. C'est encore le cas de nombreux systemes. En particulier, les services Teletel ont assez recemment popularise cette approche. Ces mecanismes, bien que frustes, permettent theoriquement la plupart des styles de dialogue que nous connaissons aujourd'hui. La difference avec les interfaces graphiques est surtout quantitative : les dessins obtenus avec des caracteres semi-graphiques sont moins precis que ceux realises sur un ecran graphique, la vitesse de transmission des informations entre un terminal et l'unite centrale est limitee, et il est plus rapide de deplacer une souris que d'appuyer repetitivement sur une touche pour deplacer un curseur. Cependant, c'est cette simple difference quantitative qui a limite l'imagination quant a l'utilisation des terminaux alphanumeriques. Les interfaces modernes, si elles sont realisables sur ces terminaux, ont ete inventees grâce aux

stations graphiques. Pour cette raison, nous n'evoquerons desormais que les systemes et les interfaces graphiques.

La base des interfaces modernes est donc l'ecran graphique. Les micro-ordinateurs et les stations de travail utilisent des ecrans adressables point par point. L'interaction entre l'utilisateur et la machine se fait pour la plus grande partie a travers cet ecran. D'un côte, les programmes y dessinent ou y ecrivent des textes, dont les caracteres sont eux-mêmes des dessins. De l'autre côte, l'utilisateur a l'illusion d'intervenir sur le contenu de l'ecran. En effet, la plupart de ses actions sont immediatement repercutees de maniere visible, que ce soit par des caracteres qui apparaissent, par un curseur qui se deplace, ou par d'autres modications. L'ecran est donc la base de l'interaction, et les outils logiciels pour dessiner ont une position centrale dans les interfaces.

Les peripheriques d'entree sont les dispositifs qui permettent a l'utilisateur d'agir sur l'ordinateur. Ils sont de plus en plus nombreux et differents, et on en trouvera de multiples exemples dans [Baecker & Buxton 87]. On retrouve cependant deux constantes sur la plupart des ordinateurs. D'une part, un clavier est presque toujours present. Les ordinateurs servent surtout a manipuler des informations textuelles, et le clavier est encore le meilleur moyen de les transmettre. Il est fort possible que ce rôle soit un jour en partie devolu aux systemes de reconnaissance de la voix ou de l'ecriture. Ainsi, la commercialisation d'ordinateurs portables sans clavier, et utilisant la reconnaissance de l'ecriture, a commence a la n de 1991. Cependant, le clavier restera utile dans de nombreux cas, ne serait-ce que pour certains handicapes. Des claviers sont aussi utilises pour realiser rapidement des operations connues a l'avance : a chaque touche correspond une fonction. La deuxieme constante est un dispositif pour transmettre des informations geometriques, qui est utilise pour deux tâches : l'entree de donnees (un dessin par exemple), et la designation d'objets a l'ecran. Cette derniere tâche est signicative de l'apport des interfaces graphiques. Les moyens d'entree sont desormais utilises pour manipuler des donnees deja presentes, et visualisees sur l'ecran. Nous reviendrons plus loin sur cette notion de manipulation directe. Le peripherique le plus repandu pour la designation est la souris. On rencontre d'autres peripheriques donnant des positions en deux dimensions : la boule, le joystick, les tablettes graphiques.

On trouve de nombreux autres moyens d'entree, dont l'utilisation est plus specique, ou reste parfois a determiner. C'est le cas des gants numeriques qui donnent la position de la main et l'angle des articulations des doigts. C'est aussi le cas des dispositifs de detection de position, que l'on place sur des gants ou des casques. On rencontre aussi des dispositifs de suivi de l'il ; d'autres suivent les mouvements du corps. Ces materiels plus ou moins exotiques constituent une part importante de systemes dits de *realite virtuelle*. Ils vont souvent de pair avec des ecrans graphiques miniaturises et places pres de l'il, de sorte que l'utilisateur a l'impression d'être plonge dans l'interface. Certains de ces dispositifs ont un autre aspect remarquable, qui aura sans

doute des applications plus larges que les realites virtuelles. Les dispositifs de suivi de l'il ou de detection de position permettent en effet un transfert d'information sans action volontaire de l'utilisateur. Jusqu'a present, les systemes informatiques ont tenu compte des *actions* de l'utilisateur. Ils auront un jour a prendre en compte son *etat*. C'est de cette maniere que les ordinateurs pourront s'integrer discretement a notre quotidien [Weiser 91].

2.1.2 *Formes d'interaction*

Les divers materiels que nous venons d'evoquer sont utilises pour realiser des programmes avec lesquels l'utilisateur interagit. Du point de vue de l'utilisateur, ou encore d'un observateur exterieur, cette interaction peut prendre plusieurs formes, selon la maniere dont se succedent et s'enchevetrent les actions de l'homme et celles de la machine. Ces differentes formes d'interaction mettent en uvre des techniques logicielles tres differentes.

Conversation

La premiere technique d'interaction est directement issue des terminaux alphanumeriques. Il s'agit de la conversation, dont le principe de base est l'alternance : l'un agit, puis l'autre, comme dans une conversation telephonique. Selon les circonstances, l'utilisateur repond aux questions de la machine (pour entrer des donnees), ou emet des commandes auxquelles la machine repond. Dans les deux cas, l'interaction est lineaire : elle se deroule le long d'un seul \l" de conversation. De plus, elle est composee d'actions isolees, qui ne se chevauchent pas entre elles. Ces deux caracteristiques reunies font que les conversations sont assez aisees a realiser et a manipuler. De plus, il est facile de leur adjoindre des services tels que l'historique des dernieres commandes.

On rencontre surtout l'interaction par conversation dans les interfaces textuelles, et en particulier les langages de commande. C'est le mode d'interaction privilegie avec des systemes d'exploitation comme Unix, VMS ou MS-DOS. Dans ce cas, les actions de l'utilisateur sont des suites de caracteres terminees par un retour-chariot. Etonnamment, la conversation est aussi utilisee par des systemes destines a des non-specialistes, comme celui dont disposent les agences de voyage pour reserver des places d'avion. On peut aussi considerer que certaines interfaces graphiques utilisent le mode conversationnel. Ainsi, une application pourra ouvrir une boîte de dialogue pour saisir un nombre ou un texte, et interdire toute autre interaction tant que cette tâche n'a pas ete realisee. La sequence d'afchage de la boîte de dialogue, saisie, puis disparition de la boîte, est peu differente de la même sequence dans une interface textuelle.

Formulaires

L'interaction par formulaire est elle aussi issue des terminaux alphanumeriques. Elle est utilisee pour des tâches bien denies a l'avance, telles que la saisie de donnees. On la rencontre par exemple sur les terminaux utilises pour la reservation de billets de train, ou encore pour l'interface de l'annuaire electronique offert sur le Minitel. Les formulaires ont deux caracteristiques principales. D'une part, des informations de contexte, ainsi que les donnees deja saisies, sont presentes en permanence. D'autre part, même s'il existe un ordre privilegie, on peut saisir les donnees dans un ordre arbitraire, en se deplacant d'une zone de saisie a l'autre. On peut même revenir sur des donnees deja saisies, ce qui est difcile en mode conversationnel.

Manipulation directe

Bien que pressentie auparavant, la notion de manipulation directe a ete identiee par Ben Shneiderman [Shneiderman 83]. Selon sa denition, une interface a manipulation directe montre en permanence les objets manipules et permet d'agir dessus par des moyens physiques (souris, ecran tactile, etc.) plutôt que par une syntaxe complexe. Par ailleurs, les operations doivent être rapides, incrementales, et reversibles. Leur effet sur les objets est immediatement visible.

Certaines applications a manipulation directe sont tres repandues aujourd'hui. On connaît les outils de dessin ou de traitement de texte. MacPaint et MacWrite en ont ete les precurseurs dans le grand public. De nombreux tableurs utilisent des techniques similaires. Le premier d'entre eux, Visicalc, est d'ailleurs anterieur a la denition de Shneiderman. C'est aussi le cas des jeux video, qui sont une parfaite illustration de ce concept. Enn, on connaît aussi les interfaces iconiques, qui representent les entites a manipuler (souvent des chiers) par de petits dessins statiques (des icônes), sur lesquels l'utilisateur agit avec la souris et parfois le clavier.

L'interaction par manipulation directe est consideree comme superieure a l'interaction par conversation, dans de nombreux cas. Cependant, le terme donne parfois lieu a interpretation. Il n'est pas rare de rencontrer des applications dites \a manipulation directe", qui ne font que representer les donnees a manipuler. L'interaction s'y fait par d'autres voies, qui prennent souvent la forme d'une conversation. Que celle-ci se fasse sous forme graphique par des menus ou des botes de dialogue, importe peu : il ne s'agit pas reellement de manipulation directe. Ou plutôt, dans ce cas on manipule directement un jeu d'options presentees par le menu ou la boîte de dialogue, mais pas l'entite abstraite sur laquelle on veut agir.

La manipulation directe s'accommode mal des modeles utilises pour realiser d'autres types d'interaction. Elle est souvent coûteuse a obtenir, ne serait-ce que parce qu'elle demande des modes d'interaction speciques a chaque abstraction a manipuler.

Par exemple, on n'agit pas de la même maniere sur une icône dans une fenêtre, ou sur un schema de câblage electrique. C'est pour cette raison qu'il est tentant de faire appel a des interacteurs existants, comme des menus. En effet, ces derniers permettent parfois une interaction moins facile, mais ils ont l'avantage d'être reutilisables. Pour realiser facilement des interfaces a manipulation directe, il faut identier des composants de base reutilisables, ce qui est encore un sujet de recherche.

Interacteurs usuels

Dans les interfaces textuelles, les modes d'interaction peuvent être varies, mais l'entite de base en est toujours la saisie de caracteres. Cette saisie est reclamee par un champ de saisie ou une \invite", et accompagnee par l'echo des caracteres tapes et le deplacement d'un curseur. Pour les interfaces graphiques, d'autres techniques de saisie ont ete inventees, dont un certain nombre d'interacteurs qui utilisent la manipulation directe.

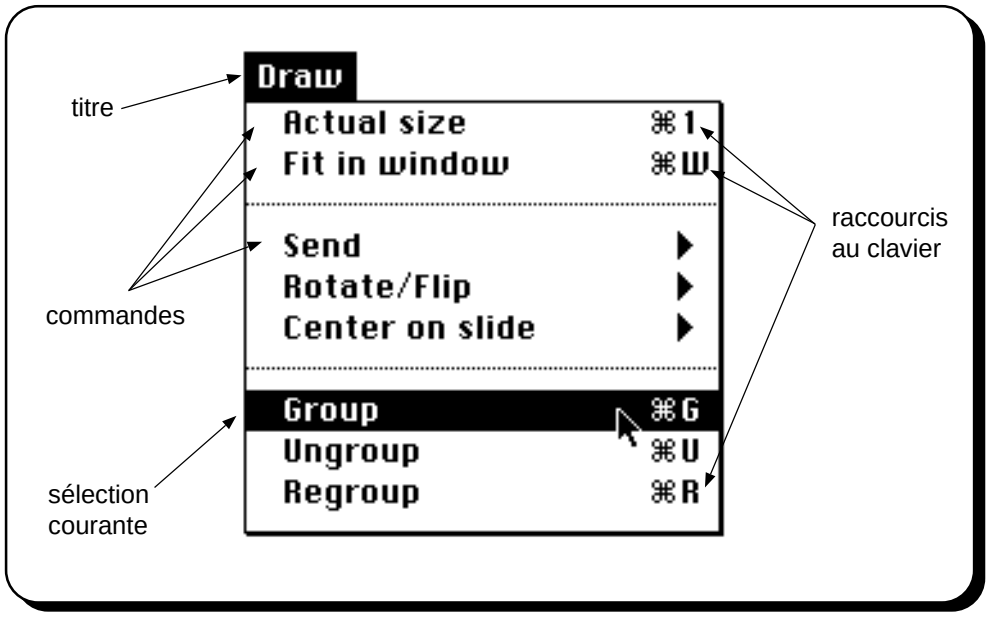

Figure 2.1 : Un menu est utilise pour declencher des commandes.

Le plus populaire des interacteurs usuels est le menu, qui permet le choix d'une valeur parmi plusieurs. On l'utilise generalement pour declencher des commandes. Un menu, bien que d'usage simple, est relativement delicat a realiser. Plus simples sont les boutons, eux-aussi utilises pour declencher des commandes. On trouve aussi des cases a cocher, pour saisir des valeurs booleennes, et des boutons-radio, pour choisir parmi des valeurs enumerees. Les boutons, cases a cocher, et boutons-radio, avec les textes

a saisir, sont les briques de base qui permettent de construire des boîtes de dialogue. Citons enn les barres de delement, qui peuvent servir a saisir des nombres entiers, mais sont surtout repandues autour des fenêtres d'afchage, pour en faire deler le contenu.

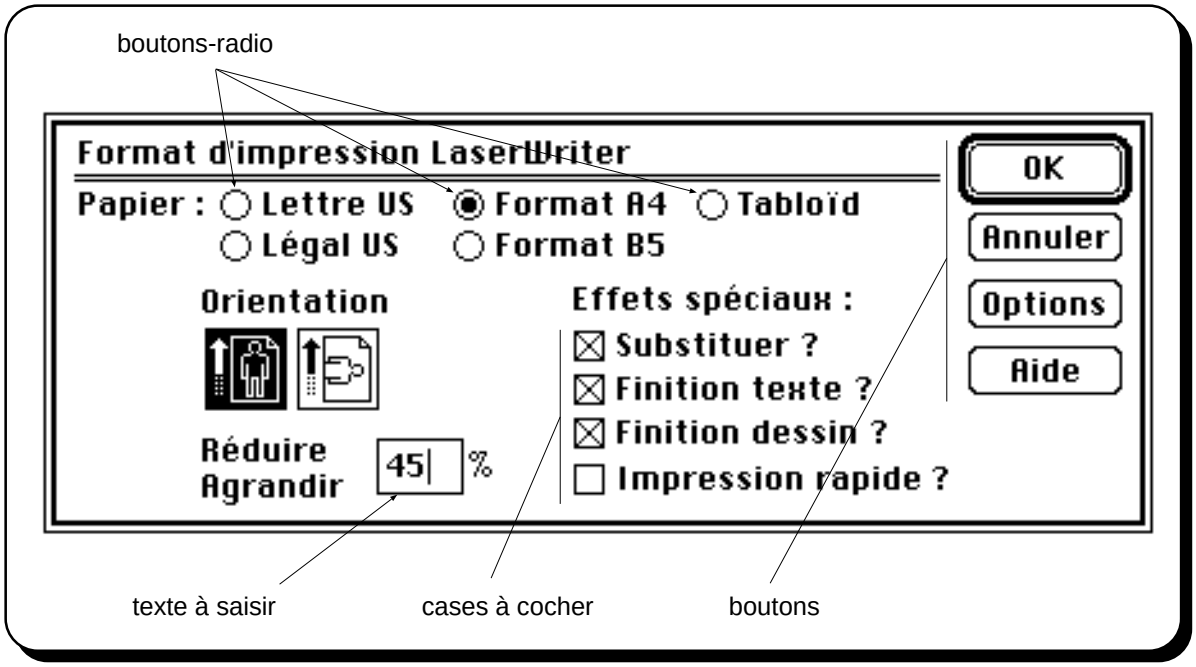

Figure 2.2 : Une boîte de dialogue constituee de divers interacteurs.

2.1.3 Techniques de base

Les materiels que nous avons evoques a la section 2.1.1 sont tres differents les uns des autres. Même parmi les ecrans graphiques, les fonctions varient d'un modele a l'autre. Cependant, on retrouve a peu pres partout les mêmes techniques de base, aussi bien pour le dessin que pour la gestion des entrees. Ces techniques sont souvent mises en uvre par des logiciels de base. Ces logiciels, qu'on nomme bibliotheques graphiques ou systemes graphiques, ont deux rôles. D'une part, ils encouragent une bonne utilisation du materiel. Et d'autre part, ils protegent les programmes, et donc indirectement les utilisateurs, des variations entre differents materiels. Nous allons presenter certaines techniques de base concernant le dessin, la gestion des entrees de l'utilisateur, ainsi que la technique du fenêtrage.

Dessin

Les techniques de dessin dependent beaucoup du type de materiel disponible. On n'utilise pas un ecran adressable point par point comme un ecran a balayage cavalier. Dans le second cas, on peut commander le trajet du faisceau d'electrons dans le tube cathodique. Cela permet donc de realiser des formes geometriques arbitraires de maniere relativement facile. En revanche, reproduire une image denie point par point, telle qu'une photographie numerisee, est plus delicat. Pour cette raison, ces ecrans sont surtout utilises pour des taches ou les dessins sont structures geometriquement, comme la conception assistee par ordinateur.

Les ecrans dont les points sont accessibles individuellement sont aujourd'hui beaucoup plus repandus. Leur surface est divisee en une grille de petits elements d'image rectangulaires, nommes pixels[1]. Chacun de ces elements possede une representation en memoire. En noir et blanc, il suft d'un bit par pixel. Pour la couleur, le nombre de bits par pixels determinera le nombre de couleurs disponibles en meme temps. Ainsi un ecran possedant 8 bits par pixel aura 2^8 couleurs disponibles. La valeur d'un pixel est le plus souvent utilisee comme index dans une table de couleurs, le nombre de couleurs possibles etant alors tres superieur au nombre de couleurs disponibles en meme temps.

Avec de tels ecrans, le dessin est obtenu par la modication de la memoire correspondant a un pixel. On peut en particulier y transferer une nouvelle valeur, ce qui revient a un transfert d'image. On peut aussi effectuer des operations logiques entre la valeur initiale et la valeur transferee. Ainsi, le OU exclusif est souvent utilise pour donner une impression de superposition d'images ; de plus, cette operation a l'avantage d'etre reversible, ce qui permet de realiser facilement des deplacements d'images sur un fond immobile. La base du dessin est donc le point, et tous les dessins doivent etre echantillonnes pour etre exprimes en termes de pixels. Ainsi, il sera facile de dessiner un rectangle horizontal, mais dessiner un segment incline demandera un soin particulier, pour eviter une apparence crenelee. On trouvera le detail des techniques utilisees dans des ouvrages generaux comme [Foley et al. 90] ou [Rogers 85]. De la meme maniere, la reproduction d'une image sera facile seulement si la taille de l'image est correcte. Les changements de taille d'une image composee de points sont particulierement difciles, et demandent des calculs couteux.

La premiere maniere de dessiner sur un ecran point-par-point est donc de modier individuellement des points. Cependant, on souhaite en general dessiner des formes plus complexes. Pour realiser cela, des bibliotheques graphiques sont apparues, qui encapsulent des algorithmes de dessin dans des procedures appelees *primitives de dessin*. Avec de telles bibliotheques, on peut donner des ordres de trace de segments, voire de rectangles ou de cercles. Les premieres de ces bibliotheques etaient inspirees

[1] de l'anglais *pict*ure *el*ement.

des mecanismes des tables tracantes. Dans ces systemes, il existe un crayon courant, qui determine l'aspect du dessin : epaisseur du trait, couleur, operation logique effectuee pour le transfert. Le crayon possede une position courante, et les operations disponibles sont a base de deplacements du crayon : aller en (10, 10) ; tirer un trait jusqu'en (100, 10). Les dimensions sont en general exprimees en pixels, avec un eventuel facteur d'echelle. La bibliotheque graphique du Macintosh, QuickDraw, utilise un tel modele a base de crayon courant [Apple Computer Inc. 85]. Des systemes plus recents ont abandonne la notion de position courante, et même de crayon courant. On y rencontre ainsi des ordres plus synthetiques, et mieux delimites : tirer un trait entre (10, 10) et (100, 10) avec le crayon 1 ; dessiner un rectangle de coordonnees (10, 10, 40, 20) avec le crayon 2. C'est le cas du systeme de fenêtrage X Window System [Scheier & Gettys 86], ou la notion de crayon a par ailleurs laisse place a celle, plus generale, de contexte graphique.

Les dernieres evolutions dans les modeles de dessin ont radicalement change la maniere de voir le dessin. Jusqu'a present, nous avons considere le dessin comme une suite d'ordres graphiques. Les systemes les plus recents, quant a eux, s'interessent plus a la description du contenu de l'ecran qu'a la maniere de l'obtenir. Cette description repose sur la notion d'objets graphiques : segments, rectangles, etc. Elle est facilitee par les langages a objets [Meyer 90 ; Masini & al. 89], qui permettent d'attacher des fonctions a des objets. Ainsi, les ordres graphiques sont desormais associes aux objets graphiques, sous la forme d'une fonction de dessin. La fonction de dessin d'un rectangle sera differente de celle d'un cercle, mais utilisable de la même maniere. Ces modeles de dessin a base d'objets graphiques, apparus recemment, sont desormais tres populaires, et mis en uvre par la plupart des boîtes a outils pour interfaces graphiques apparues recemment.

Gestion des entrees

Au plus bas niveau, la gestion des peripheriques d'entree se fait de deux manieres. Soit il faut regulierement lire l'etat d'un peripherique, soit ce dernier envoie des impulsions qui interrompent l'execution en cours et forcent la lecture. Sur certains micro-ordinateurs, c'est avec ces mecanismes que sont realisees des applications interactives telles que les jeux. Des systemes d'exploitation comme UNIX uniformisent la gestion des peripheriques en l'integrant au systeme de chiers. Chaque peripherique est represente par un chier dans lequel on peut lire, comme dans un chier de texte. Les particularites du peripherique sont alors gerees par le noyau du systeme d'exploitation. La gestion des entrees est uniformisee, mais repose toujours sur l'attente active. Avec le systeme graphique GKS, par exemple, c'est l'application qui doit prendre l'initiative de lire l'etat des dispositifs, apres avoir sollicite une action de l'utilisateur.

Utiliser ce genre de techniques presente deux inconvenients. Premierement, cela demande de connaître a l'avance les dispositifs connectes, ce qui devient une contrainte

difcilement supportable. Ensuite, la lecture active n'encourage pas une bonne structuration des programmes. En effet, le risque est grand de ne lire les entrees qu'aux endroits precis du programme ou l'on pense en avoir besoin. Par ailleurs, on effectuera souvent la même lecture a de multiples endroits du programme. En general, on obtient de cette maniere des applications ou une seule entree est possible a un instant donne. De plus, toute modication du programme devient difcile en raison de la complexite de sa structure.

La technique des evenements a ete introduite pour remedier a ces inconvenients. Avec cette technique, toutes les informations obtenues par interruption ou par lecture active sont stockees dans des evenements, lesquels sont disposes dans une le d'attente. La gestion des entrees est alors ramenee a la gestion des evenements : le programme extrait un par un les evenements de la le, et les traite. De cette maniere, il est plus facile de structurer les programmes. On rencontre en general une boucle principale, constituee de la lecture d'un evenement, suivie de son traitement, en fonction de son type. Le type d'un evenement permet d'identier sa nature, et le peripherique dont il est issu. Ainsi, on trouvera un traitement pour les evenements *MouseMove* (deplacement de la souris), et un autre pour les *KeyPress* (pression sur une touche). Par la suite, lorsque les applications atteignent une certaine taille, le traitement des evenements est reparti en plusieurs procedures. Dans les systemes les plus evolues, on pourra même trouver des fonctions de traitement des evenements dans les objets graphiques, la repartition entre les objets etant faite par le systeme graphique sur la base de la position ou l'evenement s'est produit.

Fenêtrage

Les systemes de fenêtrage sont maintenant largement repandus. Ils permettent de partager l'ecran entre plusieurs zones de visualisation utilisees independamment. Ces systemes introduisent la notion de *fenêtre*. Pour l'utilisateur, la fenêtre est une zone generalement rectangulaire occupant une partie de l'ecran. Pour le programmeur, c'est un veritable ecran virtuel, qu'une application gere sans se preoccuper de ce qui se passe sur le reste de l'ecran physique. Par ailleurs, les systemes de fenêtrage offrent aussi la gestion des actions de l'utilisateur, a travers un mecanisme d'evenements. A chaque evenement est associee la fenêtre dans laquelle il s'est produit, et seul le programme proprietaire de cette fenêtre recoit l'evenement.

Ce modele des fenêtres a donne lieu a de nombreuses variations. Dans certains cas, l'ecran contient une simple liste de fenêtres. Dans d'autres cas, les fenêtres peuvent être incluses les unes dans les autres, denissant ainsi un arbre de fenêtres. De nombreux programmes utilisent cette caracteristique pour se decharger de la gestion des evenements : ils multiplient les fenêtres, et laissent le systeme de fenêtrage determiner dans laquelle chaque evenement se produit. Par ailleurs, on trouve diverses politiques

de gestion des fenêtres. Certains systemes interdisent les superpositions ; d'autres les autorisent. Certains ajoutent des decorations autour des fenêtres principales. Enn, certains systemes comme le X Window System n'imposent aucune politique, mais permettent l'utilisation d'un *gestionnaire de fenêtres* independant. De cette maniere, toutes les politiques sont possibles selon le gestionnaire utilise.

Le fenêtrage introduit un probleme nouveau, lorsque les fenêtres peuvent se superposer. En effet, cela signie qu'une partie d'une fenêtre peut être cachee, puis decouverte. Il faut donc reafcher cette partie. Dans certains cas, le systeme de fenêtrage conserve la memoire des parties cachees, et les reconstitue automatiquement. Mais une telle technique consomme potentiellement beaucoup de memoire, alors qu'il n'y a parfois rien de signicatif dans les parties cachees. Cette technique n'est donc utilisee que dans des cas particulier, pour des zones dont on sait qu'elles ne sont masquees que de maniere transitoire. Dans les autres cas, ce sont les programmes proprietaires des fenêtres qui sont charges de redessiner le contenu des parties decouvertes. Pour cela, le systeme de fenêtrage leur envoie des evenements particuliers, qui indiquent quelles regions doivent être reconstruites. Cette nouvelle responsabilite des programmes est une forte motivation pour les modeles de dessin a base d'objets graphiques que nous avons mentionnes plus haut. En effet, un programme realise a partir d'ordres de dessin est difcile a adapter pour lui permettre de gerer le reafchage de zones arbitraires.

2.2 *Modelisation*

Qu'il s'agisse de physique, d'economie ou d'informatique, les activites de modelisation ont deux objectifs majeurs. Il s'agit dans un premier temps de decrire un phenomene, un objet complexe ou une activite a partir d'un certain nombre d'entites abstraites et de leurs relations. Le modele doit permettre la description dele de ce que l'on connaît deja. On peut alors utiliser cette description pour diverses manipulations : comparaisons, calculs, ou preuves. Ensuite, on attend du modele qu'il permette la prediction de nouveaux phenomenes, la description de nouveaux objets. Le modele doit servir de base a l'imagination pour faire evoluer les techniques. Dans le domaine des interfaces graphiques, de nombreux aspects font l'objet de modelisation. On cherche a obtenir des modeles quantitatifs decrivant le comportement de l'utilisateur ; de tels modeles, comme GOMS ou Keystroke [Card et al. 83] sont decrits dans [Coutaz 90]. On modelise aussi le cycle de conception d'une interface [Shneiderman 87]. Ces activites ne sont pas directement liees a notre propos, et nous ne les aborderons pas ici.

Dans le domaine même de l'ingenierie des interfaces, la recherche de modeles est un point essentiel. Plus les interfaces deviennent sophistiquees, et plus elles deviennent coûteuses. Elles representent souvent plus de la moitie de l'effort de conception et de realisation des applications, sans parler de leur maintenance et leur evolution. La raison

principale est que les interfaces ont une structure tres complexe. Sans cadre de travail, sans modele clair, la complexite devient impossible a gerer. De la même maniere, les relations entre la partie purement interactive et la partie purement fonctionnelle d'une application peuvent être tres complexes. Pour ces raisons, de nombreux travaux sont consacres a la recherche de modeles d'architecture.

Il existe plusieurs approches a cette modelisation. Tout d'abord, on peut mettre l'accent sur l'un ou l'autre objectif de la modelisation. Pour certains, l'interêt majeur des modeles d'architecture est la description claire des techniques actuellement employees. Les modeles servent alors de guide pour la conception, ou de critere de comparaison entre systemes. Pour d'autres, les modeles doivent aussi servir a faire evoluer les techniques, vers plus de generalite et plus de facilite d'utilisation. Ils doivent aussi servir de base a des outils permettant la construction de systemes. On retrouve cette dualite dans les centres d'interêt choisis pour les modeles. Les uns decrivent les grandes lignes de l'architecture des applications interactives, en insistant sur le respect de regles de genie logiciel. Les autres cherchent a identier les elements de base qui composent les interfaces, et la maniere dont il faut les organiser. Ces deux approches sont necessaires et complementaires : nous avons besoin de modeles concrets pour realiser des interfaces. Mais ces modeles concrets doivent respecter des regles d'architecture generale, an de ne pas perdre lors de la mise au point le temps que l'on gagne lors de la construction. Il suft pour cela de disposer de modeles abstraits et concrets compatibles entre eux, mais nous verrons que cela pose des difcultes.

Principes generaux

Il est une regle universellement admise pour l'architecture des applications interactives. Cette regle stipule qu'il doit exister une separation la plus nette possible entre deux composantes de l'application : le noyau fonctionnel et l'interface. Le noyau fonctionnel contient les donnees et les traitements speciques a un domaine d'application. C'est par exemple un programme de calcul scientique que l'on cherche a interfacer, ou un systeme expert pour jouer aux echecs, que l'on va introduire dans un jeu sur ordinateur. Quant a l'interface, c'est elle qui gere la presentation des informations et l'interaction avec l'utilisateur. Elle peut utiliser des primitives de lecture et d'ecriture de caracteres, gerer un ecran graphique et une souris, ou encore contenir un systeme de traitement du langage naturel. Selon les applications, ces deux composantes auront des importances differentes. Le noyau fonctionnel d'un editeur de texte sera tres reduit, de même que l'interface d'un systeme de calcul par elements nis.

Cette separation entre noyau fonctionnel et interface est souhaitee pour plusieurs raisons. Il faut d'abord y voir une raison conjoncturelle. Lors de l'apparition des interfaces graphiques, de nombreuses applications semblaient pouvoir tirer un benece de ces interfaces. Cependant, leurs concepteurs voulaient reutiliser la plus grande

partie de ces applications deja existantes. Il fallait donc pouvoir adapter facilement une interface a un noyau fonctionnel deja existant, d'ou un besoin d'independance entre les deux. Depuis, l'experience a montre qu'il est tres difcile d'interfacer un noyau fonctionnel qui n'a pas ete prevu pour cela. Mais de maniere plus generale, on a parfois besoin d'utiliser un même noyau fonctionnel avec des interfaces differentes, selon le materiel disponible. Une societe vendant un tableur, par exemple, voudra developper un seul noyau fonctionnel, et une interface par marque d'ordinateur. Une autre raison a cette separation est la complexite croissante des interfaces. Une interface ne consiste plus en quelques lignes de programme ajoutees rapidement a une application. Le developpement d'une interface est tres coûteux. On souhaite donc pouvoir disposer d'outils ou de composants utilisables pour interfacer divers noyaux fonctionnels. La encore, la reutilisabilite passe par l'isolation des fonctions. Enn, l'organisation du travail impose elle aussi cette separation. Developper un noyau fonctionnel ou une interface ne demandent pas les mêmes competences. Les deux parties seront donc construites par des individus differents, entre lesquels il faut minimiser les interferences.

Poser la separation entre interface et noyau fonctionnel est donc indispensable ; mais cela n'est pas d'une grande aide quant a la structure de l'interface, ni quant a ses relations avec le noyau fonctionnel. Il faut en particulier determiner la nature des informations qui circulent entre les deux parties, et leur niveau d'abstraction. Si ce sont des evenements comme des clics de la souris ou des touches du clavier, l'independance ne sera pas assuree. Si en revanche ce sont des notions du noyau fonctionnel qui circulent, ce dernier sera bien independant de l'interface, mais il faudra alors structurer l'interface elle-même, et isoler les parties qui dialoguent avec le noyau. Par ailleurs, le fait que les applications soient interactives, et non plus seulement graphiques, pose le probleme de la description du dialogue. En particulier, il faut determiner qui de l'interface ou du noyau fonctionnel prend l'initiative des echanges. Pour cela, Tanner a introduit la notion de contrôle du dialogue [Tanner & Buxton 83]. Il distingue le contrôle externe, residant dans l'interface, le contrôle interne, situe dans le noyau fonctionnel, et le contrôle mixte, ou l'initiative passe d'un côte a l'autre selon l'etat du dialogue.

Le modele de Seeheim [Pfaff 85] fournit une base pour toutes ces reexions. Il separe le noyau fonctionnel de l'interface, et divise cette derniere en trois parties : la presentation, le contrôleur de dialogue, et l'adaptateur du noyau fonctionnel (gure 2.3). La composante de presentation gere les entrees et les sorties au plus bas niveau, grâce par exemple a un systeme graphique. L'adaptateur du noyau fonctionnel est la partie qui dialogue avec le noyau fonctionnel, et sert de traducteur entre ce dernier et le reste de l'interface. Enn, le contrôleur du dialogue gere le dialogue avec l'utilisateur, l'association entre commandes et fonctions du noyau fonctionnel, et la coherence entre donnees et representations graphiques.

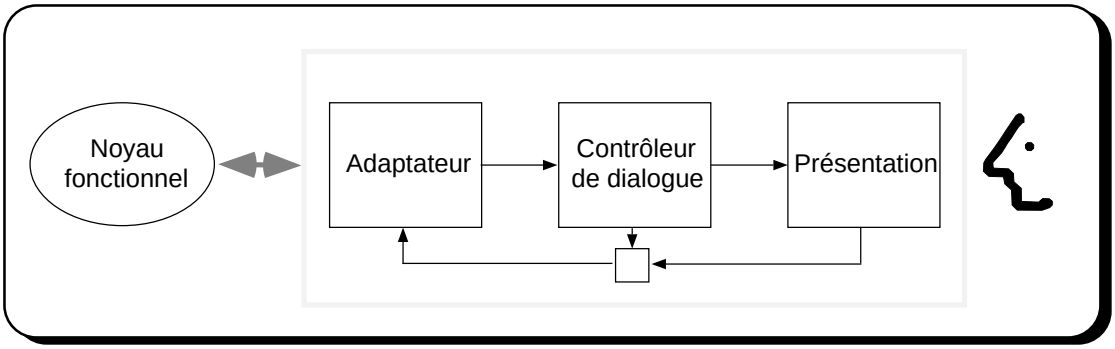

Figure 2.3 : Le modele de Seeheim. Dans le schema d'origine, l'adaptateur s'appelait \Interface de l'application" et le noyau fonctionnel n'apparaissait pas, ce qui a cause de nombreuses confusions.

Le modele de Seeheim a servi, et sert encore, de base a de nombreux outils de construction d'interfaces. Neanmoins, certains concepteurs d'outils se sont plaints de la trop grande rigidite des divisions introduites dans l'interface. Ils ont aussi constate que le modele de Seeheim ne tenait pas compte de l'existence de boîtes a outils pour interfaces, et que la place de ces boîtes a outils dans la composante de presentation n'etait pas claire. Un groupe de travail a donc presente recemment une refonte du modele de Seeheim. Ce nouveau modele, nomme Arch [UID91], puis Slinky [UID92], distingue cinq parties dans une application interactive (voir gure 2.4) : la partie specique du domaine, son adaptateur, la partie de gestion du dialogue, la partie de presentation, et la boîte a outils pour l'interaction. Dans sa version Slinky, ce modele permet a chacune de ces parties de changer d'importance selon les applications. De cette maniere, ce modele permet de decrire et de classier l'ensemble des applications interactives. Cependant comme le modele de Seeheim, le modele Arch/Slinky est essentiellement descriptif, et donc beaucoup trop general et abstrait pour servir tel quel a la construction d'interfaces. Nous allons nous interesser maintenant a des modeles plus concrets et plus prescriptifs.

Modeles linguistiques

Pendant quelque temps, de nombreux travaux ont porte sur la description du dialogue par des modeles inspires de la theorie des langages, et le lecteur en trouvera une recapitulation dans [Hartson & Hix 89]. Il est en effet seduisant de comparer les interactions en forme de conversation, composees d'une suite d'actions et de reponses, a des suites de mots dans des phrases. Comme dans un langage, il faut denir les suites d'actions valides et leur attacher un sens. Par ailleurs, les trois composantes introduites par le modele de Seeheim peuvent être comparees avec les trois niveaux d'abstraction introduits par la theorie des langages. Le niveau lexical est celui des

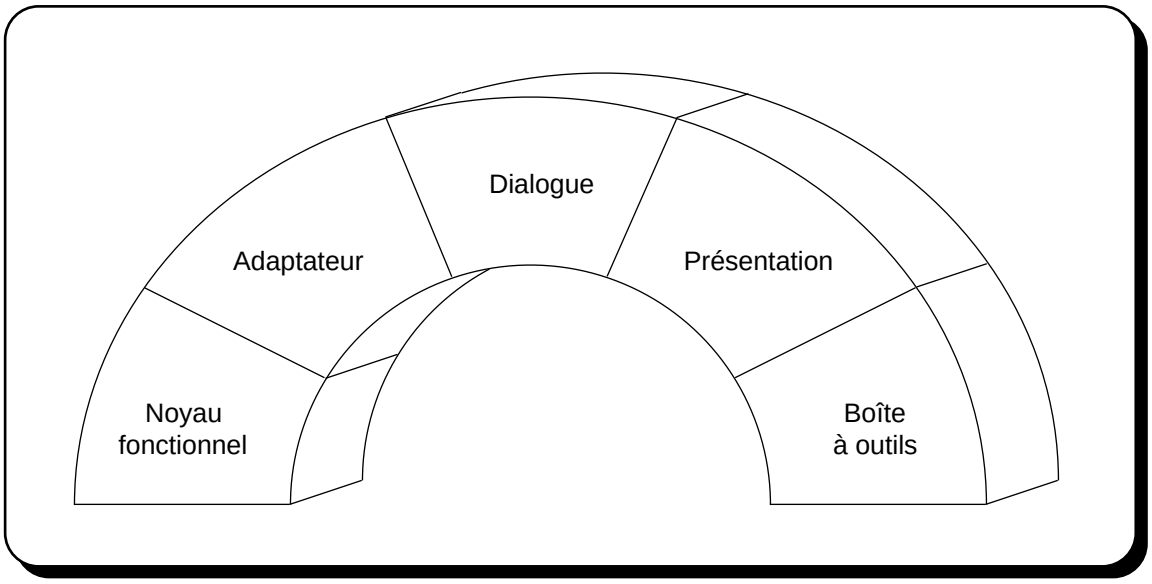

Figure 2.4 : Le modele Arch decompose une application interactive en 5 parties.

mots : c'est l'ensemble des entites atomiques du langage. Le niveau syntaxique est celui des phrases : il decrit quels assemblages de mots sont autorises. Enn, le niveau semantique est celui de la signication du discours : de quoi la phrase parle, et ce qu'elle en dit. Dans le cadre des interfaces, le niveau lexical correspond a la presentation graphique et aux actions atomiques de l'utilisateur : taper sur une touche, bouger la souris. On peut donc associer ce niveau a la composante de presentation. Le niveau syntaxique correspond aux enchaînements d'actions et de reponses, et peut être associe au contrôleur de dialogue. Quant au niveau semantique, il correspond aux operations realisees sur les donnees. On peut donc lui faire correspondre le noyau fonctionnel et son adaptateur.

Un domaine particulierement etudie par la theorie des langages est la description de la syntaxe [Aho et al. 86]. Les techniques qu'on y rencontre ont donc ete utilisees pour decrire le dialogue dans une interface. Ainsi, certains systemes decrivent le dialogue par une grammaire, comme on le fait pour un langage de programmation. Parmi ces systemes, citons Syngraph [Olsen & Dempsey 83], Mike [Olsen 86] ou Edge [Kleyn & Chatravarty 89]. D'autres modeles reposent sur des automates a etats nis. L'ensemble des transitions autorisees y decrit les dialogues possibles. Souvent, on peut enrichir les transitions par des actions arbitraires, comme dans les Augmented Transition Networks, inventes pour le traitement du langage naturel [Woods 70]. Ces actions sont souvent utiles pour produire des retours d'information en cours de dialogue. Rapid/USE [Wasserman 85] utilise des ATN. Des systemes comme UofA* [Singh & Green 91] ou Statemaster [Wellner 89] utilisent d'autres types d'automates : dans un cas, les Recursive Transition Networks [Edmonds 81], et dans

l'autre les Statecharts [Harel 87].

Un avantage des modeles linguistiques est qu'ils permettent une description formelle du dialogue. Il est donc theoriquement possible d'effectuer des vericationssur la coherence de ce dialogue. Cependant, ces modeles perdent aujourd'hui de leur popularite. En effet, ils se sont averes peu adaptes a la description de l'interaction. Tout d'abord, les modeles linguistiques permettent de decrire un langage, et pas les deux langages qui interviennent ensemble lors de l'interaction : celui de l'homme et celui de la machine. En consequence, ces modeles sont utilises en privilegiant le langage des entrees, les sorties etant considerees comme des actions semantiques subordonnees aux entrees.

Par ailleurs, le decoupage en niveaux lexical, syntaxique et semantique est mal adapte aux interfaces. En effet, la gestion des actions les plus simples demande parfois de connaître des informations semantiques. Dans une interface iconique comme le Finder du Macintosh, le passage d'une icône au-dessus d'une autre peut provoquer un changement d'aspect de cette derniere selon la nature des informations representees. On parle dans ce cas de retour semantique. La gestion de ce retour semantique impose une collaboration etroite entre les divers niveaux d'abstraction, ce que ne permettent pas les modeles linguistiques. En fait, le principal probleme des modeles linguistiques est que l'interaction n'est pas toujours reductible a un langage. En particulier, les interfaces a manipulation directe n'ont pas la structure d'une conversation, et aucun modele linguistique ne permet de les decrire de maniere satisfaisante.

Enn, des problemes techniques viennent aussi limiter l'usage des modeles linguistiques. Ainsi, les seules grammaires qu'on sache manipuler de maniere simple ont besoin de connaître un mot d'avance avant de decider quelle regle appliquer. Dans le cas des interfaces, cela signie qu'on ne peut effectuer des retours d'information qu'avec une action de retard. Par ailleurs, le traitement des erreurs, partie importante des interfaces, s'integre mal a ces grammaires. Quant aux automates, leur taille devient vite impossible a gerer pour des interfaces de complexite raisonnable.

Modeles a base d'objets

Les modeles linguistiques cherchent a decrire les interfaces de maniere globale, autour de la notion de dialogue. Les modeles a base d'objets cherchent au contraire a decrire une interface comme une collection d'objets ou d'agents, dont chacun possede une apparence et des capacites d'interaction. Ces modeles ont pu voir le jour grâce a l'apparition des langages a objets, qui facilitent leur mise en uvre. Avec ces modeles, la description de l'etat du systeme est repartie dans les objets. En particulier, l'etat de l'afchage est connu grâce a l'ensemble des objets de presentation, alors qu'avec un modele linguistique, cet etat est le resultat de l'evolution du dialogue et n'est pas decrit explicitement. En revanche, l'etat du dialogue, qui est explicite dans un

modele linguistique, est diffus dans un modele a objets. Cela permet de realiser plus facilement plusieurs interactions en parallele, mais complique toute verication globale de la coherence du dialogue. Les deux approches sont donc opposees, bien que pas necessairement contradictoires. Par ailleurs, dire que l'on decrit une interface comme un ensemble d'objets d'interaction ne fournit pas une structuration claire de l'interface et de ses relations avec le noyau fonctionnel. Nous allons donc examiner deux modeles qui decrivent plus precisement l'organisation des objets : MVC et PAC

Le modele MVC est apparu avec l'environnement Smalltalk, et le langage du même nom [Goldberg 84]. Il decrit une application interactive comme une collection d'entites composees de trois objets : le Modele, la Vue, et le Contrôleur. Le Modele est l'objet abstrait avec lequel on cherche a communiquer. Il reside donc dans le noyau fonctionnel. La Vue est sa representation graphique. Enn le Contrôleur gere les actions de l'utilisateur. Ces trois objets communiquent par des messages.

Le modele MVC permet une certaine structuration des applications, et a l'avantage d'être directement utilisable : c'est un modele concret. Mais il a quelques inconvenients. D'une part, il se revele tres dependant des techniques de programmation par objets. En particulier, la separation stricte en trois objets distincts communiquant par messages peut se reveler inadaptee pour des noyaux fonctionnels qui ne sont pas organises en objets. Par ailleurs, MVC repartit les fonctions de presentation dans les objets Vue et Contrôleur : l'un dessine, et l'autre gere l'interaction. Cette repartition n'a plus vraiment cours avec les outils actuels qui fournissent des objets d'interaction predenis, et gerant a la fois dessin et actions de l'utilisateur. De plus, ces deux fonctions sont parfois indissociables, lorsque l'on deplace un objet avec la souris, par exemple. Enn, MVC n'identie pas clairement l'endroit ou les notions du noyau fonctionnel sont traduites en notions de l'interface. Cette traduction est repartie entre les trois objets, ce qui ne permet pas une organisation claire de la communication entre noyau fonctionnel et interface.

Le modele PAC a ete developpe par Joelle Coutaz [Coutaz 87 ; Coutaz 90]. Comme MVC, PAC decrit une application interactive sous forme d'objets, nommes des agents PAC. Ces agents possedent trois facettes, nommees Presentation, Abstraction et Contrôle. Contrairement a MVC, PAC n'impose pas que ces facettes soient des objets ; de maniere generale, PAC est un modele plus abstrait que MVC, et n'impose pas de methode d'implementation. Les trois facettes des agents PAC ont les rôles suivants. L'Abstraction est l'entite abstraite que l'on cherche a representer ou manipuler. La Presentation regroupe la representation graphique et les capacites d'interaction avec l'utilisateur. Enn, le Contrôle agit comme un mediateur entre les deux autres facettes : il traduit les informations qui circulent, et assure la coherence entre Abstraction et Presentation. Par ailleurs, PAC introduit une notion de hierarchie dans l'organisation des agents. Un agent compose peut gerer plusieurs autres agents. Cet agent compose

peut posseder ses propres composantes d'abstraction et de presentation gerees par son contrôle, mais ce dernier gere en plus les relations entre les sous-agents. Ainsi, lorsque l'abstraction de l'objet de base est modiee, le contrôle de l'agent compose invoque les contrôles des sous-objets, qui repercutent la modication dans leurs abstractions et leurs presentations. Lorsque la presentation d'un sous-agent est modiee par l'utilisateur, son contrôle avertit le contrôle de l'agent compose, qui repercute les modications de la même maniere. J. Coutaz illustre cette possibilite par la representation d'un nombre sous la forme d'un \camembert" associe a la representation textuelle du nombre. Il existe alors deux agents elementaires, pour gerer le camembert et le texte, et un agent compose, qui gere la coherence entre les deux et ajoute un cadre autour des deux objets graphiques (gure 2.5). Grâce a cette notion d'agent compose,

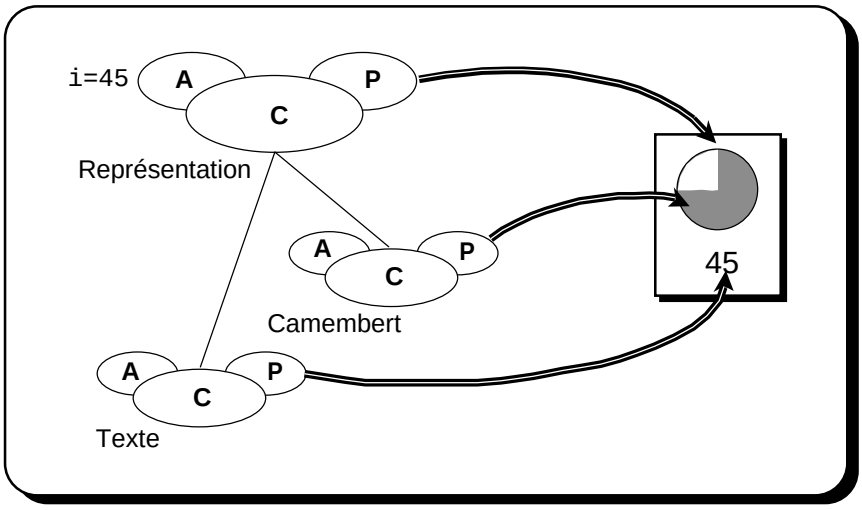

Figure 2.5 : La representation d'un nombre par un camembert et un texte. Un agent PAC compose et deux agents elementaires sont utilises.

PAC permet de decrire l'ensemble d'une application interactive sous la forme d'une hierarchie d'agents.

PAC presente les avantages des modeles a objets, tout en offrant une bonne structuration de l'interface et de ses relations avec le noyau fonctionnel. Par ailleurs, PAC est un modele abstrait et general, ce qui presente des avantages, mais aussi quelques inconvenients. En fait, PAC est precieux comme guide pour concevoir des interfaces, mais il n'est pas sufsamment precis et directif pour être utilise comme modele d'implementation. Ainsi, PAC laisse reposer sur le programmeur le choix de la maniere dont les relations entre Presentation, Abstraction et Contrôle sont realisees. De la même maniere, aucune indication n'est donnee pour la gestion de la hierarchie des agents. Lorsque le noyau fonctionnel est deja organise de maniere hierarchique, les relations entre cette hierarchie et celle des agents PAC sont difciles a etablir avec

les mecanismes des langages a objets dont on dispose aujourd'hui. Par ailleurs, PAC repartit la frontiere entre interface et noyau fonctionnel dans toute une hierarchie d'agents. Cette repartition permet de prendre en compte les nombreux cas ou la frontiere est difcile a etablir. En revanche, cela ne facilite pas la creation d'outils de construction d'interfaces mettant en uvre ce modele. PAC est donc un modele utile et adaptable a la plupart des applications, mais l'absence d'outils associes limite pour l'instant son impact aupres des programmeurs.

2.3 *Outils*

Nous avons examine a la section precedente les principaux modeles qui decrivent l'architecture des applications interactives. Nous allons maintenant nous interesser a toute la gamme d'outils qui ont ete concus pour faciliter la mise en uvre de ces modeles, ou simplement pour permettre la construction d'interfaces, sans prejuger d'un modele d'architecture particulier. Ces systemes vont de la boîte a outils, qui fournit des objets d'interaction predenis, jusqu'aux environnements de programmation pour le developpement d'applications interactives.

Boîtes a outils

Des systemes graphiques tels que le X Window System, ou QuickDraw sur le Macintosh permettent de gerer l'ecran et les actions de l'utilisateur, et donc de realiser des applications interactives. Cependant, nous avons vu qu'il existait un certain nombre de techniques d'interaction bien connues, et en particulier des interacteurs tels que menus, boutons ou boîtes de dialogue, qui ont leur place dans la plupart des interfaces. Ces interacteurs sont faciles a utiliser, mais sont etonnamment difciles a construire avec un systeme graphique.

La premiere vocation des boîtes a outils est de fournir de tels interacteurs predenis, prêts a l'emploi. C'est le cas de systemes comme la Toolbox du Macintosh, ou Motif dans l'environnement X. L'avantage de ces boîtes a outils est double. D'une part, elles economisent du temps de developpement en fournissant des composants logiciels reutilisables. D'autre part elle garantissent une certaine qualite, et surtout l'uniformite entre applications. Cette uniformite, assuree par la Toolbox et renforcee par un guide de style [Apple Computer Inc. 86], est une raison essentielle du succes du Macintosh.

Au-dela des interacteurs predenis, on trouve d'autres caracteristiques communes a de nombreuses interfaces. Ainsi, des services comme le copier-coller d'objets, le defaire-refaire d'operations, ou la personnalisation de la presentation, sont tres repandus. Certaines boîtes a outils comme Interviews [Linton et al. 89] ou ET++ [Weinand et al. 88] fournissent de tels mecanismes prêts a l'emploi.

Enn, la plupart des interfaces sont construites avec des interacteurs predenis, mais ont aussi leur propres representations de donnees. Ces representations, indispensables si l'on veut realiser une interface a manipulation directe, sont speciques a chaque domaine d'application, et donc peu reutilisables. On rencontre depuis peu des botes a outils qui aident a construire de telles representations. Pour cela, elles fournissent le moyen de fabriquer des dessins de maniere structuree, et surtout les mecanismes pour que l'utilisateur puisse agir sur ces dessins. Le chapitre 3 est consacre a l'etude des mecanismes fournis par les botes a outils pour la manipulation directe. Nous y etudierons X_{TV}, qui a ete concue specialement pour ce type d'interfaces.

Squelettes d'application

Les botes a outils fournissent les briques de base pour realiser des interfaces, mais offrent rarement les mecanismes pour organiser ces briques. De plus, elles ne donnent jamais d'indications sur la maniere dont il faut integrer interface et noyau fonctionnel. Pourtant, on retrouve souvent les mêmes structures, et en particulier une boucle principale de traitement des evenements, autour de laquelle tout s'organise. Il est donc envisageable d'economiser l'ecriture de ces parties communes. Les squelettes d'application realisent cela en fournissant des programmes deja structures, dans lesquels il suft d'ajouter les parties speciques a chaque application. De cette maniere, le concepteur de l'application est constamment guide dans ses choix.

Il existe deux techniques principales pour realiser des squelettes d'application. La premiere possibilite est de fournir un chier source incomplet, dans lequel il faut combler les trous. Sans precautions particulieres, cette technique a des consequences catastrophiques : il devient en effet impossible de distinguer le squelette initial et les fonctions ajoutees. Cela signie que toute evolution du noyau fonctionnel ou de l'interface est impossible sans reprendre la conception au depart. La seconde technique consiste a utiliser les mecanismes des langages a objets, et en particulier la possibilite de creer des classes derivees. Le squelette prend alors la forme d'un jeu de classes de base qu'il faut deriver pour les specialiser. Ainsi, les parties communes sont isolees dans l'implementation des classes de base, et les parties speciques dans celle des classes derivees. La plupart des squelettes recents utilisent cette technique. C'est le cas de Grow [Barth 86], MacApp [Shmucker 86], ou APEX [Coutaz 90]. ET++ fournit aussi des classes de base inspirees de MacApp.

Un autre defaut potentiel des squelettes est la mauvaise isolation de l'interface et du noyau fonctionnel. Il est en effet essentiel de disposer d'une composante de contrôle, qui agit comme mediateur entre les objets de presentation et les abstraction [Coutaz 90]. En l'absence de contrôle, les deux parties sont amenees a se referencer explicitement l'une l'autre, ce qui fait perdre le benece de la reutilisabilite. C'est le cas par exemple de MacApp et ET++.

Generateurs d'interfaces

Nous avons vu que les boîtes a outils, si utiles soient-elles, ne sont pas sufsantes pour programmer une application interactive. Elles possedent aussi un autre inconvenient : leur utilisation demande des efforts de programmation. Or la conception d'une interface, que ce soit pour sa partie de presentation ou pour l'organisation generale du dialogue, demande des competences differentes de la programmation. Ces competences en ergonomie, psychologie cognitive, ou simplement conception graphique, sont rarement presentes chez les programmeurs.

Une categorie d'outils a donc vu le jour pour permettre la fabrication d'une interface, ou tout au moins de sa partie de presentation, sans connaissance particuliere de la programmation. Bon nombre de ces outils sont interactifs et permettent d'assembler des interacteurs par manipulation directe, de la même maniere qu'avec un outil de dessin. Les premiers systemes qui permettaient cela etaient des produits de recherche : SOS Interface [Hullot 86], Grafti [Karsenty 87], et Dialogue Editor [Cardelli 87]. On trouve desormais de nombreux produits commerciaux similaires, comme HP Interface Architect, TeleUse de Telesoft, XFaceMaker de NSL, ou Interface Builder de NeXT. D'autres produits comme le User Interface Language de l'OSF permettent de decrire la presentation d'une interface sous une forme textuelle.

La plupart des generateurs d'interfaces sont associes a une boîte a outils, dont il permettent en fait de creer et parametrer des interacteurs. Cela signie que ces generateurs sont limites sur deux plans. D'une part, ils ne sont pas adaptes a la creation d'interfaces a manipulation directe, dans la mesure ou les boîtes a outils le sont rarement. D'autre part, les generateurs d'interfaces gerent les relations avec le noyau fonctionnel au même niveau que les boîtes a outils, c'est-a-dire de maniere tres pauvre. Seuls des systemes comme Grafti ou XFaceMaker ont introduit la notion de *valeur active*, qui enrichit cette communication. Les valeurs actives sont des donnees gerees en commun par le noyau fonctionnel et l'interface, de sorte que les modications effectuees d'un côte sont immediatement repercutees de l'autre. Ce mecanisme de valeurs actives permet au noyau fonctionnel de n'avoir aucune connaissance de la maniere dont ses donnees sont representees.

Environnements de developpement

L'abreviation UIMS signie User Interface Management System, c'est a dire Systeme de Gestion d'Interfaces Utilisateur. Cette notion a ete inventee par D. Kasik en 1982, qui s'est inspire de l'existence des Systemes de Gestion de Bases de Donnees [Kasik 82]. Notons que le terme d'UIMS a ete l'objet de nombreuses interpretations contradictoires depuis quelques annees, au point que son sens est devenu ou. Dans l'ideal, un UIMS serait un systeme qui permettrait la conception d'une interface, la specication du dialogue et de la presentation, puis l'implementation, le test et la maintenance. De tels

systemes n'existent pas a ce jour, mais on en rencontre des sous-ensembles, que B. Myers nomme User Interface Design Systems [Myers 89a]. Les generateurs d'interfaces que nous avons rencontres plus haut, et qui sont en fait des generateurs de presentation, en sont des versions primitives. Nous avons cite a la section 2.2 des systemes comme Rapid/USE ou UofA*, qui s'interessent surtout a la specication du dialogue. D'autres systemes plus recents comme Serpent [Bass et al. 88 ; Bass & Coutaz 91] ou Garnet sont plus representatifs de l'etat de l'art. Nous allons examiner le fonctionnement de Garnet.

Garnet est un ensemble d'outils destines a la construction d'interfaces, qui est developpe a Carnegie Mellon sous la direction de Brad Myers [Myers et al. 90]. Garnet est construit au-dessus d'un systeme a objets nomme KR, et d'un mecanisme de propagation de l'information a base de contraintes unidirectionnelles, ou formules. KR, plutôt que le modele habituel de classes et d'instances, utilise un modele a base de prototypes [Lieberman 86]. Pour sa composante de presentation, Garnet utilise un systeme graphique nomme Opal, et surtout un ensemble de sept techniques d'interaction, dont Myers explique qu'elles sufsent a decrire tous les styles d'interactions a base de souris et clavier [Myers 89b]. Un ensemble d'interacteurs est construit au-dessus de ces briques de base, constituant ainsi ce qui est appele la boîte a outils de Garnet, au-dessus de laquelle sont construits des outils logiciels. Le premier de ces outils est le generateur d'interfaces Lapidary. En plus de l'assemblage interactif d'objets graphiques, il autorise la declaration des contraintes entre objets, ce qui permet de denir certains comportements a l'interieur de l'interface. En particulier, l'association entre techniques d'interaction et objets graphiques est realise avec des contraintes, ce qui donne une grande souplesse. Le second outil est l'editeur de boîtes de dialogue Jade, qui permet de construire des boîtes de dialogue a partir d'une description textuelle. Le troisieme outil est C32, qui utilise la metaphore du tableur pour permettre la mise au point d'interfaces. Avec cet outil, on peut visualiser un objet de Garnet sous la forme d'une table, chaque attribut etant presente dans une case. Les attributs qui dependent d'autres objets sont presentes sous la forme de formules, comme dans les tableurs habituels. L'etat de la table evolue en même temps que celui de l'objet, et il est possible d'intervenir sur les objets a travers la table. Avec C32, il est possible de naviguer parmi les nombreux objets qui composent une interface, et la mise au point est plus facile.

Garnet est aujourd'hui l'un des systemes les plus avances dans le domaine de la construction d'interfaces, et il est utilise dans de nombreux centres de recherche. Notons toutefois que Garnet est centre autour le la composante de presentation, et que l'interface avec le noyau fonctionnel demande toujours des efforts particuliers.

2.4 Conclusion

Dans ce chapitre, nous avons examine les techniques, les modeles et les outils qui ont ete developpes pour la construction d'applications interactives. Au-dela de leur diversite, on peut retenir plusieurs faits marquants. Tout d'abord, la situation est encore en pleine evolution. Des outils de pointe comme Garnet sont incomplets, et ne correspondent pas directement a un modele. Ensuite, il est encore impossible de xer des regles concernant la communication entre interface et noyau fonctionnel. Cela signie qu'on ne peut pas garantir a un programmeur de noyau fonctionnel qu'il sera facile a interfacer. Enn, la manipulation directe, qui fournit des interfaces de bonne qualite, est difcile a decrire par des modeles et a obtenir par des outils. De plus, n'oublions pas que nous sommes restes ici dans le cadre des interfaces \traditionnelles". Les interfaces multi-utilisateurs, ou multi-modales (plusieurs moyens d'entree utilises en même temps) posent d'autres problemes qui restent a resoudre [Beaudouin-Lafon 91]. Nous verrons par la suite que l'introduction de l'animation pose aussi de nouveaux problemes, mais permet un regard nouveau sur l'architecture des interfaces.

Chapitre **3.**

Une boîte a outils : X

Dans le chapitre precedent, nous avons examine les techniques et les modeles pour decrire et realiser des interfaces homme-machine. Il est apparu qu'il existait une approche essentiellement descriptive et analytique, autour de modeles generaux comme Seeheim ou Slinky, et une approche plus pragmatique, incarnee par des boîtes a outils et des outils de construction d'interfaces. Les travaux presentes dans cette these adoptent essentiellement la seconde approche : le but est de fournir des outils reposant sur des modeles concrets, et qui facilitent la construction d'interfaces evoluees.

Dans cet esprit, ce chapitre presente X_{TV}, une boîte a outils pour la construction d'interfaces a manipulation directe, qui a ete realisee collectivement au cours des travaux exposes dans cette these. Cette presentation n'est pas exhaustive, X_{TV} n'etant pas ici le centre d'interêt principal. Elle repond a deux buts principaux. D'une part, les mecanismes des boîtes a outils vont nous aider a mieux comprendre les problemes techniques souleves par la manipulation directe. Nous retrouverons ces problemes tout au long de cette these. Ensuite, les travaux sur les interfaces animees presentes par la suite utilisent abondamment X_{TV} et son modele de construction des interfaces. La connaissance d'X_{TV} est donc necessaire a la lecture des chapitres suivants.

Le chapitre est organise de la maniere suivante. Dans un premier temps, nous analyserons les mecanismes et les caracteristiques principales des boîtes a outils pour interfaces. Ces caracteristiques serviront de reference pour l'etude de quelques boîtes a outils existantes. Le reste du chapitre sera consacre a la presentation d'X_{TV}, suivie d'un exemple d'utilisation.

3.1 Caracteristiques des boîtes a outils

Les boîtes a outils pour les interfaces graphiques trouvent leur origine dans la diversite des materiels graphiques. On a vite vu se developper des bibliotheques graphiques destinees a proteger les programmeurs de cette diversite. Cependant,

avec le developpement des interfaces telles que nous les connaissons aujourd'hui, les boîtes a outils trouvent d'autres justications. Elles ne servent pas seulement a assurer la portabilite, mais doivent offrir un certain nombre de services pour simplier la programmation d'interfaces.

On peut decomposer les services en plusieurs categories. Tout d'abord, on trouve un noyau qui contient les mecanismes de base pour dessiner, gerer l'interaction, et structurer des interfaces. Dans les systemes les plus simples, ce noyau se resume a des primitives de dessin et de lecture des evenements. Dans les systemes les plus evolues, le programmeur dispose d'un squelette d'application, et il n'a plus qu'a inscrire son programme dans ce cadre ; ces squelettes utilisent en general les mecanismes des langages a objets. Au-dessus de ce noyau, on trouve souvent un ensemble d'interacteurs predenis. La plupart des boîtes a outils actuellement disponibles fournissent des menus, des boîtes de dialogue, et autres objets habituels, prêts a être utilises. Enn, on trouve aussi des services divers, qui ne sont pas speciques des interfaces. Rentrent dans cette categorie les outils de parametrage de l'interface ou la possibilite de stocker et relire des objets graphiques.

Parmi ces services pour realiser des interfaces, l'accent est souvent mis sur les interacteurs predenis. Neanmoins, les mecanismes de base sont plus determinants, en particulier pour les interfaces a manipulation directe. En effet, l'important pour ces dernieres est de parvenir a construire des representations speciques a des applications donnees. Pour cela, la maniere dont la boîte a outils aide a structurer ces representations est cruciale, et les interacteurs predenis sont d'une utilite limitee. Ce besoin de structuration recouvre autant la representation graphique elle-même que la gestion des entrees associees a cette representation. Par ailleurs, l'architecture logicielle qu'encourage une boîte a outils est elle aussi determinante. Certaines n'offrent aucune structuration, et permettent ainsi un melange inextricable entre interface et noyau fonctionnel, tandis que d'autres denissent clairement un protocole de communication entre les deux.

Nous allons donc nous interesser a trois caracteristiques principales des boîtes a outils : le modele de dessin, celui de gestion des entrees, et la communication avec le reste de l'application.

3.1.1 Modeles de dessin

Dans le cadre des applications interactives, le dessin n'est pas un but en soi. On dessine sur l'ecran pour representer des donnees et pour permettre leur manipulation. Par ailleurs, l'interactivite impose des contraintes de performances, qui limitent la complexite des dessins. En consequence, ce qui nous interesse dans une boîte a outils pour interfaces n'est pas tant sa capacite a realiser des dessins tres sophistiques, que la simplicite avec laquelle on peut obtenir et maintenir un dessin a l'ecran. La

caracteristique qui nous interesse le plus est donc le niveau d'abstraction du systeme graphique. Pour l'utiliser, faut-il donner des ordres de dessin, ou suft-il de disposer des objets graphiques a l'ecran ? Les ordres de l'utilisateur seront-ils exprimes en termes de coordonnees de l'ecran, ou d'objets graphiques ? Dans le contexte des systemes de fenêtrage actuels, un corollaire est la question du reafchage : lorsque j'afche un dessin ou un objet graphique a l'ecran, dois-je me preoccuper de le redessiner des qu'il est endommage par d'autres fenêtres ? Selon les reponses a ces questions, on parle de dessin direct ou de dessin structure.

Dessin direct

Dessin direct signie que les fonctions disponibles permettent d'afcher des formes geometriques dans des fenêtres, mais que celles-ci ne memorisent pas leur contenu, si ce n'est en terme de pixels. Le dessin direct peut être avec ou sans etat. *Dessin sans etat* signie que les requêtes d'afchage contiennent toutes les donnees necessaires concernant les attributs graphiques (couleur, police, etc.) et ne font pas reference a un etat courant du systeme graphique ou de la fenêtre. Ainsi, X [Scheier & Gettys 86] offre un modele de dessin direct sans etat. A titre de comparaison, QuickDraw sur Macintosh [Apple Computer Inc. 85] et le langage PostScript [Adobe Systems Inc. 85] ont un modele de dessin direct, mais avec etat. GKS [Bono et al. 82] a un niveau direct avec etat et un niveau non direct (les segments).

Le dessin direct complique singulierement la tâche qui incombe a l'application. En effet, lorsque des objets graphiques representent des donnees, on a souvent besoin de les modier, les deplacer ou les effacer. Le fait que le systeme graphique ne garde pas memoire de ces objets signie que l'application doit le faire elle-même. Cela suppose qu'elle gere une structure representant l'organisation des objets graphiques. Cette structure est aussi indispensable pour reafcher des parties de fenêtres, lorsque le systeme de fenêtrage ne gere pas lui-même le reafchage. Le modele de dessin direct a aussi des consequences sur la gestion des entrees de l'utilisateur. X, par exemple, emet un evenement lors de chaque action de l'utilisateur. Ces evenements sont toujours attaches a une fenêtre, de sorte que pour realiser une application sur le modele de la manipulation directe, il est necessaire de retrouver l'objet cible de l'evenement. La recherche de cette cible se realise grâce a la même structure de donnees que celle necessaire a la gestion de l'afchage. Or gerer cette structure de donnees est une tâche complexe et repetitive d'une application a l'autre.

Par ailleurs, les modeles de dessin direct ne sont pas intuitifs. Il faut être prêt en permanence a redessiner les parties de l'ecran qui ont ete obscurcies, ce qui ne correspond pas a l'idee qu'on se fait habituellement de la tâche de dessin. Cette situation

mene souvent a des styles de programmation inadaptes a la realisation d'interfaces sophistiquees. C'est pourquoi de nombreux systemes proposent maintenant des modeles de dessin structure.

Dessin structure

Le dessin structure consiste non plus a *dessiner* sur un ecran, mais a y *visualiser* des objets que l'on a disposes dans une structure d'afchage. Pour le programmeur, les objets presents dans la structure sont consideres comme presents a l'ecran : c'est le systeme graphique qui gere le reafchage. Les fonctions offertes ne sont plus des fonctions de dessin, mais des fonctions de manipulation de la structure.

Dans les systemes les plus simples, la structure est une liste. Les operations sont donc l'ajout et le retrait d'objets, et les changements d'ordre. L'ordre de la liste determine l'empilement des objets, en ce qui concerne le dessin aussi bien que la reception des evenements. Le dernier objet de la liste, s'il se recouvre avec le premier objet, apparaîtra devant lui, et recevra les evenements avant lui.

La structure la plus courante est l'arbre. D'un point de vue pratique, un arbre est une liste ou chaque objet peut lui même être une liste, et ce a l'inni. On parle dans ce cas d'objet composite. L'interêt d'une telle structure apparaît dans des outils de dessin comme MacDraw sur le Macintosh : cela permet de grouper les objets, et de manipuler ensuite le groupe comme un seul objet. D'autres structures sont plus destinees a optimiser la gestion du reafchage et de la designation des objets. Ainsi, les *quad-trees* organisent les objets en fonction de leur position, de maniere a savoir plus rapidement a quelle partie de la structure correspond une position ou une surface a redessiner. La norme graphique PHIGS [ISO 86] utilise elle aussi une structure arborescente, et permet le rendu en 3 dimensions des objets presents dans la structure. Les differents parametres du rendu, tels que l'echelle ou le point de vue, y sont manipules independamment de la structure.

3.1.2 Modeles d'interaction

Encore plus que la maniere de dessiner, la technique de gestion des actions de l'utilisateur est determinante pour la realisation d'applications interactives. C'est en effet dans ce domaine que vont apparaître les principaux liens, et les conits, entre l'architecture de l'interface et celle du noyau fonctionnel.

Nous l'avons vu au chapitre precedent, la plupart des systemes graphiques modernes utilisent un modele a base d'evenements. Mais, outre qu'il n'y a pas aujourd'hui de methode satisfaisante pour gerer ces evenements (voir la section suivante), le modele des evenements precise seulement comment l'information est propagee. Il ne

decrit pas quelles sont ces informations, ni comment est organisee leur reception. Pour cette raison, les boîtes a outils ont chacune leur propre modele d'evenements.

Il est frequent que les boîtes a outils n'offrent que les types d'evenements fournis par le systeme de fenêtrage. Cela signie qu'il est difcile, sinon impossible, de gerer de nouveaux peripheriques avec ces boîtes a outils. Par ailleurs, il est souhaitable de pouvoir denir de nouveaux evenements ne correspondant pas directement a un peripherique. Ces evenements synthetiques sont parfois utiles pour combiner plusieurs evenements de base. Par exemple, les evenements *Double-Click* et *Drag* proviennent de la combinaison d'evenements *Click* et *Move.* Par ailleurs, les evenements sont avant tout des messages. Dans un modele a objets, on peut se demander quels en sont les destinataires. Dans les systemes les plus primitifs, il n'y a pas de destinataire. La gestion des evenements est explicitement organisee autour d'une boucle principale, ressemblant a ceci :

```
ev = NextEvent ();
while (ev) {
   HandleEvent (ev);
   ev = NextEvent ();
}
```

L'evenement contient en general la fenêtre dans laquelle il s'est produit, et une position. C'est alors au programmeur de determiner quel objet l'utilisateur manipulait, a l'interieur de la fonction *HandleEvent.* C'est pour cette raison que de nombreuses applications multiplient les fenêtres, an de simplier le traitement des evenements. Dans les systemes plus evolues, c'est le systeme qui determine quel objet est la cible de l'evenement. L'objet en question recoit alors l'evenement et peut le traiter. Cela autorise une architecture ou tout le comportement est a l'interieur des objets graphiques. Nous verrons cependant que ces facilites sont encore peu repandues.

3.1.3 Communication avec le reste de l'application

Nous avons vu au chapitre 2 qu'un point important de la construction des applications interactives etait la nature des liens entre l'interface et le noyau fonctionnel. Nous parlerons ici de communication, par reference a la situation idealisee ou ces deux composantes sont dans deux processus separes et communiquent grâce a un protocole. Par ailleurs, on ne peut pas reellement parler ici de communication avec le noyau fonctionnel. En effet, les boîtes a outils actuelles, quel que soit leur degre de sophistication, ne permettent de realiser que la partie de l'interface qui est la plus proche de l'utilisateur. Il manque precisement la partie qui permet une communication claire avec le noyau fonctionnel. C'est pour ces raisons que cette section s'intitule \Communication avec le reste de l'application".

On voit apparaître ici l'inadequation entre les outils les plus repandus et les modeles theoriques. En effet, les boîtes a outils sont en general utilisees directement, puisqu'aucun autre outil n'est fourni pour construire un mediateur entre noyau fonctionnel et couche de presentation. Il se trouve donc que la communication avec le noyau fonctionnel est souvent dans la pratique celle qui est permise par les boîtes a outils.

Cette communication repose partout sur deux mecanismes. D'une part, l'existence d'objets de presentation permet d'etablir un lien entre les donnees du noyau fonctionnel et celles de l'interface. En general, lors de la creation d'un objet du noyau, l'objet de presentation correspondant est cree, et une reference vers lui est conservee dans le noyau. Par ailleurs, l'objet de presentation contient les informations sufsantes pour retrouver l'objet du noyau qui lui correspond. Ces informations sont utilisees si l'utilisateur agit sur la presentation. Elles ont tres souvent la forme d'un nom ou d'un pointeur. L'autre mecanisme repose sur le traitement des evenements de l'interface, qui sont en general traduits en appels de fonctions du noyau fonctionnel. Pour cela, l'adresse de la fonction a appeler est le plus souvent stockee dans une table a l'interieur de l'interface : c'est le mecanisme des \callbacks".

On voit donc que les mecanismes de communication offerts par les boîtes a outils sont assez peu evolues. En fait, ces mecanismes sont directement issus des solutions aux problemes poses par les interfaces graphiques. Les considerations architecturales n'y ont que peu de place. Les distinctions que l'on peut etablir entre differentes boîtes a outils reposent donc surtout sur la possibilite d'ajouter des mecanismes de communication plus evolues. Certaines d'entre elles imposent fortement leur modele de communication, tandis que d'autres se prêtent plus facilement a des extensions.

3.2 Les boîtes a outils existantes

3.2.1 X Toolkit

La X Toolkit est une boîte a outils pour interfaces, developpee au MIT au sein du projet Athena. Son but est de fournir un ensemble d'interacteurs pour la construction d'applications interactives, ainsi que des mecanismes pour les organiser et les parametrer. Ces interacteurs, nommes \widgets", sont les objets que l'on rencontre habituellement dans les interfaces : boutons, menus, ou boîtes de dialogue. La X Toolkit est decomposee en un noyau de base qui offre les mecanismes generaux, et un ensemble d'interacteurs. On trouve aujourd'hui les jeux de widgets Athena, Motif et OpenLook, qui fournissent a peu pres les mêmes fonctions avec des apparences differentes.

La X Toolkit est organisee selon un modele a objets, mis en uvre en langage C par le noyau de base. Les widgets sont organises selon un arbre d'heritage. On peut

distinguer deux branches principales a cet arbre : les widgets simples et les widgets composites. Les premiers sont des objets comme des textes ou des boutons, et les seconds sont les conteneurs qui permettent de les composer. Les differents types de widgets composites mettent en uvre differentes politiques de disposition de leur contenu. La disposition se fait grâce a une negociation entre le conteneur et chacun des objets contenus, en particulier en ce qui concerne la place allouee.

La notion de modele de dessin n'intervient pas vraiment avec la X Toolkit. En effet, elle ne fait qu'offrir des objets predenis. Ces objets gerent eux-mêmes leur structure et leur reafchage. En ce sens, on peut dire qu'il s'agit de dessin structure. Cependant, si le programmeur veut presenter les donnees de son noyau fonctionnel sans utiliser les objets predenis, la X Toolkit ne lui offre aucune aide. Dans ce cas, il doit utiliser les fonctions de dessin direct du systeme de fenêtrage, rassemblees dans la bibliotheque Xlib. En ce sens, la X Toolkit offre le même modele de dessin que le X Window System, c'est-a-dire un modele de dessin direct de tres bas niveau.

En ce qui concerne le modele d'interaction, la X Toolkit repose largement sur le systeme de fenêtrage. A chaque widget correspond une fenêtre. Donc le modele repose sur la gestion d'evenements de X. Pour chaque widget, il est possible d'associer un type d'evenements a une serie d'actions. Ces actions sont des comportements predenis du widget. Une action peut produire un changement d'apparence du widget, avant d'appeler une fonction du noyau fonctionnel. Les fonctions qui sont ainsi appelees depuis des actions sont nommees des \callbacks". Elles constituent le principal moyen de communication entre l'interface et le noyau fonctionnel.

Le traitement d'un evenement peut être parametre de deux manieres. D'une part, il existe dans chaque widget une table d'association qui fait correspondre une suite d'actions a un evenement, ou même a une sequence d'evenements. Cette table permet de modier la maniere dont on manipule un widget. On peut par exemple decider que l'enfoncement du bouton du milieu dans un widget *XmText* declenchera l'action *beep()*, plutôt que l'action *mode-destination()*, qui deplace le curseur dans le texte. Cette table, qui a un contenu par defaut pour chaque widget, peut être modiee par le programmeur de l'application. Elle peut aussi l'être par l'utilisateur, grâce a un chier de conguration. L'autre moyen d'intervention permet d'etablir le lien entre l'interface et le noyau fonctionnel : ce sont les listes de callbacks. Plutôt que d'appeler directement une fonction du noyau fonctionnel, les widgets utilisent des listes de callbacks a appeler. En general, chaque action possede sa propre liste de callbacks. Le programmeur peut modier ces listes, ce qui lui permet d'associer la ou les fonctions qu'il desire a une action.

Discussion

La X Toolkit est tres utilisee a l'heure actuelle, en particulier avec le jeu de widgets Motif. Cependant, il faut constater qu'elle ne correspond pas aux besoins de tous les programmeurs d'applications interactives. Elle ne permet de realiser qu'une partie de l'interface utilisateur : celle qui est composee de menus et de boîtes de dialogue. Cette partie, bien que repetitive et penible a realiser, n'est pas la plus importante. Dans le cas des interfaces a manipulation directe, tout ce qui concerne la representation des objets de l'application et l'interaction sur ces representations doit être realise selon d'autres mecanismes. Une reaction frequente devant cette situation consiste a adapter les applications pour pouvoir utiliser les interacteurs predenis, au detriment de la manipulation directe. Il faut voir la une des raisons de la pietre qualite des applications interactives disponibles dans l'environnement X.

3.2.2 InterViews

InterViews est une boîte a outils realisee a l'universite de Stanford, puis chez Silicon Graphics [Linton & Vlissides 87 ; Linton et al. 89]. Elle utilise les mecanismes du langage a objets C++, et permet la construction d'interfaces a manipulation directe.

InterViews offre un modele de dessin structure [Vlissides & Linton 88]. Des objets graphiques de base tels que des rectangles, des cercles ou des textes sont places dans des fenêtres. C'est InterViews qui gere leur reafchage. Ces objets comportent chacun un etat, qui conditionne leur aspect. L'etat est compose d'attributs tels que la couleur ou l'epaisseur des traits, avec des valeurs associees. Par ailleurs, il existe une notion d'objet graphique composite, qui peut contenir d'autres objets graphiques. Les objets composites ont eux aussi un etat, qui est compose avec l'etat de chacun des sous-objets lors de leur dessin. Il est possible de modier la loi de composition, qui par defaut resout les conits en faveur du parent. De cette maniere, il est possible de modier l'aspect de tous les objets ls en changeant seulement l'etat du parent.

Le modele d'interaction d'InterViews est dissocie de son modele de dessin structure. En effet, les objets graphiques structures decrits plus haut ne peuvent pas recevoir d'evenements. Ce sont les *interacteurs*, similaires aux widgets de la X Toolkit, qui gerent les entrees. Il existe des interacteurs simples et des interacteurs composites, qui peuvent contenir plusieurs interacteurs. Pour la disposition des interacteurs, InterViews utilise un modele de boîtes et de glu similaire a celui de TEX [Knuth 84]. Un interacteur particulier est prevu pour contenir des objets graphiques structures. Les autres interacteurs, et en particulier les objets predenis tels que les menus ou les boutons, n'utilisent pas les objets graphiques structures, et sont construits avec un modele de dessin direct.

Discussion

InterViews est une boîte a outils relativement simple a utiliser. Son modele de dessin structure est puissant, et permet de realiser facilement des objets de presentation complexe.

Cependant, le modele d'interaction d'InterViews, qui ressemble fortement a celui de la X Toolkit, est moins convaincant. S'il est possible de fabriquer facilement des representations pour des donnees, il est impossible de faire reagir directement ces representations aux actions de l'utilisateur. C'est au niveau de l'interacteur qui les contient qu'il faut gerer ces actions. Une partie importante de la gestion des entrees est donc a la charge du programmeur de l'application. Les objets graphiques structures permettent de simplier cette tâche, mais cela ne contribue pas a donner une structure claire aux programmes interactifs. Les menus offerts par InterViews sont revelateurs a ce sujet : ils n'utilisent pas les capacites de dessin structure, mais sont programmes \a la main" selon un modele de dessin direct.

Une nouvelle version d'InterViews en cours de developpement (version 3) devrait corriger ce defaut, grâce a l'introduction des *glyphes*. Ces nouveaux objets de base permettent d'integrer la hierarchie des objets graphiques et celle des interacteurs. Par exemple, les interacteurs seront des glyphes qui savent gerer des evenements. Il est prevu que d'autres glyphes plus simples aient ces mêmes capacites de gestion des evenements. De cette maniere, il devrait être possible de gerer les evenements au niveau des objets graphiques.

3.2.3 ET++

ET++ [Weinand et al. 88 ; Weinand et al. 89] est un ensemble de classes C++ et d'outils pour la construction d'applications, similaire a l'environnement de Smalltalk [Goldberg 84]. Un grand nombre de ces classes sont destinees a la creation d'applications interactives. Leur but est de fournir des squelettes d'applications similaires a ce qu'offre MacApp [Shmucker 86].

ET++ offre un modele de dessin structure et un modele de gestion des evenements qui fonctionnent de maniere coherente. A la base des deux mecanismes se trouvent les objets visuels, qui ont une apparence graphique et recoivent des evenements. Ces objets visuels sont places dans des surfaces abstraites nommees *vues*. Pour visualiser le contenu d'une vue, il est possible de lui associer une fenêtre. Il est par ailleurs possible de sauver le contenu d'une vue dans un chier, ou de l'imprimer. Les objets visuels peuvent être simples, composites ou complexes. Les objets simples sont des textes, des lignes ou des images. Certains objets composites servent a regrouper d'autres objets visuels, et a gerer leur disposition et leur geometrie. D'autres sont des objets de base comme les boutons. D'autres enn gerent a la fois la disposition de leurs sous-objets et leur semantique. C'est le cas des objets *OneOfCluster*, qui permettent d'associer des

boutons en un ensemble de boutons-radio : un seul des boutons peut être enfonce a la fois. Enn, les objets visuels complexes servent a obtenir des interactions plus speciques. Ils sont derives des vues, qui sont elles-mêmes des objets visuels. On trouve ainsi des vues specialisees pour manipuler du texte, d'autres pour realiser des boîtes de dialogue, ou encore pour representer des arbres.

En ce qui concerne le lien entre l'interface et le noyau fonctionnel, ET++ va plus loin que la plupart des boîtes a outils, et propose de veritables squelettes d'applications. En effet, une application fabriquee avec ET++ contient generalement une instance d'une classe derivee de la classe *Application*. Une application realisee de cette maniere repose sur le modele de l'edition de documents. La classe *Application* contient des menus contenant des commandes de base (couper, coller, creation d'un nouveau document, etc.), et un ensemble de *documents*. Les documents correspondent aux abstractions contenues dans le noyau fonctionnel de l'application. Pour chaque type d'application, il faut donc denir de nouvelles classes de documents. C'est a ce moment que l'on determine quels types de vues sont associees a chaque type de document. De cette maniere, une vue est a la fois la Vue et le Contrôleur du modele MVC, tandis que le document represente le Modele.

L'association entre les actions de l'utilisateur et le noyau fonctionnel de l'application se fait a travers des *commandes*. Les commandes sont des objets qui sont envoyes au noyau fonctionnel lorsque l'utilisateur a execute une action. Il existe par exemple des commandes *ButtonCommand*, qui sont associees au declenchement d'un bouton ou d'une ligne d'un menu. Lorsqu'un objet visuel recoit un evenement, il fabrique une commande, puis l'envoie a son parent. Ce dernier peut traiter la commande, ou l'envoyer a son propre parent. Finalement, les commandes sont envoyees au document associe, qui peut alors executer des fonctions ou modier des donnees en consequence. Les commandes sont entreposees dans un historique. Ainsi, il est possible de defaire les dernieres commandes de maniere simple.

Les objets visuels d'ET++ ont des comportements predenis. Par exemple, lorsqu'on clique sur un bouton avec le bouton de gauche de la souris, une commande *ButtonCommand* est envoyee. Lorsque ce sont les autres boutons de la souris qui sont enfonces, une commande \nulle" est envoyee. Pour denir d'autres comportements, il faut fabriquer une classe derivee et redenir la fonction correspondante, soit par exemple *DoRightButtonDownCommand*.

Discussion

L'inconvenient majeur d'ET++ est son caractere statique. L'association entre une abstraction et sa representation se fait en denissant de nouvelles classes C++. De la même maniere, la denition du comportement des interacteurs se fait avec des classes derivees.

Ensuite, si le modele d'interaction d'ET++ est bien integre avec son modele de dessin, il n'est absolument pas extensible. A chaque type d'evenements du systeme de fenêtrage est associee une fonction particuliere de chaque objet visuel. Pour ajouter de nouveaux types d'evenements, il faut donc modier les classes d'objets visuels, ce qui est difcilement acceptable.

Enn, le dessin structure d'ET++ n'est pas completement convaincant. D'une part, il n'existe que tres peu d'objets de base (les lignes, les textes et les images), ce qui ne permet pas de realiser facilement les differents types de representations connus. Et surtout, beaucoup d'objets dits composites ne sont pas composes d'objets de base. Par exemple, les boutons et les menus ne sont pas fabriques avec des textes et des lignes.

ET++ est une bonne boîte a outils pour fabriquer rapidement de petites applications interactives. En revanche, les inconvenients que nous venons d'enumerer sont des obstacles aussi bien a la construction d'applications de grande taille qu'a celle d'outils pour la creation d'interfaces. En ce qui concerne ces derniers, il faut considerer que la creation de nouvelles interfaces passe par la creation de nouvelles classes C++. Un outil de creation d'interfaces devra donc fabriquer du C++, qui devra ensuite être compile avant la phase de test. Le cycle de developpement de l'interface sera donc inutilement ralenti. Par ailleurs, le programme produit sera difcilement adaptable ou modiable. Ensuite, ET++ comme MacApp encourage l'instauration de liens etroits entre la presentation et le noyau fonctionnel, sans aucun mediateur entre les deux. Les documents et les vues sont specialises en parallele, et se referencent explicitement. ET++ n'encourage donc pas au respect des principes de base de l'architecture des applications interactives.

3.3 X_{TV} : Principes de base

X_{TV} est une realisation commune du groupe Interfaces Homme-Machine du LRI, a l'Universite de Paris-Sud [Beaudouin-Lafon et al. 90 ; Beaudouin-Lafon et al. 91]. Son objectif principal est de servir de base a des recherches sur divers domaines de l'ingenierie des interfaces. Ainsi, X_{TV} a deja ete utilise pour des experiences sur les interfaces multi-utilisateurs et les interfaces gestuelles [Baudel 90]. Par ailleurs, il a servi de support a l'utilisation de contraintes dans la construction d'interfaces [Cournarie & Beaudouin-Lafon 91]. Enn, la boîte a outils pour interfaces animees que nous presenterons au chapitre 5 est une extension d'X_{TV}. Par ailleurs, X_{TV} a ete developpe dans le cadre du sous-projet AVIS/UIS du projet Eureka ESF. A ce titre, il est destine a fournir la base des services d'interface utilisateur de l'atelier de genie logiciel ESF. Une premiere application d'X_{TV} dans ESF a ete realisee : il s'agit de SemDraw, un editeur de graphiques utilisant des contraintes. X_{TV} est donc a la fois

une boîte a outils extensible, utilisable pour des buts de recherche, et une boîte a outils utilisable dans des conditions reelles.

Contrairement aux boîtes a outils que nous avons decrites plus haut, X_{TV} offre un modele de dessin structure et un modele de gestion des evenements homogenes, permettant ainsi de traiter l'interaction au niveau des objets presents a l'ecran. Ces modeles sont decrits aux sections 3.4 et 3.5. Le modele de dessin structure repose sur un modele de dessin direct dont nous allons decrire les grandes lignes.

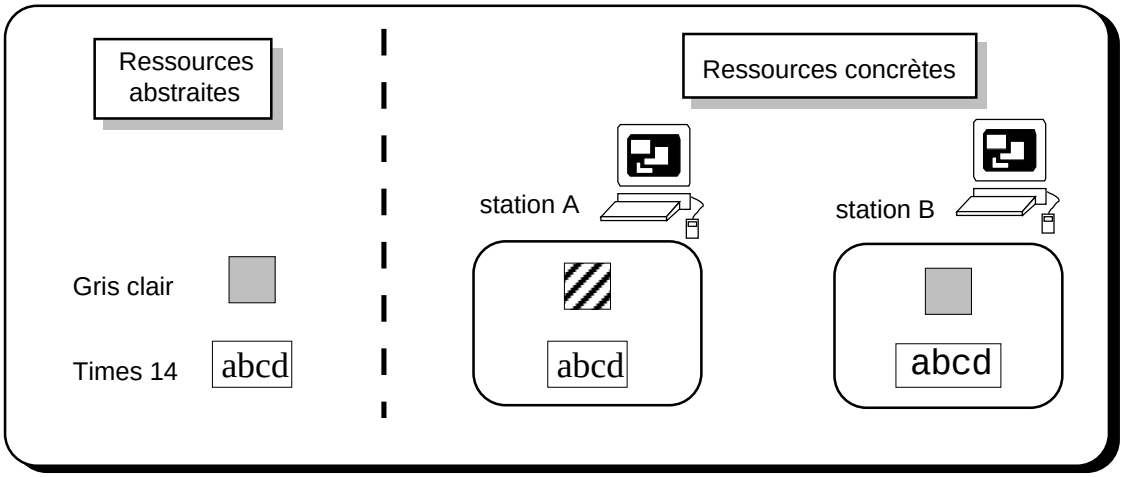

Figure 3.1 : Les ressources abstraites manipulees par le programmeur sont creees sur chaque machine selon ses possibilites. La station A possede un ecran noir et blanc, et la couleur grise y est simulee. La station B n'a pas la police Times, et une police par defaut est utilisee.

Tout d'abord, les operations de dessin font appel a des ressources propres a chaque ecran. Ces ressources sont par exemple des couleurs, ou des polices de caracteres. X_{TV} gere ces ressources sous la forme d'objets abstraits que l'on peut creer et detruire a volonte. Dans le cas le plus simple, la creation correspond a la lecture ou l'initialisation de la ressource par le systeme graphique sous-jacent. Par ailleurs, X_{TV} utilise ce mecanisme pour gerer plusieurs positions de travail (ecran et moyens d'entree) a la fois, tout en rendant cette gestion transparente pour le programmeur. Ce dernier cree et manipule une ressource abstraite, par exemple la couleur Mauve, et c'est cette ressource abstraite qui se charge automatiquement des operations d'initialisation, en fonction des ecrans sur lesquels elle est utilisee. Grâce a ces ressources abstraites, X_{TV} simplie a l'extrême le dessin sur plusieurs ecrans.

Comme support de son modele de dessin structure, X_{TV} utilise un modele de dessin direct sans etat. Le dessin est realise par des *objets graphiques*, dont l'aspect est parametre par des *environnements graphiques*. L'objet graphique est responsable de la

forme du dessin. C'est lui qui a l'initiative de l'operation de dessin. On trouve dans X_{TV} des objets graphiques \point", \segment", \polygone", \spline", \icône", etc. Un environnement graphique contient des attributs utilises par les objets graphiques pour se dessiner. Ces attributs sont en general des ressources abstraites comme des couleurs,

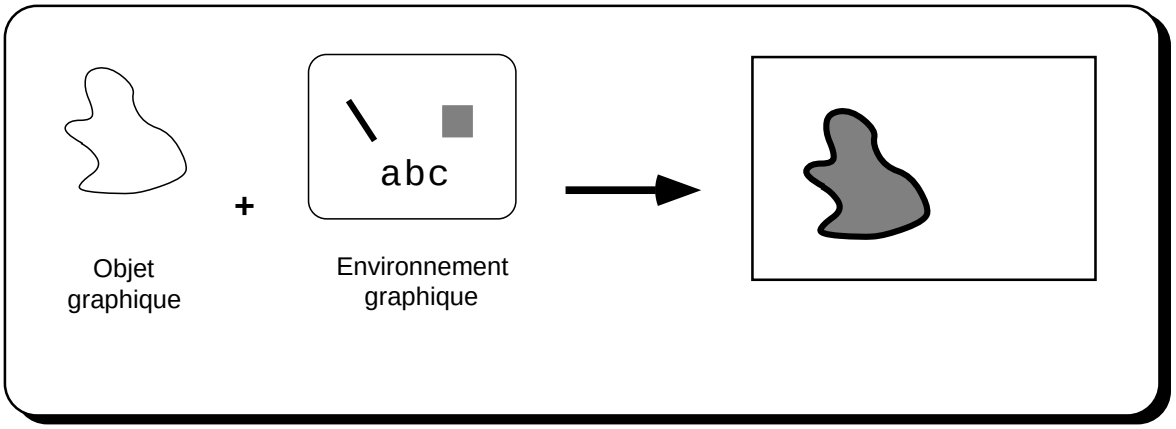

Figure 3.2 : Le dessin fait intervenir un objet graphique et un environnement graphique.

ou encore des nombres, comme ceux qui determinent l'epaisseur du trait. Chaque type d'objet graphique utilise un certain nombre d'attributs. Par exemple, un rectangle a besoin d'une couleur de bord, d'une couleur de fond et d'une epaisseur de trait. Il demande ces attributs a l'environnement qui lui est transmis lors de l'ordre de dessin. Si l'environnement ne possede pas cet attribut, une valeur par defaut est utilisee. Par ailleurs, le nombre et le type des attributs peuvent être etendus a volonte, en particulier pour denir de nouveaux types d'objets graphiques. Enn, il est possible d'organiser les environnements selon un arbre similaire a un arbre d'heritage. Un environnement \herite" les attributs de son pere, sauf s'il les redenit a son usage propre. Lorsque la valeur d'un attribut herite est modiee, elle l'est pour tous les environnements qui la partagent. Cela permet de gerer de maniere centralisee l'aspect de la presentation. On peut ainsi imaginer que tous les textes soient en permanence de la même couleur, quelle que soit leur police de caractere. En utilisant l'heritage d'attributs, il sufra d'une operation pour changer cette couleur.

3.4 Scenes, acteurs et vues

Le modele conceptuel d'X_{TV} est inspire du monde de la television. Il fait intervenir des *acteurs*, des *scenes*, des *cameras* et des *vues*. Le principe en est tres intuitif : des

acteurs sont presents sur une scene ; ils sont lmes par des cameras, et sont ainsi visibles dans des vues.

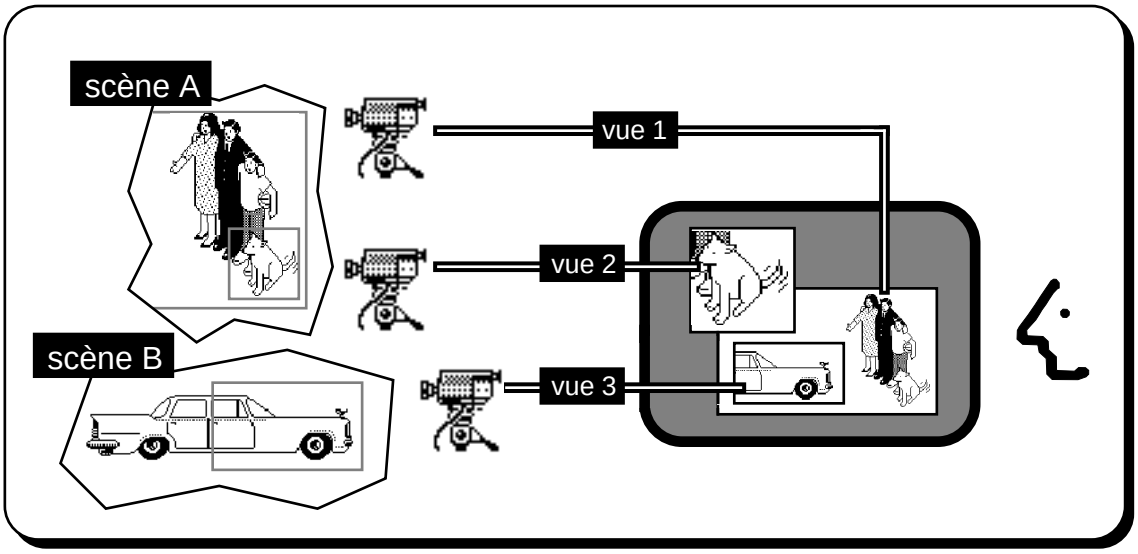

Figure 3.3 : Le modele d'X_{TV}.

Les vues sont des fenêtres ; elles encapsulent celles du systeme de fenêtrage. Ce sont donc des surfaces de dessin concretes.

Les scenes sont des espaces abstraits dont on visualise une partie du contenu. Elles sont en cela similaires a des surfaces d'afchage virtuelles. Cependant, la scene n'est pas une surface de dessin dont une partie est visible. Ce n'est pas elle qui decide de la maniere dont les acteurs sont dessines. Le dessin est realise par une collaboration entre l'acteur et la vue. De cette maniere, une même scene peut donner lieu a des representations differentes des mêmes acteurs, selon la vue dans lesquels on les represente. Par exemple, il est possible d'avoir des scenes contenant des acteurs en 3 dimensions. Selon l'angle des cameras, differentes vues montreraient differents aspects des acteurs. De maniere plus modeste, les cameras permettent dans l'implementation actuelle de modier le point de vue et le facteur d'echelle. Cela permet des vues plus ou moins detaillees comme dans la gure 3.3.

Plusieurs cameras peuvent lmer une même scene. Il est donc possible de visualiser les mêmes acteurs dans plusieurs vues a la fois. Cette possibilite est utile par exemple pour certaines applications de dessin. On peut ainsi effectuer un zoom sur une partie du dessin, tout en conservant une vue d'ensemble a l'echelle normale. On peut aussi utiliser cette possibilite pour realiser des applications multi-utilisateurs simples, qui montrent les mêmes objets a tous les utilisateurs. Il suft pour cela d'ouvrir une vue sur l'ecran de chaque participant, dans la mesure ou X_{TV} gere de maniere transparente l'utilisation de plusieurs positions de travail.

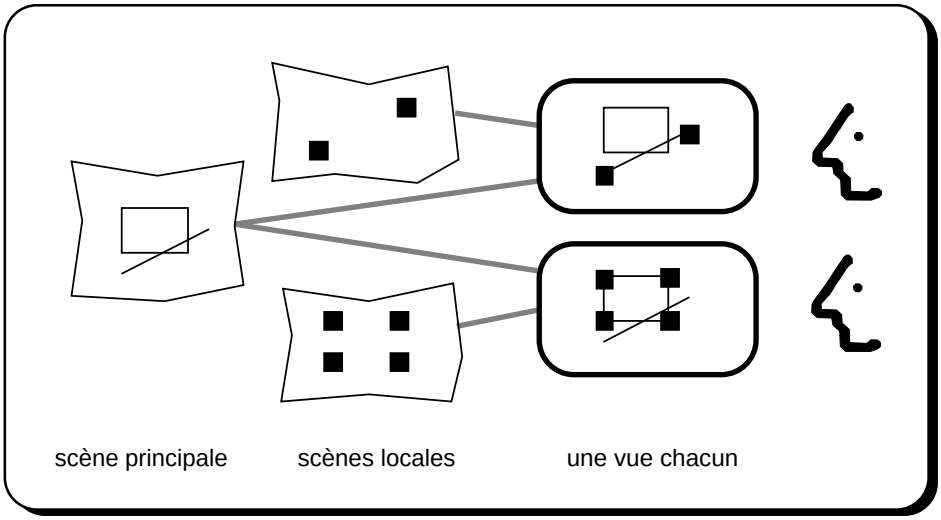

Figure 3.4 : Un effet special : visualiser plusieurs scenes dans une vue.

Cependant, les applications interactives pour plusieurs utilisateurs montrent rarement exactement la même chose a chacun. En particulier, chacun a besoin de retour d'information (feed-back) sur les operations qu'il est en train d'effectuer. Cela passe en general par des dessins supplementaires, appeles *poignees*. Pour cela, le modele d'X_{TV} permet un effet special : la superposition de plusieurs images. Une vue peut recevoir les images de plusieurs scenes a la fois. De cette maniere, chaque utilisateur peut voir une scene locale superposee a la scene generale, et contenant des acteurs exclusivement destines a l'interaction. Dans ce cas, la gestion des relations entre les acteurs generaux et les acteurs locaux est a la charge du programmeur. La gure 3.4 represente une telle situation.

Les acteurs sont des objets destines a être places dans des scenes, et dessines dans des vues. Ils sont le plus souvent composes d'un objet graphique et d'un environnement graphique. Toutefois, il leur est seulement demande de connaître leur geometrie et de savoir se dessiner dans des vues, ce qui autorise de nombreuses combinaisons. En particulier, les acteurs de base d'X_{TV} peuvent partager le même objet graphique ou le même environnement a plusieurs.

3.5 Gestion des entrees

Le modele de gestion des entrees d'X_{TV} est une de ses caracteristiques les plus originales. Il est sufsamment general pour permettre les modes d'interaction les plus varies, ainsi que l'introduction de nouveaux peripheriques. Ce modele distingue trois types d'entites qui interviennent dans le traitement d'un evenement. Ce sont le *dispositif*

qui l'emet, l'*objet reactif* qui en est la cible, et les *ltres*, qui determinent les operations declenchees par l'evenement. Une notion supplementaire a ete introduite par l'auteur pour la gestion des interactions longues : les *actions*.

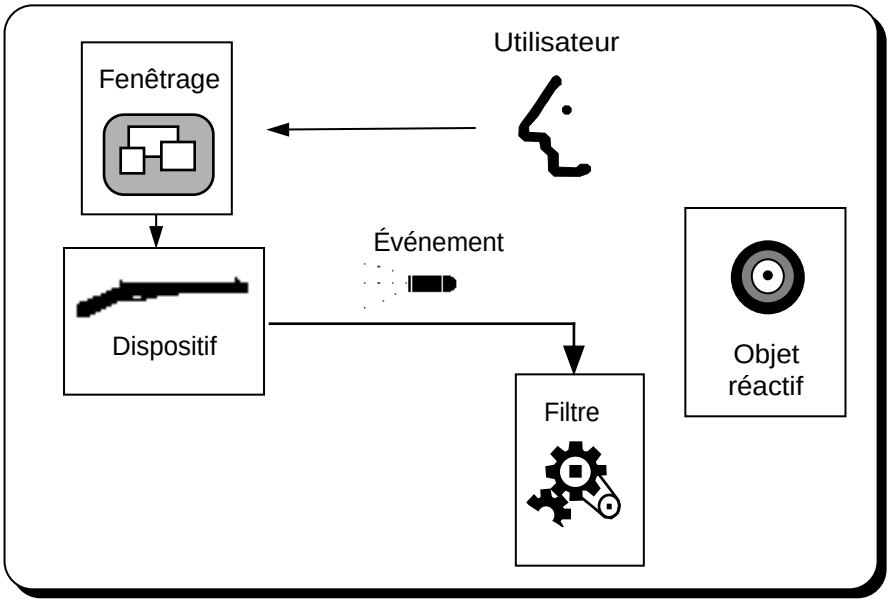

Figure 3.5 : Le cheminement des evenements.

Dispositifs

Les dispositifs sont des abstractions qui representent les diverses sources d'emission d'evenements. A chaque type de dispositif correspondent des types d'evenements. La souris, par exemple, emet des evenements relatifs a ses deplacements ainsi qu'a l'action de l'utilisateur sur ses boutons. Un dispositif n'est pas necessairement lie a un peripherique d'entree. Par exemple, une partie du systeme de fenêtrage est destinee a produire des evenements relatifs a la gestion des fenêtres : changements de taille, necessite de redessiner une zone, etc. Il existe donc un dispositif specique qui correspond a ces evenements. On trouve de fait trois classes de dispositifs predenis : souris, clavier, et systeme de fenêtrage. Chaque classe a une instance unique par ecran, qui represente le peripherique reel.

Il existe d'autres types de dispositifs. Tout d'abord, certaines interfaces font appel a des materiels supplementaires. On rencontre des potentiometres, des tablettes graphiques ou des systemes de reconnaissance vocale par exemple. Ces materiels peuvent être representes par des dispositifs. Il emettent des evenements qui leur sont propres : changements de valeur, deplacements du stylo, ou ordres vocaux. On peut aussi creer des dispositifs entierement simules, tels qu'une horloge, ou un bouton dessine a l'ecran

qui envoie des evenements quand l'utilisateur clique dessus. L'utilisation de tous ces dispositifs est uniforme, de telle maniere qu'il est possible de remplacer un dispositif simule par un dispositif reel (ou reciproquement) sans grand effort.

Objets reactifs

Les objets reactifs sont les cibles des evenements. De maniere intuitive, on peut dire que ce sont eux qui les recoivent. Cependant, ce ne sont pas eux qui reagissent directement aux evenements, mais plutôt les ltres, que nous examinerons plus loin. Le rôle d'un objet reactif est de determiner si un evenement lui est destine. En fait, la premiere phase du traitement d'un evenement est la determination de sa cible. Celle-ci est parfois connue des la creation de l'evenement, surtout lorsqu'il a ete emis par un dispositif abstrait. Mais le plus souvent, il faut examiner les objets contenus dans une vue. C'est le cas lorsque l'utilisateur clique dans une fenêtre, par exemple. Les principaux objets reactifs sont les acteurs et les scenes. En general, les acteurs decident s'ils interceptent un evenement en fonction de leur geometrie et de l'endroit de l'ecran ou s'est produit l'evenement, si cela a un sens. Quant aux scenes, elles recoivent normalement les evenements dont aucun acteur n'a voulu. En fait, c'est la scene qui determine lequel de ses acteurs est la cible. Dans les scenes les plus generales, cela se fait en demandant successivement a chacun s'il est interesse. Dans des situations plus particulieres, la scene peut directement calculer la cible. C'est le cas lorsque les acteurs sont places a des positions bien determinees, sur une grille par exemple.

Filtres

Les ltres sont les objets qui gerent les evenements. Un ltre est associe a un ensemble de types d'evenements, et a un ensemble d'objets reactifs. On peut par exemple imaginer un ltre associe aux evenements issus du clavier, pour toutes les icônes d'une interface iconique. Lorsqu'un tel evenement se presente sur une icône, le ltre est active. C'est donc en denissant le comportement des ltres que le programmeur peut denir le comportement de l'interface. Il dispose pour cela des informations transportees par les evenements, ainsi que de leur cible.

Il existe des ltres de base, qui permettent d'associer une fonction a un evenement, sur le même modele que les \callbacks". De maniere plus generale, le programmeur peut denir de nouveaux ltres par derivation ; il doit pour cela redenir la fonction qui est appelee lorsqu'un ltre est active. Parmi les comportements les plus courants, on trouve l'appel de fonctions de l'application, ou la modication de l'aspect de l'interface. Un ltre peut aussi modier l'evenement qu'il est en train de traiter, ou même le remplacer par un autre evenement. Ce dernier mecanisme permet un traitement progressif des evenements, par rafnements successifs. L'exemple le plus

simple est la notion d'equivalent-clavier : l'enfoncement d'une touche est transforme en clic de la souris.

Actions

Les trois entites que nous venons de decrire sont la base du modele de gestion d'evenements d'X_{TV}. La notion d'action que nous allons introduire ici n'est pas du même niveau : elle repond a un probleme particulier, celui des interactions longues, et est construite au-dessus des trois notions fondamentales, dispositifs, objets reactifs et ltres.

Certaines interactions entre l'utilisateur et l'ordinateur ne se resument pas a un evenement isole. L'enfoncement de la touche 'q' pour quitter une application est clairement un evenement isole. Mais le deplacement d'une icône, comme la deformation d'un objet graphique dans un outil de dessin, correspond au traitement d'une serie d'evenements. S'il s'agit de l'enfoncement repetitif d'une touche (une des 4 touches portant une eche, par exemple), on peut a la rigueur considerer chaque evenement de maniere independante. Lorsqu'il s'agit de la sequence enfoncement-deplacement-relâchement de la souris, cela devient plus difcile. En effet, les evenements de deplacement et celui de relâchement ont une signication differente selon qu'ils sont ou non precedes d'un enfoncement sur un objet prêt a être deplace. De plus, dans le cas de l'enfoncement successif de touches comme dans celui du deplacement avec la souris, il faut se souvenir de l'objet qu'on est en train de deplacer.

Les boîtes a outils proposent rarement de solution a ce probleme des interactions longues, et il est plus difcile a resoudre qu'il n'y paraît. La solution la plus immediate consiste a placer le programme dans un mode particulier. Cela resulte parfois en une boucle locale de lecture des evenements jusqu'a la n de l'action, comme dans la Toolbox du Macintosh. Cette solution est a rejeter, car elle interdit de mener plusieurs interactions en parallele et conduit a des programmes mal structures. On peut aussi stocker quelque part l'etat de l'interaction et l'objet en cours de deplacement. Mais il reste a determiner ou stocker ces informations. L'experience prouve qu'aucun objet existant n'est un candidat satisfaisant pour les conserver. La raison apparaît clairement dans le cas d'une interface multi-utilisateurs ou multi-modale : plusieurs actions identiques peuvent se produire simultanement. En particulier, plusieurs objets peuvent être deplaces en même temps.

Pour resoudre ce probleme, X_{TV} introduit des objets \action". Une action existe pour chaque interaction en cours, et est detruite lorsque l'interaction s'acheve. Ce sont les actions qui conservent les informations liees a l'interaction. Dans le cas d'un deplacement avec la souris, ce sont l'objet deplace et la position relative de la souris par rapport a cet objet au cours du deplacement. Par ailleurs, les actions permettent de clarier la situation vis-a-vis des dispositifs. Considerant qu'en general une seule

action est en cours par dispositif, chaque action est un objet reactif qui recoit tous les evenements emis par son dispositif associe. Pour des interactions multi-modales, on peut imaginer des actions qui recevraient les evenements de plusieurs dispositifs. Notons que les actions d'X_{TV} sont tres differentes des commandes d'ET++, qui sont des objets emis vers le noyau fonctionnel a la n d'une interaction. Les deux mecanismes s'adressent a des problemes differents, et peuvent coexister. Lorsque le debut d'une interaction longue est detecte (selon des criteres plus ou moins simples), une action est instanciee ; lorsque l'interaction est terminee (la encore, les criteres sont a determiner selon le type d'interaction), l'action est detruite, et une commande est instanciee et propagee vers le noyau fonctionnel. On peut aussi imaginer que des commandes soient emises au cours de la vie de l'action : si dans une application similaire au Finder la poubelle est remplacee par un aspirateur que l'on deplace sur des chiers, des commandes de destruction vont être emises au cours de l'action de balayage.

Le mecanisme des actions permet de gerer clairement des formes d'interaction comme le clic ou le pointer-tirer, qui echappent en general aux boîtes a outils. Ainsi, une extension d'X_{TV} denit un dispositif \souris de haut niveau", qui emet de tels evenements synthetiques. Ces evenements sont produits par des actions specialisees, qui ltrent les evenements de plus bas niveau. Les evenements emis sont de type *Click* (si aucun mouvement n'est detecte), *DragStart*, *DragMove*, ou *DragEnd*. En general, on utilisera *DragStart* pour creer un objet rendant compte du mouvement : l'ombre de l'objet qu'on veut deplacer, par exemple. *DragMove* permettra de faire evoluer cet objet transitoire, et *DragEnd* de le detruire et d'effectuer une commande. Par ailleurs, ces actions gerent aussi tous les cas particuliers comme l'entree et la sortie dans une fenêtre au cours de l'interaction.

3.6 Un exemple d'utilisation

Nous allons maintenant examiner un exemple simple d'application realisee avec X_{TV} : un jeu de reussite. Il s'agit d'un jeu de cartes ou toutes les cartes sont etalees sur la table sur une grille de 4 sur 14. A chaque instant, il existe quatre positions libres. Les seules operations possibles sont des deplacements de cartes vers ces positions libres, selon certaines regles.

Le noyau fonctionnel de notre application est mis en uvre par deux classes d'objets : les cartes, et le jeu. Les cartes connaissent leur rang et leur couleur. Quant au jeu, il contient un tableau de pointeurs sur des cartes, et des donnees liees au deroulement du jeu. Il sait tester si deux cartes peuvent être permutees, et les permuter. Il sait aussi s'initialiser en disposant aleatoirement les cartes.

L'interface graphique adaptee a ce jeu est evidente : il s'agit de representer toutes les cartes, et de permettre a l'utilisateur de les deplacer avec la souris. Pour cela,

nous allons associer un acteur a chaque carte. Le jeu sera associe a une scene. Pour simplier, procedons par heritage multiple, et fabriquons une classe acteur-carte, et une classe scene-jeu. Les acteurs doivent savoir se dessiner. Pour cela, les acteurs-cartes contiennent chacun une icône a leur image. Selon leur couleur, ils se dessinent en rouge ou en noir. La scene-jeu, elle, doit savoir determiner la cible des evenements. Nous sommes dans une situation ou la recherche de la cible peut être optimisee par rapport a celle qui est fournie par defaut. Pour cela, la scene-jeu calcule par simple division euclidienne dans quelle case de la grille l'evenement s'est produit. La cible est alors l'acteur-carte qui se trouve dans la position correspondante du tableau.

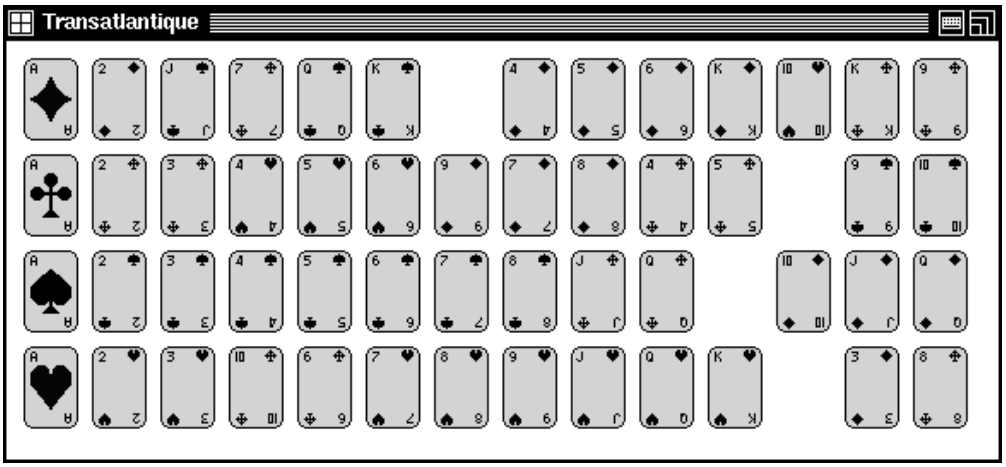

Figure 3.6 : Le jeu de reussite realise avec X_{TV}. Les cartes sont deplacees par manipulation directe avec la souris.

Venons-en maintenant au comportement de l'interface. L'utilisateur peut cliquer sur une carte et deplacer la souris. Pendant le deplacement, une image representant la carte est attachee au curseur. Finalement, il relâche le bouton de la souris. Il faut alors determiner s'il a lâche la carte a un endroit autorise. Si c'est un mouvement legal, on effectue la permutation. Ce comportement est obtenu grâce a trois ltres. Le premier est associe a l'enfoncement des boutons sur les cartes. Il change l'aspect de la carte selectionnee (changement de l'environnement graphique), et cree une action, qui elle-même cree une image prête a être deplacee. Le deuxieme ltre est associe a l'action, pour traiter les deplacements de la souris avec un bouton enfonce. L'image est deplacee en consequence. Enn, le troisieme ltre est associe au relâchement du bouton sur les cases vides. Le plus simple pour cela est de representer ces cases vides par des acteurs invisibles. Ce ltre demande au jeu de verier si la permutation est autorisee. Si c'est le cas, il lui demande d'effectuer la permutation. Dans le même temps, l'action s'auto-detruit, et l'image transitoire disparaît.

Le programme que l'on construit avec X_{TV} sur la base de cette description est relativement court. L'annexe A en donne les passages les plus signicatifs.

3.7 Conclusion

Nous avons presente dans ce chapitre la boîte a outils X_{TV}. Nous avons vu comment ses mecanismes de dessin structure et de gestion des evenements permettent la construction aisee d'interfaces a manipulation directe. De plus, son extensibilite en fait un support privilegie pour la recherche sur les interfaces. En revanche, X_{TV} n'echappe pas au defaut des boîtes a outils : il n'y existe pas de veritable mecanisme de communication avec le noyau fonctionnel. On court ainsi le risque de voir des applications ou cette communication est mal realisee. Il ne faut donc pas perdre de vue le besoin d'outils et surtout de modeles pour denir cette communication d'une maniere compatible avec le fonctionnement des boîtes a outils.

Enn, X_{TV}, comme toutes les boîtes a outils, ne permet de construire facilement que des interfaces dont l'apparence est statique. L'appel de fonctions de dessin y est remplace par la creation d'objets graphiques, mais le comportement dynamique de ces objets est toujours decrit par des fonctions, ce qui complique sa realisation. Les chapitres suivants sont consacres a la description de ce comportement dynamique.

Chapitre **4.**

Animation et interfaces

Nous allons dans ce chapitre nous interesser aux animations dans les interfaces. Les mecanismes des interfaces, que nous avons detailles dans les chapitres precedents, commencent a être bien connus. En revanche, la notion d'animation est moins bien denie, et nous devrons en construire une denition adequate. Un utilisateur peut en donner une denition intuitive : l'afchage evolue spontanement, ou tout au moins sans l'intervention de l'observateur. Helas, cette denition n'aide pas beaucoup les programmeurs d'interfaces. Pour eux, tout le probleme est d'identier ce qui se cache derriere le mot \spontanement". Quels sont les mecanismes, les entites logicielles qui provoquent et gerent le mouvement ? Ce sera par exemple une horloge dont les battements declencheront des mises a jour de donnees et les reafchages correspondants. Nous verrons qu'en fait la situation est plus complexe, et que les animations sont dirigees par des sources differentes selon les applications.

Nous essaierons donc dans ce chapitre de mieux cerner la notion d'animation, et d'identier les techniques necessaires pour realiser des interfaces animees. Pour cela, nous examinerons d'abord les diverses formes d'animation que l'on rencontre en informatique. Nous envisagerons ensuite les utilisations possibles de l'animation dans les interfaces homme-machine. A partir de ces observations, nous proposerons une description de ce qu'est l'animation dans une interface, et une classication des services que doit offrir un systeme d'animation. Ces observations nous ameneront aussi a une reexion sur l'architecture des applications interactives animees. Nous tenterons de montrer comment elles generalisent naturellement les interfaces plus traditionnelles, et comment il peut être protable de decrire ces dernieres selon les mêmes modeles. Nous examinerons ensuite des travaux lies a l'animation dans les interfaces, et la maniere dont ils permettent de decrire certaines caracteristiques des animations. Enn, nous nous livrerons a une reexion sur la description et la gestion du temps. Cela nous amenera a etudier des systemes permettant de decrire des phenomenes dynamiques, et en particulier les systemes de composition musicale.

4.1 Animation et informatique

Depuis que les ordinateurs ont des capacites graphiques satisfaisantes, on a vu se developper des applications utilisant ces capacites pour produire un afchage anime. Parmi ces applications, on rencontre en particulier l'animation d'images de synthese, inspiree par le dessin anime traditionnel ou liee a la visualisation scientique de phenomenes dynamiques. On rencontre aussi les jeux video, qui melangent animation et interaction. Il est aussi interessant d'examiner les tentatives recentes pour informatiser la production de dessins animes traditionnels, qui est longtemps restee manuelle.

4.1.1 Animation d'images de synthese

L'animation d'images de synthese est le resultat le plus spectaculaire de la rencontre entre animation et informatique. On peut resumer les applications de ces images animees en une phrase : montrer des objets ou des phenomenes qui ne sont normalement pas visibles. Cela va des objets qui ne sont pas accessibles a l'il ou la camera a ceux qui n'existent pas du tout, en passant par des entites qui ont une realite mais pas d'apparence. En general, les images de synthese representent des objets en 3 dimensions, pour obtenir un meilleur effet visuel, ou un plus grand realisme.

Le premier domaine ou ces images animees sont utilisees est la visualisation scientique. Pour les physiciens, les chimistes ou les archeologues par exemple, elles offrent un contact visuel avec les phenomenes, les objets, ou les animaux disparus sur lesquels ils travaillent. Cette nouvelle possibilite facilite leur comprehension, et parfois permet des decouvertes. On peut ainsi visualiser le comportement de molecules (objets inaccessibles), les turbulences de l'air autour d'un avion (phenomene difcilement visible), ou examiner le visage d'un homme prehistorique (objet hypothetique). Ces visualisations sont fabriquees de plusieurs manieres. En effet, les objets et les phenomenes que l'on visualise sont obtenus soit par des mesures, qui fournissent des jeux de donnees, soit par des modeles physiques qu'il faut exploiter par des simulations. Des techniques speciques ont ete developpees pour fabriquer des images de qualite en fonction des informations disponibles. On trouvera dans [Thalmann 90] l'etude de ces techniques, que ce soit la representation de volumes echantillonnes, celle de ots de uides, la visualisation de surfaces, ou l'animation de personnages.

Un autre domaine d'application est en evolution tres rapide : c'est la synthese d'images animees pour le grand public. En effet, les diverses formes de communication visuelle peuvent tirer parti de ces images. On peut penser au cinema, ou des societes comme Lucaslm utilisent des images de synthese pour leurs effets speciaux depuis le debut des annees 1980. Les effets speciaux de Terminator 2, qui melangent images reelles et images de synthese, ont fait l'objet d'une presentation a la conference SIGGRAPH'91. On peut aussi penser au dessin anime, avec par exemple l'illustration

des fables de Jean de La Fontaine par des animaux stylises [Fantôme 91]. Bien entendu, la publicite exploite elle aussi ces possibilites. Ainsi, de nombreuses publicites mettent en scene des objets de consommation a qui l'on a donne vie : une boîte de boisson gazeuse qui se contorsionne, par exemple. Enn, on peut envisager a terme l'usage de ces images de synthese animees pour \fabriquer" de l'information, avec toutes les derives possibles. Il semble cependant que l'animation realiste de personnages pose de serieux problemes, ce qui ecarte pour l'instant les lms ou l'on ferait dire ce qu'on veut a son rival, sans possibilite de discerner le faux du vrai. Fait amusant a noter, il semble que la tendance actuelle soit de produire des images visiblement synthetiques, pour donner une impression de modernisme.

Venons-en brievement aux techniques generales utilisees pour fabriquer ces animations. Le lecteur pourra se reporter a [Magnenat-Thalmann & Thalmann 85] ou [Thalmann 90] (d'ou les gures sont reprises) pour plus de details. Il faut tout d'abord noter que l'animation par ordinateur est fortement inuencee par le dessin anime traditionnel. Dans les deux cas, l'animation est obtenue par la succession d'images a un rythme determine a l'avance. Dans un cas, les images sont dessinees a la main ; dans l'autre elles sont calculees a partir de donnees et de directives. Avant de decrire la maniere dont sont decrites les images a fabriquer, notons que l'animation par ordinateur distingue toujours deux phases : la fabrication, puis la restitution. En effet, la complexite des dessins est telle qu'on ne peut calculer une image pendant le 1/25eme ou le 1/60eme de seconde qui la separe de l'image precedente. La forte croissance des puissances de calcul disponibles est compensee par la croissance de la complexite des dessins...

On peut distinguer trois techniques pour realiser des animations. La premiere consiste a fournir toutes les images, ce qui fait reposer le travail sur le dessinateur, ou le systeme de mesure. Cette technique, qui permet des animations \parfaites", est aussi tres coûteuse.

La seconde technique est celle des \images-cles[1]". Avec cette technique, on donne au systeme d'animation les dessins les plus signicatifs, et c'est lui qui fabrique les images intermediaires (les \inbetweens"). Plusieurs techniques ont ete inventees pour fabriquer ces images. D'une part, on trouve des techniques d'interpolation de formes. Pour cela, il faut fournir pour chaque forme apparaissant dans l'image-cle un ensemble de points-cles, et donner la correspondance entre ces points d'une image a l'autre. Les positions dans les images intermediaires sont calculees par interpolation. L'interpolation lineaire, illustree par la gure 4.1, est simple mais produit souvent des distorsions inacceptables. Des techniques d'interpolation plus evoluees sont utilisees pour eviter distorsions et discontinuites. Un autre moyen de fabriquer les images intermediaires est l'utilisation d'images-cles parametriques. Ici, les images-cles sont denies par un jeu de parametres. Pour l'animation d'un personnage, ces parametres

[1] keyframe animation

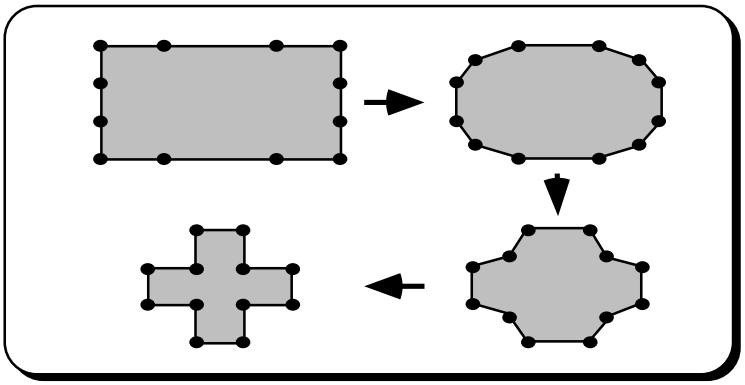

Figure 4.1 : Une animation par images-cles. Les positions des 12 points cles sont interpolees pour fabriquer les deux images intermediaires.

seront par exemple les angles des articulations. La technique consiste alors a calculer par interpolation les valeurs des parametres pour les images intermediaires. Les mêmes techniques d'interpolation que pour l'interpolation de formes peuvent être utilisees.

La troisieme technique de construction d'animations est la description procedurale, qui est equivalente a la simulation physique. Au lieu de decrire les images, on decrit l'evolution de leur contenu au cours du temps. Avec les techniques d'interpolation, un physicien qui veut representer la chute d'un objet dans le vide doit donner des positions echantillonnees. Ensuite, le systeme d'animation obtient les positions intermediaires par interpolation, mais il est peu probable que les positions calculees soient realistes. Avec une description procedurale, le physicien peut exprimer la loi exacte, comme dans la gure 4.2. On trouvera une recapitulation des techniques a base de lois physiques dans [Gascuel & Puech 91].

```
TIME = 0;
while Z > 0 do
    Y = 100 - 5 * TIME * TIME;
    MOVE_ABS (OBJECT, X, Y, Z);
    draw OBJECT;
    record OBJECT;
    erase OBJECT;
    TIME := TIME + 1/25;
```

Figure 4.2 : L'animation procedurale de la chute d'un objet.

L'avantage de l'animation procedurale est qu'elle permet des animations exactes. Neanmoins, elle suppose de connaître les lois d'evolution des objets, ce qui n'est pas toujours possible même pour les systemes physiques. Pour les systemes dont on ne

connaît pas la loi, mais dont on connaît les equations differentielles, on peut proceder par simulation, mais la aussi cela ne recouvre pas tous les cas. Dans la pratique, les animations sont donc obtenues par la combinaison des techniques procedurales et des techniques d'images-cles.

Nous avons detaille jusqu'a present les techniques de base pour fabriquer des images animees. Il existe aussi des methodes pour les structurer, an de reutiliser des sequences ou des objets. Pour cela, on utilise des notions inspirees du cinema. On considere d'abord qu'une sequence d'animation est composee de scenes, decrites par des scripts. Pour chaque scene, il existe des objets statiques (le decor) et des objets animes (les acteurs). Ces objets sont eclaires par des sources de lumiere, et lmes par des cameras. L'animation est alors obtenue par le deplacement des objets animes (par les techniques decrites plus haut), la variation des sources de lumiere, ou le deplacement de la camera. La description des objets, des deplacements et des synchronisations entre deplacements est appelee choregraphie. Les systemes d'animation permettent d'exprimer cette choregraphie dans les scripts. A partir de ces scripts, ils effectuent les calculs, et produisent les scenes d'animation.

4.1.2 Jeux video

Depuis le milieu des annees 1970, on voit proliferer de nouveaux types de jeux, reposant sur l'utilisation d'ordinateurs et d'ecrans graphiques. On les appelle jeux video ou parfois jeux d'arcade. On les rencontre sur des ordinateurs individuels, des consoles de jeux (des ordinateurs specialises), ou des consoles dediees a un jeu particulier et placees dans les lieux publics.

Ces jeux se repartissent essentiellement en deux categories. D'une part on trouve des jeux classiques tels les echecs, le tic-tac-toe ou les jeux de cartes, pour lesquels l'ordinateur joue le rôle de support, et parfois de partenaire. Ces jeux rentrent dans le cadre habituel de l'ajout d'une interface graphique a un noyau fonctionnel preexistant. La deuxieme categorie recouvre des jeux entierement nouveaux, reposant entierement sur l'interaction. En general, ils simulent une situation plus ou moins realiste, dans lequel le joueur intervient en temps reel. On trouve dans cette categorie les jeux de tennis et toutes leurs variations a base de raquettes et de balles, les simulations de pilotage d'automobile ou d'avion, les multiples versions de \Space Invaders", le desormais classique \PacMan", etc. C'est cette deuxieme categorie qui nous interesse ici.

On peut considerer que ces jeux d'arcade sont a de nombreux titres des precurseurs des applications interactives. Bien entendu, il s'agit d'applications tres speciques. Leurs techniques ne sont pas transposables immediatement a des applications interactives plus generales. Cependant, leur observation est riche d'enseignements.

Tout d'abord, les jeux video representent sans nul doute le plus grand nombre d'applications graphiques et le plus grand nombre d'utilisateurs. Ils se sont repandus et ont atteint une excellente qualite avant la banalisation des interfaces graphiques dans la bureautique. Ils touchent aussi un public tres divers, et surtout jeune. Le fait que des enfants s'amusent en les utilisant laisse penser que leur interface est bien adaptee.

Ensuite, les techniques d'interaction utilisees sont souvent novatrices. D'un point de vue materiel, les jeux repercutent tres vite les inventions. Par exemple, des jeux utilisant un gant tactile comme moyen d'entree ont rapidement ete disponibles. De même, les techniques de dialogue sont interessantes. On y trouve par exemple une notion de manipulation directe. En effet, les objets manipules sont en permanence visibles, et les actions de l'utilisateur immediatement repercutees (en revanche, elles sont rarement reversibles, mais c'est la l'interêt du jeu). D'ailleurs, Shneiderman mentionne abondamment les jeux video dans l'article ou il introduit la notion de manipulation directe [Shneiderman 83].

Par ailleurs, ces jeux sont dynamiques : ils utilisent intensivement l'animation. La balle du jeu de tennis est animee, comme le sont les envahisseurs du \Space Invaders". Souvent, l'animation est aussi utilisee pour le fond d'ecran. Parfois, le but est purement decoratif, mais il s'agit souvent de donner une illusion de mouvement. Pour cela, des techniques particulieres, reprises du monde de la video, sont employees. Par exemple, il est possible de donner une impression de mouvement en changeant seulement les couleurs d'un dessin immobile. D'autre part, l'animation est intimement mêlee avec les actions du joueur. Le joueur qui deplace sa raquette pour que la balle rebondisse dessus (gure 4.3) ou qui evite un projectile en ecartant son vaisseau, emploie une forme particuliere de dialogue avec l'ordinateur. L'etat du systeme y est modie en parallele par le mouvement propre des objets et les actions du joueur, et ce a une echelle tres ne. Nous verrons que ce type d'interaction est rarement possible avec les systemes d'animation disponibles actuellement. Si l'on observe differents jeux video, on le rencontre sous des formes diverses. La plus simple est le deplacement direct d'un objet par le joueur, de la même maniere que le curseur d'une souris. Parfois, le joueur ne fait qu'indiquer la direction ou la vitesse d'un objet. C'est ce qui se produit pour le personnage du PacMan. Une forme intermediaire consiste a ce que le joueur indique la position que doit prendre l'objet, mais que l'objet mette un certain temps a y parvenir. On la rencontre dans certains \Space Invaders", ou le temps de reaction du vaisseau est plus ou moins grand selon la difculte. Ces differentes formes donnent un apercu du lien qui peut exister entre l'animation et l'interaction avec l'utilisateur.

Les jeux sont donc d'excellentes sources d'inspiration pour ce qui est de l'interaction avec l'utilisateur. En particulier, leur maniere d'utiliser l'animation est tres instructive. En revanche, les jeux ne sont pas d'un grand secours en ce qui concerne les solutions techniques. Tout d'abord les techniques employees sont souvent speciques aux

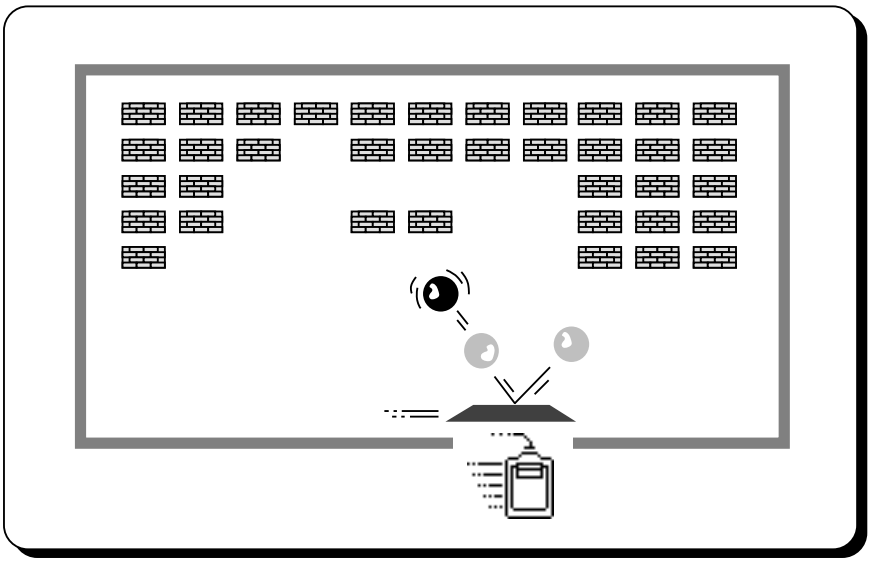

Figure 4.3 : Le jeu de casse-briques : l'utilisateur deplace la raquette pour faire rebondir la balle dessus. La balle doit heurter et detruire les briques.

jeux, et même parfois speciques a un jeu donne. Et surtout, les jeux sont avant tout des applications commerciales. Les techniques utilisees font rarement l'objet de publications par leurs constructeurs.

4.2 *Animation et interfaces*

Nous allons maintenant nous interesser aux applications de l'animation dans les interfaces homme-machine. Pour cela, nous allons envisager les circonstances ou l'afchage graphique evolue, sans prejuger de ce qui est ou n'est pas de l'animation. Certaines de ces applications existent deja, et sont mises en uvre par des techniques traditionnelles, ou developpees speciquement. D'autres applications sont nouvelles ou hypothetiques. Nous en demontrerons le plus souvent la faisabilite, et tenterons d'en illustrer l'interêt.

L'animation comme media

L'application la plus immediate de l'animation est de l'utiliser comme un moyen de communication, au même titre que la video, le cinema, ou les images animees decrites a la section 4.1.1. Par exemple, des systemes d'aide permettent d'illustrer le fonctionnement d'un outil ou d'une commande au moyen d'une animation. C'est le cas des icônes animees introduites experimentalement dans un outil de dessin sur

Macintosh [Baecker et al. 91]. Chaque icône de la palette d'outils (crayon, gomme, tire-ligne, etc.) peut s'animer pendant quelques secondes pour demontrer l'usage et l'effet de l'outil associe. Des animations explicites sont aussi utilisees dans des systemes de presentation d'information, tels des hypertextes. Lorsque par exemple on clique sur un personnage, il s'anime et raconte une histoire. Notons que ce type d'animations, comme la video ou certains effets sonores, n'est pas utilisable en tant que tel comme moyen d'interaction. En effet, le systeme ne fait que reproduire pendant un certain temps des informations determinees a l'avance. L'interactivite est faible, et se limite generalement a l'interruption possible de l'animation. Il existe des logiciels commerciaux, tels que MacroMind Director sur Macintosh, pour realiser de telles animations.

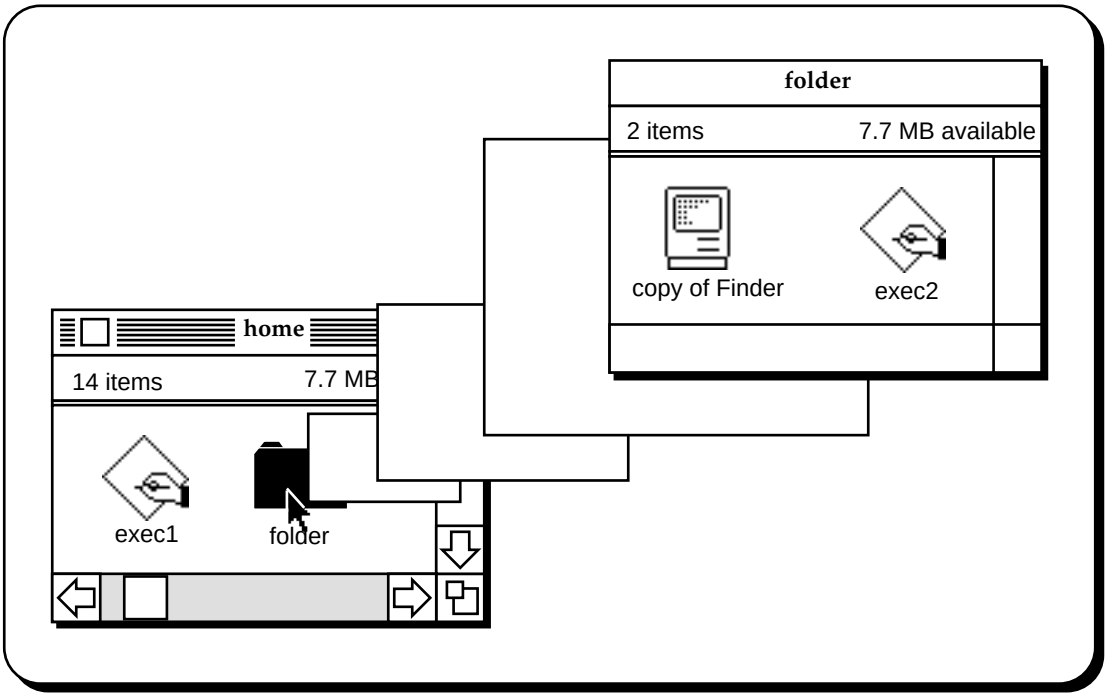

Figure 4.4 : Animation dans le Finder du Macintosh : L'utilisateur effectue un double-clic sur l'icône d'un repertoire. L'ouverture de la fenêtre est precedee par l'animation d'un rectangle.

L'animation comme artice

Une autre application de l'animation est beaucoup plus discrete. On peut même dire que son but est de se faire oublier. Il s'agit de l'utilisation d'animations comme transition entre deux etats de l'afchage. Ces animations sont generalement destinees a faciliter a l'utilisateur la comprehension de cette transition. Un exemple passe souvent inapercu : c'est l'ouverture d'une fenêtre a l'ecran, qui est souvent precedee par l'apparition

fugace d'un rectangle en train de grandir. Avec le Finder du Macintosh, ce rectangle part de l'icône sur laquelle on a effectue un double-clic, pour prendre la taille de la fenêtre en train de s'ouvrir (voir gure 4.4). Ce type d'animations est aussi utile dans des systemes qui afchent de grandes quantites de donnees, lorsque le centre d'interêt change. Par exemple, une equipe du Xerox PARC a propose l'utilisation d'arbres \coniques" representes en trois dimensions [Robertson et al. 91]. Lorsque l'utilisateur

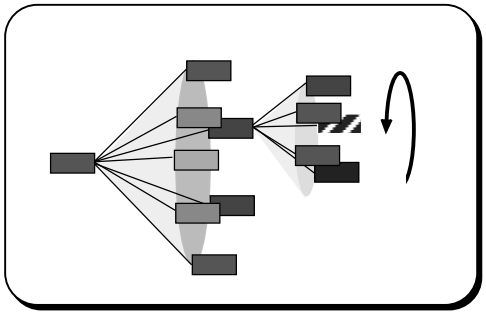

Figure 4.5 : Un arbre conique. Lorsqu'une feuille de l'arbre est designee (hachures), l'arbre tourne sur lui-même pour amener la feuille devant.

selectionne une donnee afchee sur cet arbre, celle-ci passe au premier plan. Plutôt que de passer brutalement d'une position a l'autre, l'arbre tourne progressivement sur lui-même. Les auteurs afrment que cette rotation progressive, malgre le delai qu'elle induit, permet de gagner du temps tout en diminuant la charge de travail de l'utilisateur. D'apres eux, elle fait reposer une partie de la tâche d'adaptation sur le systeme perceptif, evitant ainsi une importante activite cognitive pour se situer. De maniere similaire, on peut utiliser l'animation pour attirer l'attention de l'utilisateur. L'exemple le plus repandu en est le curseur clignotant des interfaces textuelles.

Interfaces multi-utilisateurs

Les animations ont des applications dans les interfaces multi-utilisateurs. En effet, un des problemes qui se pose pour ces dernieres est le degre d'independance de chacun des utilisateurs. Ce probleme ramene indirectement a un besoin d'animations-artices. Imaginons un outil de dessin reparti : deux utilisateurs travaillent sur le même dessin. Lorsque l'un d'entre eux deplace un rectangle, on ne souhaite pas necessairement que l'autre utilisateur voie tous les petits deplacements successifs. En effet, ceux-ci ne sont pas signicatifs pour lui, et peuvent detourner son attention. Par ailleurs, cela risque d'encombrer inutilement le canal de communication de donnees. On preferera donc avertir le second utilisateur seulement lorsque le deplacement est ni. Mais dans ces conditions, on obtient un saut brutal de la position initiale a la position nale. Pour eviter cela, on peut synthetiser une animation qui va de l'une a l'autre

position, sans tenir compte du chemin d'origine. Cette solution a ete experimentee avec succes [Beaudouin-Lafon & Karsenty 91].

Visualisation de donnees

Une autre utilisation de l'animation est la representation animee de donnees. Dans un certain nombre de domaines, on trouve des donnees variant avec le temps, et dont on veut donner une representation graphique. C'est le cas de la visualisation scientique, ou du contrôle de procede. On peut imaginer un reacteur chimique dont on veut representer la pression et la temperature par un cadran et un thermometre. Dans ce cas, on fait appel a des representations predenies, et plusieurs logiciels du commerce le permettent. Dans d'autres applications, il est impossible de s'en tenir a ces representations. Dans le cas du contrôle du trac aerien, par exemple, les donnees a representer sont des positions d'avions. Pour cela, on voudra utiliser une presentation cartographique, ou se deplaceront des dessins representant les avions. Ce type de representation est tres dependant du domaine d'application, et le nombre de representations possibles est donc immense. Des systemes comme DataViews de V.I. Corporation permettent de creer un certain nombre de representations, a partir de composants de base. Ces composants sont pour la plupart des objets graphiques dont on peut lier les caracteristiques geometriques a des donnees a representer.

Animation d'algorithmes

L'animation d'algorithmes est sans doute l'application de l'animation qui a ete la plus etudiee. Le but est la meilleure comprehension d'algorithmes informatiques. Prenons par exemple le tri d'un tableau d'entiers. Il existe de nombreux algorithmes de tri, avec des proprietes variees. Pour chaque algorithme, on souhaite representer son execution sous des angles varies : l'evolution du tableau au cours du temps avec les deplacements de valeurs, le nombre d'elements bien places, etc. La representation peut être differente selon qu'il s'agit d'enseigner un algorithme ou d'etudier ses proprietes. Nous reviendrons plus largement sur l'animation d'algorithmes au chapitre 6. Cependant, nous pouvons noter des maintenant qu'animer un algorithme ne revient pas seulement a animer des donnees. En effet, dans un tri de tableau, la permutation de deux valeurs n'est pas simplement la modication de ces deux valeurs. Il faut donc representer l'operation \permutation", et pas seulement l'evolution des valeurs. Animer un algorithme revient donc a animer des donnees et les operations sur ces donnees, ce qui en fait un probleme assez complet. Toutefois, il s'agit surtout de presentation d'informations, et les systemes d'animation d'algorithmes ne permettent que rarement l'interaction sur les presentations.

Interfaces vivantes

Dans la plupart des utilisations que nous avons mentionnees, l'interface permet l'animation an de transmettre des informations, pour le compte de l'application ou d'un autre utilisateur. Mais on peut aussi imaginer d'utiliser l'animation dans l'interface elle-même : l'animation au service de l'interactivite. C'est ce qui se produit pour les jeux video. Dans ces jeux, il est difcile de tracer une frontiere entre interface et noyau fonctionnel : tout concourt a l'interaction. Le joueur qui envoie un projectile a-t-il commence une interaction supportee par le projectile, et qui se terminera lorsque celui-ci explosera, modiant alors l'etat du systeme ? Ou bien a-t-il deja modie l'etat du systeme en rajoutant un projectile ? Du point de vue de la realisation, les deux peuvent se justier. Cependant, la premiere approche semble plus fertile. Elle introduit l'idee d'objet d'interaction anime. On peut transposer l'idee dans les interfaces plus traditionnelles. Quel utilisateur du Finder n'a pas pense que la poubelle etait bien eloignee, et qu'il serait plus rapide de lancer l'icône plutôt que de l'amener si loin ? Un des principes des interfaces est de proposer des metaphores d'interaction inspirees du reel. Le monde reel autorise les mouvements, pourquoi pas les interfaces ?

Cette reexion amene a se poser des questions sur les interfaces d'aujourd'hui. On dit que les interfaces a base de fenêtres et d'icônes sont efcaces parce qu'elles rappellent la surface d'un bureau. Mais l'association entre les fenêtres et les feuilles de papier est-elle due a leur seule forme rectangulaire ? Le fait qu'on puisse les deplacer et changer leur ordre d'empilement y est sans doute aussi pour beaucoup. Il ne suft pas qu'un objet graphique ait l'*apparence* d'un objet du monde reel, il faut aussi qu'il en ait le *comportement*. William Gaver prend une position similaire en disant que les objets des interfaces doivent tenir leurs promesses et se comporter comme leur apparence le suggere [Gaver 91]. Il utilise pour cela la notion de *potentialites*[2]. Les potentialites sont les caracteristiques visibles d'un objet qui rendent intuitives les manipulations possibles sur cet objet. Par exemple, l'ombre portee d'une icône evoque la possibilite de deplacer cette icône. Nous ne faisons ici qu'elargir ce propos a des comportements dynamiques. Si on peut deplacer un objet, pourquoi ne pourrait-on pas le lancer ? Si une icône represente un outil de compression par une presse, pourquoi n'ecrase-t-elle pas les chiers qu'on lui presente ?

Dans un esprit similaire, le philosophe Pierre Levy propose une autre application des interfaces animees : les ideogrammes dynamiques [Levy 91]. Son opinion est que les ordinateurs nous offrent une opportunite de faire evoluer l'ecriture. Il propose pour cela d'utiliser des objets susceptibles de s'animer (les ideogrammes dynamiques), et que le \lecteur" pourrait manipuler. De cette maniere, la lecture se transformerait en exploration, et l'ecriture en \mise en scene". Les ideogrammes dynamiques peuvent aller du plus simple, par exemple un objet que l'on peut examiner sous tous ses

[2] affordances

angles en le faisant pivoter, au plus complexe : un ensemble d'objets animes de leur mouvement propre, et dont les interactions tant entre eux qu'avec l'utilisateur sont porteuses d'information. P. Levy parle alors de connaissance par la simulation. Les ideogrammes dynamiques, s'ils sont peut-être utopiques, permettent cependant de prendre conscience d'une autre application pour les interfaces animees. Il ne s'agit plus en effet de communication entre un humain et un programme, mais entre un auteur et un lecteur humains, de la même maniere que les hypertextes utilisent les techniques des interfaces statiques pour la communication entre humains.

Les applications de l'animation sont donc nombreuses. Elles sont egalement variees dans leur forme. L'animation d'une representation en fonction d'une variable est tres differente de l'animation d'une icône qu'on lance dans la poubelle. La section suivante est consacree a l'analyse de ces applications, an d'identier points communs et differences.

4.3 Nature et besoins de l'animation

A partir des applications mentionnees a la section precedente, tentons maintenant de formaliser la notion d'animation et d'identier ses caracteristiques.

Animation en temps reel

Une premiere remarque porte sur la nature de la tâche. Pour l'animation d'images de synthese, nous avons souligne que les images animees etaient construites au cours d'un processus complexe, puis utilisees a de nombreuses reprises avec un simple mecanisme de reproduction. Les applications que nous avons decrites ne peuvent pas toutes fonctionner selon ce principe. En effet, l'execution d'une application interactive se deroule de maniere imprevisible, des que l'utilisateur a l'initiative du dialogue. Par ailleurs, le nombre d'etats possibles de l'afchage est tel qu'il est difcile de les prevoir tous a l'avance, même si on accepte de les reproduire dans un ordre arbitraire. Pour des applications utilisant l'animation comme simple media, il est certes possible de faire deler des images calculees a l'avance. Mais dans le cas general, ces images ne seront qu'une partie de l'interface, le reste devant être calcule en temps reel. En consequence, l'animation dans les interfaces sera une tâche a realiser en temps reel, et devra posseder des mecanismes de structuration permettant d'incorporer des animations en differe.

Animer des objets graphiques

L'animation passe par des transformations de l'afchage. Par la suite, nous aurons a etudier la maniere dont ces transformations sont provoquees et decrites. Mais il

faut pour cela choisir en quels termes on va decrire l'afchage et ses transformations. On pourrait choisir de decrire l'afchage comme une image denie point par point, et l'animation serait obtenue par la succession de ces images. Neanmoins, cela ne correspond pas aux besoins des interfaces graphiques. Rappelons que la structuration de l'image est necessaire a la fois pour etablir des liens entre les donnees a representer et des parties de l'image, et pour permettre la designation de ces parties. L'image doit donc être structuree, et il paraît raisonnable de disposer d'une structuration en objets graphiques, comme pour les interfaces graphiques traditionnelles. La maniere la plus simple de decrire l'animation est alors de l'exprimer en termes de transformations de ces objets graphiques.

Examinons maintenant quels types de transformations peuvent se produire sur des objets graphiques. On pense tout d'abord a des deplacements. Par exemple, l'index d'une barre de delement est un rectangle qui se deplace. Mais il faut aussi prevoir des deformations : un thermometre est constitue de deux rectangles dont l'un change de taille. Les deformations peuvent être plus complexes pour des objets comme des

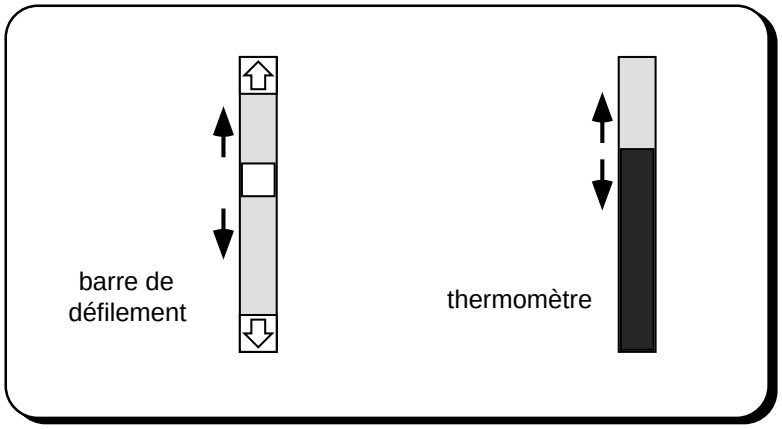

Figure 4.6 : Deux transformations geometriques.

polygones, a qui l'on peut ajouter ou supprimer des sommets. On peut cependant rassembler toutes ces modications sous le terme de transformations geometriques. D'autres transformations peuvent affecter l'aspect des objets, et non plus leur forme. Ce sont par exemple des transformations de couleurs. On realise un clignotant simple avec un rectangle qui change de couleur. On peut imaginer d'autres transformations d'aspect, liees a l'epaisseur des traits, aux polices de caracteres, etc.

D'autres types de transformations interessent autant les objets graphiques que la structure d'afchage elle-même : ce sont les creations et destructions d'objets. On peut par exemple realiser une animation en creant successivement des points alignes sur une droite : toutes les secondes, un nouveau point apparaît a une certaine distance du precedent. Ces transformations radicales doivent donc pouvoir être decrites dans

un systeme d'animation. D'autres transformations sont plus delicates : ce sont les changements de nature des objets, les mutations. On peut imaginer un rectangle qui se transforme en polygone, avant de se deformer en losange. Dans le cadre d'un langage a objets, ces transformations sont difciles a exprimer sans construire un mecanisme a objets plus souple, au-dessus du langage lui-même. Un tel mecanisme de prototypes a ete etudie recemment [Zeleznik & al. 91], mais il serait sans doute plus facile a realiser avec un langage a prototypes comme Self [Ungar & Smith 87]. Avec un langage a objets, on peut dans un premier temps considerer la mutation comme une destruction suivie d'une creation.

Enn, d'autres modications de la structure d'afchage peuvent être utilisees pour realiser des animations. On peut penser aux changements d'ordre dans cette structure : un objet passe devant, puis derriere un autre. Il est aussi possible de changer de point de vue sur la structure d'afchage, comme dans l'animation d'images en 3 dimensions. Ici, cela peut prendre la forme d'une translation de tous les objets. Par exemple, on peut, comme dans un dessin anime, terminer la chute d'un objet par une \vibration" de l'ecran, realisee par de petites translations rapides. On peut en fait constater que toutes les operations habituellement realisees sur les objets et la structure d'afchage sont susceptibles d'être utilisees pour l'animation. Un systeme d'animation pour les interfaces devra donc choisir les operations les plus utiles, sous peine de devenir tres complexe.

Decrire des trajectoires

Nous avons etabli qu'on pouvait ramener la description de l'animation a des transformations d'objets graphiques et de la structure d'afchage. Certaines de ces transformations sont ponctuelles et ont des descriptions speciques, comme la destruction d'un objet par exemple. Mais les transformations geometriques ou les transformations d'aspect ont un caractere plus general. Il est donc souhaitable de fournir un formalisme permettant de les decrire.

Un point de vue qui paraît fructueux est celui qui ramene ces descriptions a des descriptions de trajectoires. Pour cela, il faut considerer que des evolutions geometriques telles que deplacements ou deformations sont decrites par des grandeurs geometriques simples. Ce sont souvent des suites de points, qui proviennent en general de l'echantillonnage d'une courbe. Ce sont parfois des nombres, quand on decrit l'evolution du rayon d'un cercle par exemple. On peut aussi considerer que les transformations d'aspect sont decrites par des suites de donnees. On utilisera par exemple une suite de couleurs pour un effet de fondu. Que ces couleurs soient exprimees par trois coordonnees (rouge, vert, bleu par exemple) ou par leur nom, ce sont des elements d'un ensemble, et la transformation est decrite par une suite d'elements de cet ensemble. Pour cette raison, il semble judicieux d'assimiler la

description d'une evolution a celle d'une trajectoire dans un espace de points, de couleurs, ou de tout autre type d'objets. On peut alors utiliser plusieurs formalismes pour exprimer ces trajectoires. Une solution consiste a les decrire point par point. Il est aussi possible de les decrire par des lois physiques ou des formules, comme cela se fait pour la visualisation scientique. Un point est alors obtenu par application d'une formule pour un temps ou un parametre donne. Les techniques sont multiples, mais la notion de trajectoire a decrire est toujours presente. Un des avantages de cette approche est qu'elle permet des descriptions modulaires, isolant bien les phenomenes. Ainsi, si un cercle se deplace en grossissant, on decrira la trajectoire de son centre, et celle de son rayon dans l'espace des dimensions.

Denir des bases de temps

Un autre aspect important de l'animation est la maniere dont les evolutions se produisent, s'enchaînent, ou se deroulent en parallele. Il faut pour cela decrire des phenomenes temporels, qui sont souvent repetitifs. Cela revient donc d'abord a denir des bases de temps. Dans le cas le plus simple, celui du dessin anime par exemple, il suft d'une base de temps, qui peut être fournie par une horloge emettant des impulsions a une certaine frequence. Chaque impulsion est alors utilisee pour passer a l'image suivante. De nombreux systemes utilisent cette technique simple reposant sur une base de temps unique. Cela revient alors a determiner pour chaque impulsion quels objets graphiques doivent evoluer.

Cependant, cette technique utilisant une base de temps unique paraît insufsante. En effet, on rencontre souvent des phenomenes ayant des frequences differentes. Imaginons que l'on veuille realiser en parallele le clignotement d'un curseur toutes les 500 millisecondes, et le deplacement d'un objet, avec la frequence la plus elevee possible. Selon les capacites du materiel, on pourra obtenir une periode de 150 ou 200 millisecondes, par exemple. Pour qu'une base de temps puisse servir a ces deux animations en parallele, il faut des calculs de plus grand diviseur commun, ce qui donne 100, voire 50 millisecondes, ce qui ne se justiait pas a priori. Refuser d'avoir une frequence de base trop elevee conduit donc a des contraintes fortes sur les frequences des mouvements, ce qui complique la fabrication d'animations. Par ailleurs, les echelles de temps peuvent être tres differentes, si par exemple on ajoute aux deux animations precitees un carillon qui emet un son toutes les heures. On peut alors arriver a des lourdeurs inutiles dans la gestion de la base de temps.

De plus, il existe dans une animation des phenomenes plus complexes a decrire que la simple succession d'operations. Si l'on veut deplacer deux objets en parallele et a la même vitesse, il suft qu'ils bougent a chaque impulsion. Mais si l'un doit faire deux pas tandis que l'autre en fait trois, il faut alors organiser une synchronisation entre les deux mouvements. Sur six impulsions, un objet bougera aux instants 1, 3 et 5, et l'autre

aux instants 1 et 4. De telles synchronisations sont complexes a realiser si le systeme d'animation n'offre pas de services speciques.

En conclusion, un systeme d'animation doit permettre d'exprimer des phenomenes temporels, qui peuvent avoir des echelles tres differentes. Des phenomenes peuvent aussi être lies les uns aux autres, ce qui pose des problemes de synchronisation. Nous verrons que les systemes existants ne fournissant pas de tels services, et la section 4.5 sera consacree a la description des phenomenes temporels dans d'autres domaines, et en particulier la composition musicale.

De l'animation au comportement dynamique de l'interface

Lorsqu'on parle d'animation, on pense au temps, et le paragraphe precedent resume les besoins dans ce domaine. Cependant, certaines applications que nous avons rencontrees ne font pas reposer leur animation sur le temps. Ou plus exactement, le temps intervient, puisqu'il y a animation, mais il n'est pas explicite dans l'interface. Pour faire grossir un rectangle apres un double clic, l'interface a besoin d'une base de temps, qui est en general reguliere ; elle emettra des impulsions tous les 1/10emes de seconde, par exemple. Mais pour visualiser les variations de donnees du noyau fonctionnel, voire des donnees acquises par un peripherique, il n'y a pas besoin de base de temps dans l'interface : l'animation est dirigee par des sources exterieures. Par ailleurs, c'est ici le moment de se souvenir que certaines actions de l'utilisateur sont retransmises par un mecanisme d'echantillonnage. Le deplacement de la souris, par exemple, est assimilable a une acquisition de donnees echantillonnees. Donc ces actions de l'utilisateur fournissent elles aussi la base de temps necessaire a l'animation. Si l'on regarde le deplacement d'une icône avec la souris, il s'agit bien du mouvement d'un objet graphique au cours du temps, les instants des deplacements etant determines par l'echantillonnage du peripherique ou par le systeme de fenêtrage qui envoie les evenements.

L'animation par le temps est donc un cas particulier d'animation dans une interface, qui vient s'integrer avec d'autres types de *comportement dynamique.* On peut considerer qu'il existe trois types de comportement dynamique :

- dirige par le temps,
- dirige par l'utilisateur, et
- dirige par le noyau fonctionnel

Les origines du mouvement sont donc diverses. Mais le mouvement, lui, est le même quelle que soit son origine. Prenons un rectangle dont la couleur alterne entre rouge et vert. Le resultat visuel et les ordres graphiques pour l'obtenir sont les mêmes, que le clignotement soit le resultat d'impulsions d'horloge, des variations d'une valeur booleenne, ou des clics successifs de l'utilisateur. De même, la trajectoire de l'index dans une barre de delement est identique, qu'il soit deplace automatiquement pendant

que l'utilisateur appuie sur un bouton, ou deplace directement avec la souris, ou encore lie a la valeur d'un entier. Ces similarites ont deux consequences. D'une

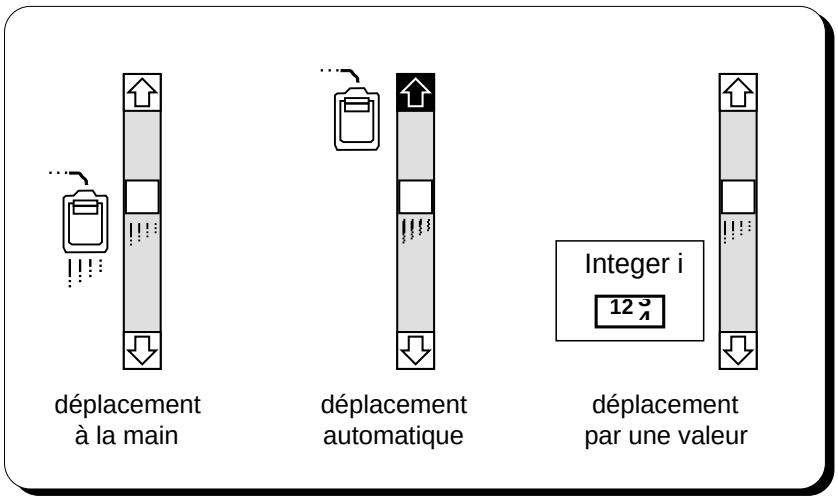

Figure 4.7 : Trois manieres de deplacer l'index d'une barre de delement.

part, il semble utile de manipuler les trois types de comportement dynamique d'une maniere homogene. Et d'autre part, il faut separer autant que possible la description de l'origine du mouvement de celle du mouvement lui-même. De cette maniere, il sera possible d'utiliser des objets animes tels qu'une barre de delement avec des sources de mouvement differentes.

En conclusion, la recherche d'une denition pour l'animation nous a mene a l'identication de la notion de comportement dynamique de l'interface, qui peut être dirige par le temps, l'utilisateur ou le noyau fonctionnel de l'application. L'animation dans son sens habituel est donc un cas particulier de comportement dynamique. Par ailleurs, il semble utile de decrire ces trois types de comportement selon les mêmes formalismes.

Vers l'autonomie des objets

Nous avons vu qu'il etait commode de decrire l'animation comme l'evolution d'objets graphiques au cours du temps. Cette evolution peut être separee en une forme d'evolution et une source de mouvement. Nous avons aussi vu que les sources du mouvement peuvent prendre plusieurs formes. Avec ce type de description, l'animation se resume donc a fournir une trajectoire et une source de mouvement a un objet, puis a le laisser evoluer.

Cependant, si ce mecanisme est sufsant pour des animations simples comme un clignotement, il devient vite insufsant dans des cas plus complexes. En effet, les

mouvements ou evolutions se font rarement sans contraintes ou cas particuliers. Si un objet se deplace sur l'ecran, on ne souhaitera pas toujours le laisser sortir de sa fenêtre et disparaître. Dans un jeu video, les objets peuvent se heurter, et changer de trajectoire. Quand l'utilisateur deplace une icône, il peut l'amener au-dessus d'autres icônes, ce qui declenche des operations. Un certain nombre d'evenements risquent ainsi de ponctuer l'animation en cours. Quelques-uns de ces evenements peuvent eventuellement être pris en compte avant le lancement de l'animation. Ainsi, avant de faire bouger un objet, on peut imaginer de determiner le moment ou il va sortir du champ de visibilite, et lui donner une trajectoire qui s'arrête a ce moment-la. Mais cette technique peut être delicate a appliquer, ne serait-ce que par la complexite des calculs pour des trajectoires non lineaires. Par ailleurs, dans le cadre d'une application interactive ou l'utilisateur peut intervenir sur la trajectoire, faire des calculs a l'avance risque d'être souvent inutile. Enn, si c'est l'utilisateur qui deplace lui-même un objet, il est impossible de prevoir ce qui va se produire.

Plutôt que de determiner a l'avance ce qui va arriver aux objets graphiques que l'on lance sur des trajectoires, l'autre solution consiste a leur laisser une apparente autonomie, et demander a être prevenu lorsque quelque chose se produit. Pour realiser cela, on peut imaginer un systeme ou les objets animes examinent leur environnement a chaque pas, et envoient des evenements dans certaines circonstances. C'est alors au programmeur de prevoir ce qu'il faut faire. Il y a deux avantages a cette approche. D'une part, cela autorise le traitement de situations difciles a prevoir. D'autre part, cela permet une programmation de type declaratif, ou l'on specie un comportement plutôt que de le commander. Une autre consequence interessante est l'importance accrue qui est accordee aux objets graphiques : porteurs de leur apparence, ils sont maintenant responsables de leur comportement. Des informations telles que leur position ne sont pas connues en permanence du reste de l'application. De plus, ils ont souvent besoin d'informations semantiques pour mettre en uvre ce comportement. L'introduction de l'animation a donc des consequences sur l'architecture des applications interactives.

Animation et architecture des interfaces

Nous avons vu au chapitre 2 qu'il existait un large consensus autour de la separation entre interface et noyau fonctionnel. Cette separation amene souvent a considerer que la semantique d'une application est contenue dans le noyau fonctionnel, et que l'interface ne contient que les entites lexicales ou syntaxiques necessaires a la presentation et l'interaction. Nous avons aussi constate que des difcultes apparaissent souvent lors de la mise en uvre de tels modeles pour des interfaces a manipulation directe. En effet, ces interfaces ont souvent besoin de se referer a des informations semantiques pendant l'interaction, et on est toujours a la recherche de modeles pour structurer ces echanges d'information. Des notions telles que la delegation semantique ont ete introduites pour tenter de formaliser ce phenomene.

L'apparition d'interfaces animees accroît notablement ce besoin d'informations semantiques dans l'interface. En effet, nous avons vu qu'il etait souhaitable que les objets de presentation soient responsables de leur evolution, pour simplier le fonctionnement de l'interface. Cela signie qu'ils doivent maîtriser leur trajectoire, mais aussi leurs reactions aux actions de l'utilisateur, voire leurs interactions avec d'autres objets de presentation. Or ces reactions font en general appel a des informations semantiques. Une icône ne reagira a la proximite d'une autre icône que si les donnees qu'elles representent sont susceptibles d'interagir. Dans un jeu, un projectile n'explosera qu'en heurtant un vaisseau ennemi. L'animation demande donc de disposer d'encore plus de semantique dans l'interface, ce qui remet en cause une certaine conception de l'architecture des applications interactives.

4.4 Les systemes existants

Les sections precedentes ont ete consacrees a l'identication de l'animation dans les interfaces : ses applications, sa nature, ses besoins, et ses consequences. Nous allons maintenant examiner deux systemes qui ont chacun a leur maniere introduit l'animation dans des applications interactives. Ces deux systemes utilisent la notion d'objet graphique pour construire des animations, mais leurs methodes de description du mouvement sont totalement differentes.

4.4.1 Animus

Animus est un systeme destine a la visualisation animee de phenomenes physiques et l'animation d'algorithmes [Duisberg 86]. Il a ete realise par Robert Duisberg a l'Universite de Washington. Il s'agit d'une extension de ThingLab, un systeme de programmation par contraintes. ThingLab a ete cree pour construire des simulations physiques interactives. ThingLab a ensuite ete applique a la construction d'interfaces [Borning & Duisberg 86]. Il a aujourd'hui un descendant, ThingLab II, explicitement destine a la construction d'interfaces [Maloney et al. 89].

ThingLab a ete developpe dans l'environnement Smalltalk. Il offre la possibilite de declarer des contraintes entre objets, c'est-a-dire des relations qui doivent être maintenues au cours de l'execution d'un programme. Les contraintes servent a lier des donnees entre elles. Elles peuvent aussi servir a maintenir la coherence entre des donnees et leur representation graphique, ou a disposer les objets graphiques selon certaines regles. Une contrainte contient generalement une description de la relation a maintenir, ainsi qu'un ensemble de procedures pour y parvenir. Ces procedures sont utilisees par un moteur de resolution, qui retablit les relations lorsque l'une d'entre elles est perturbee.

L'application principale de ThingLab est la simulation de systemes physiques, comme des systemes de poutres reliees entre elles. Les contraintes servent a exprimer les contraintes physiques entre ces poutres. L'utilisateur peut tenter de deplacer une partie du systeme. ThingLab retablit alors l'equilibre, et l'utilisateur voit comment ses actions se repercutent sur le reste du systeme physique. Il faut noter que les systemes simules avec ThingLab ne sont pas animes. Ce sont uniquement les actions de l'utilisateur qui provoquent les modications d'etat et d'apparence, et le temps n'intervient pas. ThingLab est donc limite a la simulation de systemes en equilibre. Les systemes dynamiques ne peuvent être simules qu'avec les mecanismes d'animation d'Animus.

Contraintes temporelles

Animus etend le modele de contraintes de ThingLab par des *contraintes temporelles.* Ces contraintes ont la forme d'equations differentielles ou de fonctions faisant intervenir le temps. Elles lient le temps et des grandeurs qui varient avec lui. Par exemple, il est possible de simuler un circuit electrique avec des objets contenant des contraintes pour representer des composants. Un condensateur contiendra la contrainte $i = \frac{dq}{dt}$, et une bobine la contrainte $U = L\frac{di}{dt}$. L'animation est obtenue grâce a un objet global, l'horloge. Cette horloge a connaissance de toutes les contraintes temporelles, et les met a jour a chaque impulsion. Dans le cas du circuit electrique, Animus produit pour chaque contrainte un programme d'approximation par differences nies. Il calcule les nouvelles valeurs a partir des anciennes a chaque increment de temps. On obtient ainsi une oscillation du courant dans le circuit, comme dans le circuit electrique reel. Ainsi les contraintes temporelles permettent-elles de representer des phenomenes dynamiques continus.

Contraintes gardees

Animus permet aussi de decrire des animations d'algorithmes, grâce a des *contraintes gardees.* Ce sont des contraintes qui ne sont activees que lorsqu'elles recoivent un message qui repond aux conditions exprimees dans la *garde.* Les contraintes temporelles decrites precedemment sont un cas particulier de contraintes gardees, declenchees par les impulsions de l'horloge. Dans le cas de l'animation d'algorithmes, des messages sont envoyes par l'algorithme en cours d'execution. Ces messages declenchent des contraintes gardees, lesquelles provoquent des reajustements dans les representations graphiques. Les contraintes gardees sont aussi capables d'emettre a leur tour une serie de messages vers d'autres contraintes. Par exemple, il est possible de realiser une contrainte gardee \Trajectoire", qui emet une serie de positions lorsqu'elle est activee. D'apres Duisberg, les contraintes gardees sont ainsi equivalentes aux commandes gardees de CSP [Hoare 78].

Discussion

Animus permet de decrire des phenomenes dynamiques continus ou discrets. En cela, il semble sufsamment general pour realiser des interfaces animees. Cependant, si des mecanismes generaux sont fournis, Animus n'identie pas les briques de base necessaires pour fabriquer des animations simples. Par ailleurs, ses inuences d'origine se font fortement sentir : tout est ramene a des systemes physiques decrits par des contraintes.

Le modele des contraintes paraît adapte lorsqu'il s'agit de rendre compte de lois physiques liant le temps a d'autres grandeurs. Mais il s'agit la de cas particuliers, que l'on peut difcilement generaliser a la construction des interfaces. Il est naturel de representer un phenomene physique par une equation. Mais pour une animation simple comme un objet se deplacant sur une droite, il est plus intuitif pour un programmeur de donner simplement le point de depart et la vitesse.

Les contraintes gardees, elles, rendent mieux compte des phenomenes que l'on desire decrire pour des interfaces animees. Des evenements declenchent des suites d'autres evenements, qui peuvent produire par exemple le deplacement d'un objet. En revanche, le nom de *contraintes* qui leur est donne paraît abusif. Une contrainte est censee maintenir une relation entre des objets. Ici, on etablit une relation de cause a effet entre des evenements, ce qui est une interpretation assez eloignee de l'idee d'origine. Qu'on utilise des contraintes comme mecanisme d'implementation peut se justier, mais moins comme paradigme de construction.

La notion de contrainte est visiblement peu adaptee a la description d'animation dans les interfaces. Cependant, les objets que Duisberg nomme contraintes gardees sont tres interessants. Ils permettent d'isoler un certain type de comportement a l'interieur d'un objet, comme les objets \Trajectoire" ou \Clignotement". Il est possible de les associer a volonte a tel ou tel objet graphique. On obtient ainsi une tres grande modularite dans la creation d'animations.

4.4.2 Tango

Tango est un systeme d'animation d'algorithmes developpe par John Stasko a l'Universite Brown. Contrairement a d'autres systemes d'animation d'algorithmes, sa contribution reside surtout dans la maniere de denir des animations graphiques. Tango est en effet une des premieres tentatives visant a formaliser la description des animations. Stasko y introduit un modele simple, qu'il nomme le *modele chemin-transition*[3]. Ce modele isole certains composants de base d'une animation, ce qui permet de manipuler ces composants avec des outils specialises. En particulier, Stasko

[3] path-transition paradigm

a developpe Dance, un outil a manipulation directe pour la creation d'animations simples.

Les objets de base de Tango

Quatre classes d'objets sont identiees dans Tango. Ce sont les *images*, les *emplacements*, les *chemins* et les *transitions*. Les images sont des objets graphiques tels que rectangles, lignes ou cercles. Les emplacements sont des positions, decrites par deux coordonnees decimales. Les chemins sont des suites nies de positions. C'est en manipulant des chemins que l'on decrit l'evolution des objets graphiques au cours du temps. Finalement, les transitions sont les briques de base de l'animation. Une

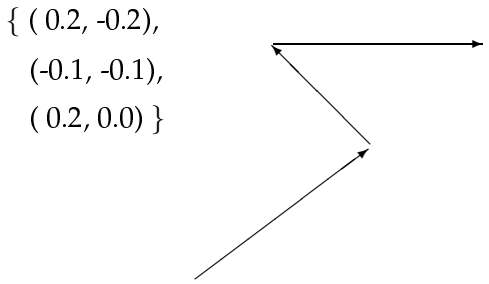

Figure 4.8 : Un chemin de longueur 3.

transition simple associe une image et un chemin, et decrit la maniere d'utiliser le chemin. Par exemple une transition de visibilite ignore les positions successives du chemin, pour simplement rendre l'image visible ou invisible a chaque etape. De maniere similaire, une transition de remplissage utilise la coordonnee x pour modier le coefcient de remplissage de l'image. Et bien entendu, une transition de deplacement positionnera l'image successivement a chacun des emplacements du chemin. Une fois qu'une transition est denie, on peut l'executer, et ainsi produire l'animation desiree.

Manipuler des chemins

Un des points essentiels du modele chemin-transition est la manipulation de chemins. Les chemins sont des ressources qui peuvent être creees, stockees et reutilisees. Mais surtout, Tango fournit un ensemble d'operations algebriques sur ces chemins. Cela permet de fabriquer des deplacements complexes a partir de chemins de base.

Tout d'abord, des operations geometriques sont denies. Ces operations sont la rotation et l'homothetie. La translation est inutile, dans la mesure ou les chemins donnent des positions relatives.

Ensuite, il est possible de grouper des chemins. On peut les concatener, c'est-a-dire les mettre bout-a-bout. On peut aussi les composer, c'est-a-dire additionner leurs deplacements successifs. On peut enn iterer un chemin, ce qui consiste a le concatener avec lui-même un certain nombre de fois.

Enn, deux grandeurs sont denies pour chaque chemin. Ce sont la longueur, qui represente le nombre de positions, et le deplacement total. Ces grandeurs rendent compte des caracteristiques les plus utiles a connaître pour un chemin : ou il va, et en combien de temps. Elles sont determinantes en particulier pour la composition et la synchronisation de deplacements. Il existe des operations pour donner a un chemin une certaine longueur ou un certain deplacement total. Cela permet d'adapter un chemin de base pour une tâche donnee, tout en conservant sa forme.

Creer des scenes

De la même maniere qu'il est possible de manipuler les chemins pour obtenir des deplacements complexes, les transitions peuvent être associees pour fabriquer des transitions complexes. On obtient ainsi de veritables scenes d'animation. Les operations les plus importantes sont la concatenation et la composition. Elles permettent d'executer des transitions successivement ou en parallele. Par exemple, si l'on dispose d'un cercle, d'un rectangle et d'un chemin, on peut fabriquer deux transitions avec le chemin et chacune des deux images; on peut ensuite composer les deux transitions. Lorsque l'on executera la transition complexe obtenue, le cercle et le rectangle se deplaceront ensemble sur le même chemin.

Dance

Le modele de Tango permet la creation interactive d'animations. Dance est un outil a manipulation directe pour la construction de scenes. Dance se presente a peu pres comme un outil usuel de dessin : une surface de travail surmontee d'une barre de menus deroulants. La surface de travail contient les images, les chemins representes par des suites de petits cercles, et les transitions representees par des icônes dans un coin de la surface. Il existe un menu pour chacun des quatre types de base. Ce menu contient des commandes pour creer des objets et leur appliquer les operations decrites plus haut, apres les avoir selectionnes a travers leur representation.

Une fois une scene construite, Dance produit une description des transitions sous la forme de fonctions ecrites en C. Il faut ensuite compiler le chier obtenu et appeler ces fonctions dans un programme.

Discussion

Le systeme Tango fait gure de pionnier dans le domaine de la description d'animations. Par ailleurs, malgre son caractere relativement general, l'inuence de l'animation d'algorithmes y est tres forte.

Parmi les contributions notables de Tango, on peut noter la maniere dont il permet de manipuler des chemins. On peut ainsi fabriquer simplement des trajectoires complexes. Un autre point original est l'introduction de scenes parametriques, utilisables dans des congurations variees.

D'autres problemes sont moins bien traites dans Tango. Citons d'abord sa description embryonnaire des phenomenes temporels, tels que la synchronisation. La seule synchronisation facile a realiser est le parallelisme, ou deux transitions sont effectuees exactement ensemble. Si l'on veut par exemple les alterner (un pas pour l'une, un pas pour l'autre), cela devient plus delicat. Par ailleurs, Tango offre un faible niveau d'abstraction pour la description des trajectoires. Les chemins sont utiles, mais on aimerait pouvoir manipuler des trajectoires circulaires ou rectilignes autrement que point par point. Ensuite, on notera l'absence de traitement des operations ponctuelles. Pour realiser un simple changement de couleur, il faut denir un chemin ne comportant qu'un pas, et lui rajouter des pas non signicatifs pour le synchroniser avec les autres animations. La simplicite apparente du modele amene donc a des constructions complexes pour des phenomenes relativement simples. Par ailleurs, Tango n'est pas extensible. Son modele est explicitement destine a decrire des deplacements dans le plan. Les deformations ou les transformations d'aspect sont ajoutes de maniere peu claire. Enn, Tango souffre d'une mauvaise integration dans les applications interactives. Quand Tango joue une scene, les entrees de l'utilisateur sont gelees. C'est la le principal reproche qu'on peut lui faire, car cela limite considerablement son utilisation dans des interfaces.

4.5 La gestion du temps

Nous avons mentionne a la section 4.3 l'importance d'un mecanisme de gestion des phenomenes temporels pour un systeme d'animation utilisable dans des interfaces. Or les systemes que nous avons decrits a la section precedente ne possedent pas de tels mecanismes. Nous allons donc nous attarder sur cette notion, et examiner les mecanismes de gestion du temps que l'on rencontre dans d'autres domaines. Dans un premier temps, nous examinerons un langage destine a la description de phenomenes en temps reel. Ensuite, nous nous interesserons aux systemes de synthese musicale.

4.5.1 *Les langages temps-reel*

Esterel

Esterel est un langage destine a la programmation des systemes reactifs, et realise a l'Ecole des Mines de Paris [Berry et al. 87]. Sous le nom de *systemes reactifs*, on regroupe des systemes qui reagissent de maniere deterministe a des entrees provenant de leur environnement, et produisent des sorties vers cet environnement. Tous les systemes de commande en temps reel sont des systemes reactifs, de même que les jeux video, ou les interfaces homme-machine.

Esterel repose sur la constatation que les langages de programmation parallele comme OCCAM ou ADA sont mal adaptes a la description de tels systemes. En particulier, ils imposent une notion d'asynchronisme entre les causes et les effets. Avec ces langages, il est impossible que deux tâches soient en permanence synchronisees. Il faut avoir recours aux communications ou aux rendez-vous pour les synchroniser de maniere ponctuelle. Cet asynchronisme ne permet pas une description rigoureuse des phenomenes temporels. Esterel, quant a lui, repose sur une hypothese de synchronisme : les sorties sont fournies de maniere synchrone aux entrees. Cette hypothese est evidemment une approximation, mais les auteurs d'Esterel estiment que cela n'est pas un obstacle majeur. En effet, toutes les sciences physiques utilisent la même hypothese de synchronisme entre causes et effets, en sachant parfaitement qu'elle est fausse. L'important est qu'a l'echelle de l'observateur, la reaction paraisse instantanee. Ainsi, cette hypothese peut s'appliquer aux systemes temps-reel \lents", pour lesquels le temps de calcul est tres faible devant le temps de reponse attendu.

Un programme Esterel est compose de modules communiquant entre eux et avec leur environnement. La communication repose sur des signaux et des capteurs. Les capteurs fournissent des valeurs qu'un module peut lire quand il le desire. Ils sont inspires des capteurs physiques, tels que thermometres ou manometres. Les signaux representent le mode de communication le plus important. En effet, Esterel est surtout destine a representer des systemes fonctionnant par evenements. C'est l'environnement qui declenche l'activite d'un module, et non le module qui sonde regulierement son environnement. Ce sont les signaux qui permettent ce fonctionnement. Un module qui ne recoit pas de signaux est inactif. En revanche, la reception d'un signal declenche de maniere instantanee le programme contenu dans un module. Toujours de maniere instantanee, celui-ci va alors lire des capteurs, emettre d'autres signaux, et eventuellement en recevoir de nouveaux.

Pour Esterel, le temps est un signal parmi d'autres, qui n'a pas de rôle privilegie. Il peut par exemple provenir directement d'un dispositif physique. Il peut même ne pas apparaître si le systeme n'en a pas besoin. De cette maniere, il est possible que d'autres signaux denissent leur temps propre, comme dans l'instruction \s'arrêter au bout de 2 metres".

Le traitement d'un signal, que ce soit sa diffusion, les calculs associes, ou l'emission d'autres signaux en consequence, est instantane. Prenons un exemple simple. Supposons qu'un dispositif exterieur envoie des signaux *Milliseconde*. Il est possible de realiser un module qui capte ces signaux, et envoie un signal *Seconde* pour un *Milliseconde* sur mille. Ce signal *Seconde* sera envoye de maniere synchrone avec le signal *Milliseconde* correspondant. Pour illustrer ce synchronisme d'une autre maniere, on peut considerer un des paradoxes mentionnes par les auteurs d'Esterel. Supposons un module qui soit sensible au signal S. Le comportement de ce module est le suivant : si S est present, il ne fait rien. Sinon, il emet S. On se trouve alors face a une oscillation inniment rapide. En pratique, le compilateur detecte et interdit ce genre de boucles.

Les signaux que nous avons mentionnes jusqu'a present ne transportent pas de valeur. Ce sont des signaux purs. Mais d'autres types de signaux peuvent transporter des informations supplementaires, accessibles aux modules qui les recoivent. Rien n'empêche deux modules d'emettre le même signal en même temps. Par exemple, on peut imaginer deux instances du module emettant des *Seconde* a partir de *Milliseconde*. Dans le cas d'un signal pur, la conjonction de deux signaux identiques est sans consequence. En revanche, pour les signaux avec valeur, Esterel permet de denir la valeur du signal resultant, a l'aide d'une loi commutative associee a chaque type de signal. Cette loi est appliquee aux valeurs des deux signaux identiques pour obtenir la valeur qui sera nalement recue par les modules destinataires. Ainsi, dans un systeme de vote, on pourra denir des signaux *Oui* et *Non* portant une valeur entiere, et leur associer l'addition comme loi de composition. Le module qui recueillera les votes recevra alors un signal *Oui* contenant le nombre de modules ayant emis un *Oui*, et de même pour les *Non*.

Le langage Esterel n'est pas un langage de programmation complet. L'automate fabrique lors de la compilation d'un programme Esterel est destine a être integre dans un programme. Cette approche est similaire a celle de YACC, qui permet de decrire des analyseurs syntaxiques, destines a être utilises dans des programmes ecrits en C.

4.5.2 *La synthese musicale*

Parmi les domaines ou apparaît la modelisation de phenomenes temporels, la musique par ordinateur a ete abondamment etudiee. La musique avait toujours fait bon menage avec les mathematiques. La même chose s'est produite avec l'informatique des ses debuts. Comme dans d'autres domaines, l'informatique permet de simplier des tâches fastidieuses comme la saisie ou la typographie de partitions. Mais l'important est ailleurs. Tout d'abord, le traitement numerique du son a multiplie les degres de liberte offerts au compositeur. Ensuite, l'informatique a permis l'avenement de nouveaux formalismes pour decrire la musique. On est progressivement passe de theories analytiques a des theories constructives, qui permettent de produire des

compositions a partir des regles. D'un autre côte, les informaticiens ont trouve dans la musique un terrain d'etudes tres riche. On y rencontre entre autres des problemes de representation des connaissances, de modelisation de phenomenes, et de performance. Cette convergence d'interêts a fait de la musique par ordinateur un domaine de recherche privilegie.

Le sous-domaine qui nous interesse ici est celui de la composition et de la synthese musicale. Il s'agit de decrire un phenomene temporel complexe, plus complexe sans doute que l'animation : le son. La musique est riche en repetitions et variations de motifs. Elle met en parallele plusieurs instruments, ou les fait se repondre en sequence. Il est donc utile d'etudier les solutions apportees dans ce domaine avant d'envisager des choix de modelisation pour l'animation.

Nous ne decrirons pas ici le contexte de la musique, ni les problemes speciques qu'elle pose. Le lecteur pourra se reporter a [Loy & Abbott 85] qui est une excellente synthese sur la musique par ordinateur. Nous presenterons d'abord deux systemes parmi les plus connus. Nous tenterons ensuite de degager les grandes lignes des modeles utilises, et de voir en quoi ceux-ci sont applicables a l'animation.

Csound

CSound [Vercoe 86] est un descendant direct des premiers langages de synthese musicale. Ces langages, nommes de Music I a Music V, ont ete developpes aux Bell Labs au cours des annees 1960. Ils sont collectivement denommes Music N. De nombreuses implementations et variantes ont vu le jour depuis, et CSound est l'une des plus recentes. Il a ete realise au Media Lab du MIT par Barry Vercoe, precedemment auteur de Music 360 et Music 11.

Le but de CSound est de fabriquer des echantillons de son numerique directement utilisables dans des convertisseurs de signal numerique en signal analogique. Il se distingue en cela de systemes plus recents, qui ont plutôt tendance a produire des commandes pour des processeurs de son numerique (DSP), ou des ordres pour des synthetiseurs selon le protocole MIDI. En fait, certains de ces systemes recents produisent même des chiers de commandes pour CSound. Les calculs a effectuer et les donnees a produire pour obtenir du son numerique sont tres volumineux. Il a donc ete longtemps irrealiste d'imaginer une synthese musicale en temps reel. En consequence, CSound est avant tout un compilateur, qui prend des ordres en entree, et fabrique des donnees en sortie.

CSound permet de decrire un morceau musical sous la forme de deux chiers. Le premier chier decrit un *orchestre*. Il est compose d'*instruments*, et decrit comment le son est synthetise. Le second chier est la *partition*. Elle contient des *notes*, qui decrivent quand les instruments doivent jouer, et avec quels parametres.

Les instruments de CSound sont fabriques a partir de modules de base. Ces modules realisent des fonctions simples telles que l'oscillation, le gain d'amplitude, l'addition de signaux, ou le ltrage. Ils peuvent être organises selon un modele de ot de donnees : ce que produit un module est fourni a une des entrees du suivant. Les donnees qui transitent dans ces modules representent le signal, comme dans un circuit electrique.

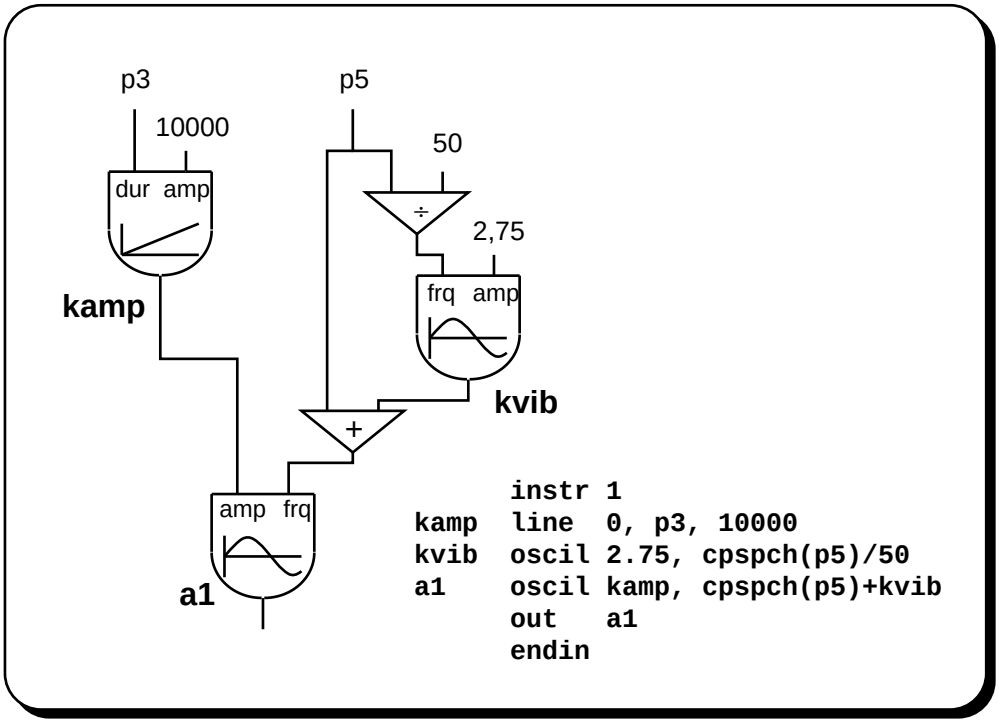

Figure 4.9 : Un instrument de CSound et le programme correspondant.

Chaque note de CSound contient le numero de l'instrument qui doit la jouer, le moment de son debut, et sa duree. Elle peut contenir d'autres parametres, qui sont interpretes par l'instrument. Parmi ces parametres, on trouve en general la hauteur de la note. L'unite de temps utilisee pour la description des notes est dependante du *tempo*. Ce dernier peut être change a tout moment dans la partition, ce qui est une necessite en musique.

L'anciennete des premiers Music N fait helas que la syntaxe utilisee par CSound est plus proche de l'assembleur que des langages modernes. On peut donc considerer le fait qu'il soit encore utilise comme une preuve de l'interêt des concepts utilises. Plus serieusement, on notera que CSound offre une description tres ne du son, mais a un niveau tres bas. Il n'impose ni ne propose aucune structure au compositeur. Cela donne une grande liberte pour creer sa propre structure musicale. Cependant, les systemes plus recents ont une approche beaucoup plus structuree de la musique. Formes, que nous allons examiner maintenant, en est un bon exemple.

Formes

Formes est un environnement de programmation destine a la composition et la synthese musicale [Cointe & Rodet 84]. Il a ete developpe a l'IRCAM par Pierre Cointe et Xavier Rodet, et utilise par des compositeurs professionnels. Bien que la musique en soit l'application principale, ses auteurs soulignent que Formes peut-^etre utilise pour d'autres processus dynamiques tels que l'animation graphique ou la synthese de la parole.

A l'origine de Formes se trouvait la constatation que beaucoup de langages musicaux ne permettaient de manipuler que des entites de bas niveau, comme des oscillateurs. Des notions plus importantes comme la structure musicale ou l'expression du temps etaient souvent negligees. Formes met donc l'accent sur ces notions, au point qu'il lui a ete reproche de decrire la structure de la musique, mais pas sufsamment le son lui-m^eme. Les principales caracteristiques de Formes sont la notion de hierarchie de processus et la communication par \pipe-line".

L'objet principal de Formes est le *processus*. Un processus est decrit par une extension temporelle (le debut et la n), un ensemble de *ls* (des processus), et un *moniteur*, qui decrit comment sont organises les ls. Un processus contient aussi un ensemble de variables locales, et un ensemble de procedures, qui decrivent son execution. C'est a travers l'execution des processus que les ordres sonores sont produits. L'ensemble des processus actifs a un moment donne denit un arbre d'execution, sous-ensemble de l'arbre de tous les processus. A chaque impulsion de l'horloge, l'arbre d'execution est examine, et les procedures des processus actifs sont executees. L'arbre est ensuite mis a jour en fonction du debut ou de la n des processus. La maniere dont les ls d'un processus sont actives est determinee par le moniteur de ce processus. C'est lui qui determine si les ls sont actives en parallele ou en sequence. Le temps a l'interieur de chaque ls est relatif au debut de son activation par le moniteur.

La communication entre les processus s'effectue par des \pipe-lines". Sous ce nom, les auteurs de Formes designent des variables globales, qui sont utilisees par les processus pour passer a leurs ls les signaux calcules. Les ls vont lire ces variables, les transformer, et les passer aux petits-ls. En n de parcours, les signaux sont transmis aux entrees du synthetiseur ou stockes. Ces pipe-lines peuvent aussi servir a la communication entre deux ls d'un m^eme processus. Cela permet par exemple de resynchroniser deux voix gr^ace a un systeme de semaphores. Lorsqu'un silence est rencontre par une voix, elle attend l'autre voix pour attaquer la note suivante en parfaite synchronisation. Ce genre de construction permet de denir des processus avec des tempos differents, ou m^eme uctuants, sans calcul prealable.

Discussion

Les deux systemes que nous avons presentes sont assez differents. Ils le sont en particulier dans leur approche de ce qu'est la musique. Pour l'un il s'agit de decrire la forme des sons a l'echelle la plus ne possible. Pour l'autre, l'important est de decrire la structure de la musique, aussi bien a l'echelle de la note qu'a celle du morceau musical. Cependant, on retrouve un certain nombre de points communs, qui apparaissent aussi dans d'autres systemes de composition musicale.

Tout d'abord, on retrouve dans la plupart de ces systemes un modele plus ou moins explicite de ot de donnees. Le signal sonore nal est obtenu en faisant circuler des signaux dans des modules connectes entre eux. Ce sont les modules qui transforment et afnent progressivement le signal. Il faut sans doute voir la, a l'origine, l'inuence des premiers systemes analogiques de synthese musicale, eux-mêmes inspires des circuits electroniques comme ceux des radios. Cependant, la constance avec laquelle on retrouve ce modele laisse penser qu'il est bien adapte a la description d'un processus dynamique. Ainsi, on le trouve de maniere explicite dans CSound, Play [Chadabe & Neyers 78], Flavors Band [Fry 91] ou le Music Kit de NeXT [Jaffe & Boynton 91]. Les pipe-lines de Formes mettent en uvre un modele similaire. Dannenberg, quant a lui, afrme que \Pour les programmes qui echantillonnent des entrees, realisent des operations et produisent des sorties echantillonnees, le modele de calcul approprie est un graphe de ot de donnees. . . "[4] [Dannenberg 84].

Un autre point interessant concerne la maniere dont la production du signal est provoquee. On peut distinguer les systemes par sondage (ou scrutation[5]) et les systemes a base d'evenements. Les premiers prennent une base de temps (la frequence d'echantillonnage du son par exemple), et calculent repetitivement l'etat du systeme. Les seconds ne le font que lorsqu'un evenement se produit. Ces evenements sont par exemple les notes dans CSound, ou le debut et la n des notes dans Flavors Band. MODE propose même une hierarchie d'evenements allant de la note a la piece musicale entiere. Le sondage est adapte lorsque le signal est tres repetitif. En revanche, lorsque des evenements signicatifs se produisent a une frequence beaucoup plus faible que la frequence d'echantillonnage, le sondage apparaît comme un gaspillage. Et c'est ce qui se produit pour le debut et la n des notes. C'est pour cette raison que CSound utilise une approche mixte. D'une part, les oscillateurs fonctionnent par sondage. Et d'autre part, les notes sont les evenements qui declenchent les calculs. D'autres systemes comme Arctic font le même choix de fonctionner par sondage pour les parties de bas niveau, et par evenements pour les parties de plus haut niveau.

Les systemes de composition musicale fournissent des exemples instructifs de

[4] For programs that sample inputs, perform operations and produce sampled output, the appropriate model of computation is a data-ow graph.

[5] polled systems

description de phenomenes temporels. Cependant, les considerations qui ont conduit a la construction de ces systemes ne sont pas toutes transposables a l'animation. En effet, la musique est un phenomene tres complexe compose de structures imbriquees, alors que l'animation a une structure temporelle beaucoup moins ne. Il faut sans doute voir la une consequence des sensibilites differentes de l'oue et de la vue : on considere qu'une animation est parfaite a 60 images par seconde, alors que la musique fait intervenir des frequences de l'ordre de 20000 cycles par seconde. De plus, toute l'information portee par la musique est de nature temporelle, alors que l'animation est surtout une suite d'etats porteurs d'informations, auxquels on ajoute un caractere dynamique. Un systeme d'animation peut donc se contenter d'exprimer des structures temporelles plus simples.

Une autre difference majeure entre la musique et l'animation telle que nous l'avons decrite a la section 4.3 est que la musique est prevue a l'avance, alors que l'animation peut être perturbee par des phenomenes exterieurs. Ainsi, la construction \commencer cette action et s'arrêter au bout de n secondes" proposee par Formula [Anderson & Kuivila 90] doit être generalisee a d'autres types d'interruptions. De plus, ces interruptions ne sont pas previsibles a l'avance, que ce soient des actions de l'utilisateur (\arrêter le clignotement quand on appuie sur une touche") ou des evenements en provenance de l'application (\s'interrompre quand cette variable devient negative"). L'animation a donc un caractere plus dynamique que la musique, ce qui necessite des amenagements particuliers. On pourra en particulier y distinguer les phenomenes continus, similaires a la musique, et les evenements imprevus, plus ponctuels.

Une fois ces remarques enoncees, il faut toutefois noter que l'on rencontre des points communs entre les systemes musicaux et des langages plus generaux comme Esterel. En particulier, tous font intervenir la notion de propagation de l'information comme moteur du comportement dynamique du programme. La consequence est que la plupart de ces systemes ont de maniere plus ou moins explicite un modele de ot de donnees. On peut donc supposer qu'un systeme d'animation pourra tirer prot d'un tel modele.

4.6 Conclusion

Ce chapitre a ete consacre a l'etude de l'animation dans les interfaces. Nous en avons examine les applications possibles et la nature, ce qui nous a amene a denir les principales caracteristiques d'un systeme de construction d'interfaces animees. En particulier, nous avons identie des fonctions relativement independantes : la description des mouvements et changements d'aspects, et la description des phenomenes temporels. De plus, nous avons note que l'animation vient s'integrer a des phenomenes deja existants dans les interfaces, permettant ainsi d'introduire la notion de comportement

dynamique de l'interface. Cette analyse de l'animation va maintenant nous servir a proposer un modele et un systeme pour construire des interfaces animees.

Chapitre 5.

Whizz

Dans ce chapitre, nous allons examiner Whizz, une boîte a outils pour la construction d'interfaces animees. Whizz a ete developpe en fonction des besoins decrits au chapitre precedent. C'est une extension d'X_{TV}, concue pour decrire le comportement dynamique d'une interface, quelle que soit l'origine de ce comportement.

Ce chapitre est organise de la maniere suivante. La section 5.1 presente la metaphore utilisee par Whizz, inspiree du monde musical. La section 5.2 decrit de maniere plus complete son modele, reposant sur les notions d'evenements et de ots de donnees. La section suivante est consacree aux differentes parties de Whizz : sources du mouvement, synchronisation, description du mouvement, animation graphique, sources et traitement des evenements. Nous examinerons ensuite la maniere dont Whizz s'integre avec X_{TV}, puis nous decrirons des extensions de Whizz pour les comportements autres que graphiques. Nous detaillerons ensuite deux exemples d'utilisation, avant d'etudier Whizz'Ed, un editeur graphique de scenes d'animation. Enn, les deux dernieres sections sont consacrees aux problemes de realisation les plus signicatifs, et aux aspects restant a etudier.

5.1 Une metaphore musicale

Nous avons vu au chapitre 3 qu'X_{TV} utilise une analogie avec un domaine du monde reel pour faciliter la comprehension et la memorisation de son modele. C'est aussi le cas pour Whizz. Cette metaphore va nous servir d'introduction au fonctionnement de Whizz.

Whizz utilise une metaphore musicale. En cela, il complete de maniere plaisante X_{TV}, qui fait intervenir le monde du spectacle. Par ailleurs, Whizz a ete inuence par les systemes de composition musicale, et la metaphore choisie rend compte de ces inuences. Cependant, il ne faut pas y voir plus qu'une image. En plus de l'animation

graphique, Whizz permet eventuellement d'ajouter une composante sonore a une interface, mais il n'est pas prevu pour representer de la musique.

Danseurs, instruments, rythmes

Whizz fait intervenir des danseurs, des instruments, des notes, des rythmes et des tempos. Les *danseurs* sont les entites qui produisent l'effet perceptible par l'utilisateur. Ce sont le plus souvent des objets graphiques, qui se deplacent ou se deforment. Les danseurs agissent en fonction des *notes* qu'ils entendent. Ces notes sont emises par des *instruments*. Ce sont des generalisations des notes musicales. Elles transportent

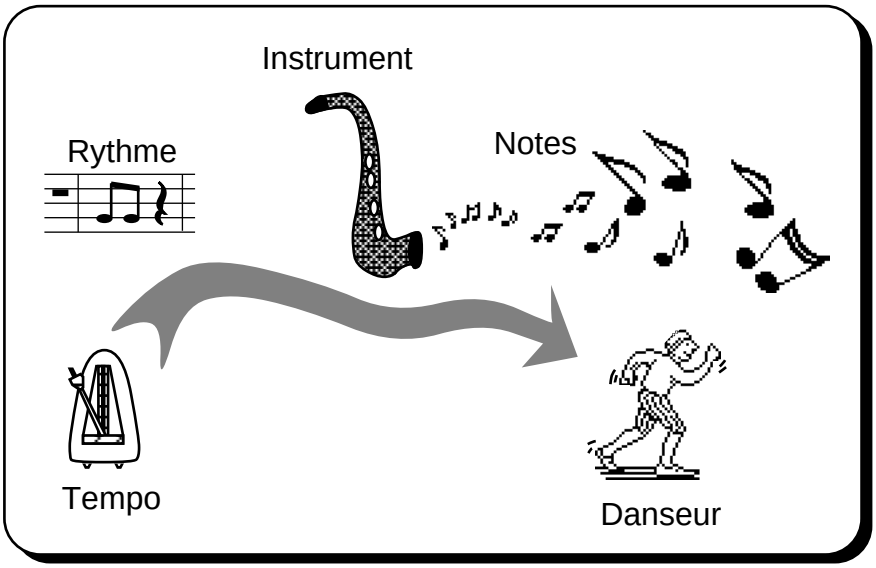

Figure 5.1 : La metaphore de Whizz.

des informations simples, telles que des nombres, des positions ou des couleurs. Les instruments jouent ces notes en fonction de leur type et de leur etat. Certains instruments emettent des positions le long d'une trajectoire. D'autres emettent des entiers lus dans un chier. Ils jouent ces notes a des instants determines par un *tempo* et un *rythme*. Le tempo est une frequence de base, qui determine une suite d'instants. Parmi ces derniers, le rythme determine les instants auxquels l'instrument doit jouer une note.

Il n'y a pas de bijection entre danseurs et instruments. Plusieurs danseurs peuvent ecouter les mêmes instruments, et un même danseur peut ecouter plusieurs instruments a la fois. Dans ce dernier cas, la metaphore musicale est un peu deformee. Generalement, les instruments d'un orchestre jouent des morceaux complementaires, et les danseurs ecoutent la musique qui en resulte. Ici, les danseurs ecoutent independamment chaque instrument, et generalement y reagissent independamment. Imaginons un instant un

danseur qui se deplace au son de la batterie, bouge son bras droit avec la guitare electrique et son bras gauche avec la trompette. Dans le cas de Whizz, cela donne par exemple un rectangle qui se deplace selon les positions donnees par un instrument, et change de couleur en fonction d'un autre instrument.

De la même maniere, les tempos et les rythmes peuvent être partages par plusieurs instruments. Deux instruments qui ont le même rythme jouent leurs notes aux mêmes instants. Des synchronisations plus complexes sont obtenues en utilisant des rythmes differents et un même tempo. Utiliser des tempos differents ne permet pas de synchronisation. En revanche, cela permet d'exprimer des phenomenes dont les echelles de temps sont differentes.

Evenements imprevus

Les mouvements de base des danseurs sont determines par les notes qu'ils entendent. Cependant, il peut se produire des evenements imprevus qui modient leur comportement. Ceci est egalement vrai des instruments. Dans un veritable orchestre, les instrumentistes lisent leur partition en suivant le tempo qui leur est donne par le chef d'orchestre. Mais de temps en temps, ce dernier se tourne vers un instrumentiste et lui donne un ordre particulier. D'autres ordres ponctuels apparaissent sur la partition, au-dessus de la portee. Dans le cas des danseurs, ces evenements peuvent être par exemple l'arrivee au bord de la scene ou le contact avec un partenaire.

Des evenements similaires peuvent se produire pour les danseurs et les instruments de Whizz. Chaque danseur ou instrument est sensible a un ensemble d'evenements, et possede des comportements associes. Ces evenements peuvent être produits en particulier lors des collisions avec d'autres danseurs, ou lorsque des danseurs traversent des zones particulieres de l'ecran.

5.2 Le modele de Whizz : ots et evenements

La metaphore musicale decrite plus haut repose sur un modele sous-jacent plus abstrait et plus general, que nous allons decrire maintenant.

Le modele de Whizz repose sur deux modes de propagation de l'information entre objets. Il rend ainsi compte du fait que les phenomenes dynamiques, et en particulier l'animation, ont deux modes d'evolution. Tout d'abord, on trouve les evolutions qui representent un phenomene continu. Elles sont obtenues par l'echantillonnage a intervalles reguliers d'une grandeur representant ce phenomene. Le deplacement d'un objet graphique sur un chemin, represente par une suite de positions, ou un son numerise, represente par une suite d'amplitudes, sont des exemples de telles

evolutions. Mais il existe aussi des evolutions plus brutales, isolees dans le temps, comme le debut d'un son, le changement de couleur d'un objet graphique, ou la disparition du même objet graphique. Dans Whizz, ces deux types d'evolutions sont representees respectivement par des ots de donnees et des evenements.

Les evolutions continues sont representees par un modele de ot de donnees. Des ux de donnees (les *notes*) sont emis par des objets nommes *modules*, puis transformes ou captes par d'autres modules. La reception d'une note par un module est susceptible de provoquer une action de sa part. En particulier, certains modules ont une apparence graphique qui evolue en fonction des notes recues. La propagation des notes se fait par vagues. Lorsqu'un module recoit une note, il la traite, ce qui provoque eventuellement l'emission d'autres notes. Ainsi, l'emission d'une note par un module declenche une vague de notes et d'actions, qui se propage dans le graphe des connexions, qui doit être acyclique. La propagation de la vague et les actions associees sont considerees comme instantanees. Toutes les actions provoquees par une même note sont donc simultanees.

Les evolutions ponctuelles sont representees par des evenements. Un evenement peut être emis par un module lorsque certaines conditions se presentent ou lorsqu'une modication se produit dans l'environnement. Les evenements sont ensuite captes par tous les modules interesses, et provoquent des actions de leur part. Les evenements fournissent un moyen de communication moins structure que les ots decrits precedemment. La raison a cela est le grand nombre de types d'evenements susceptibles de se produire, et le relativement faible nombre d'evenements se produisant reellement.

Nous allons maintenant etudier les differentes entites qui interviennent dans le modele de Whizz. Tout d'abord, nous examinerons les composants servant a etablir les ots de donnees, ainsi que la maniere dont on peut les structurer. Nous nous interesserons ensuite a l'emission et au traitement des evenements. Nous detaillerons enn les mecanismes de propagation de l'information.

5.2.1 Modules, notes et connecteurs

Whizz repose sur les notions de modules, de notes et de connecteurs. C'est avec ces elements de base que l'on fabrique l'ossature des animations. Les evenements et leur gestion ont aussi leur importance, mais ce ne sont pas eux qui imposent la structure d'une animation. Il faut voir une animation construite avec Whizz comme une structure de ots de donnees, a laquelle s'ajoute une communication par evenements, avec une infrastructure plus legere.

Le ot de donnees est donc realise avec des modules connectes les uns aux autres, et qui se transmettent des donnees sous forme de notes. Selon son type, un module possede un certain nombre d'entrees et de sorties, que nous nommerons *connecteurs*. Certains modules ne possedent pas de connecteur d'entree ; d'autres n'ont pas de connecteur de sortie. On trouve aussi des modules dont le nombre de connecteurs peut

changer dynamiquement. Cependant, la plupart des modules possedent un nombre xe et non nul d'entrees et de sorties.

Comme leur nom l'indique, les connecteurs servent a etablir des connexions entre modules. Chaque connecteur possede un type, qui determine quel genre de notes peut circuler a travers lui. On peut relier un connecteur de sortie a un connecteur d'entree s'ils ont le même type. Une entree ne peut être connectee qu'a une sortie, mais une sortie peut être connectee a plusieurs entrees. Une note emise sur la sortie sera alors transmise simultanement aux entrees connectees. Le module, lui, n'a pas a se preoccuper du nombre de modules qui lui sont connectes. Il se contente d'emettre une note sur une sortie, et c'est le connecteur qui prend en charge les tâches de communication.

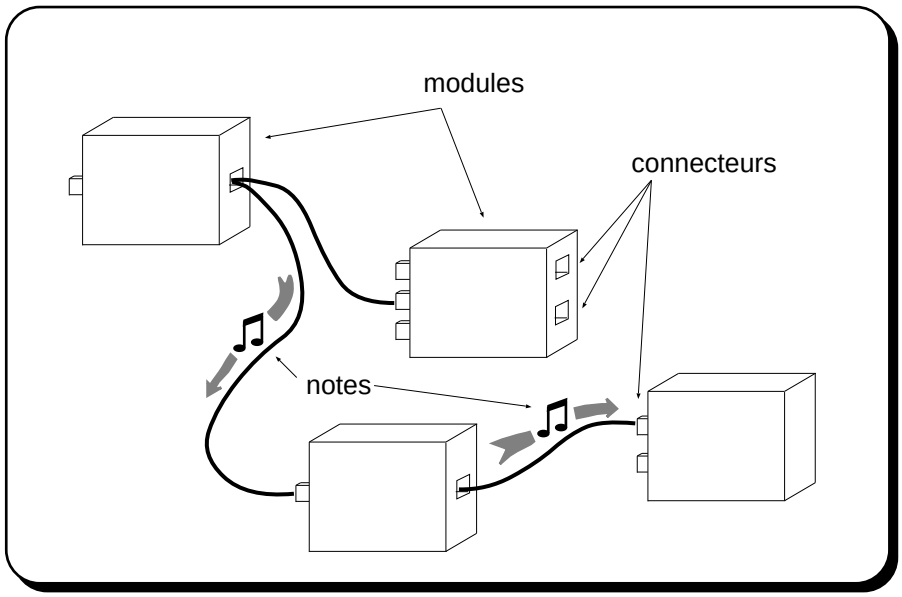

Figure 5.2 : On obtient le ot de donnees en connectant des modules.

Les notes sont des objets transportant des informations, generalement de petite taille. Elles possedent un typeet des informations speciques dependant du type. Les notes les plus simples sont des impulsions, qui ne transportent aucune information. Les autres notes les plus courantes transportent des nombres entiers ou ottants, et des positions. L'animation est obtenue grâce aux modules qui recoivent, interpretent et reemettent des notes.

Ce sont donc les modules qui, en fonction de leur type, donnent un sens aux notes. Certains modules emettent spontanement des notes. Mais la plupart des modules ne font que reagir aux notes qu'ils recoivent. Certains les transforment, tout comme des fonctions produisent des valeurs en fonction des arguments qu'on leur passe. D'autres reagissent aux notes recues sous la forme de comportement perceptibles. Cette structure permet de separer les donnees, qui circulent entre les modules, et les traitements ou

les comportements, qui sont dans les modules. Une consequence interessante est que l'on peut fabriquer des presentations animees uniquement en creant, en parametrant et en reliant des modules. D'une part, cela permet de realiser plus facilement des outils de construction interactifs, comme l'editeur Whizz'Ed qui sera decrit a la section 5.7. D'autre part, cela permet a ces outils de construction de produire une presentation animee sous la forme de donnees (des modules et des connexions), plutôt que sous la forme d'un programme.

Nous pouvons maintenant revenir a la metaphore musicale de Whizz, et voir comment elle s'interprete dans ce modele. Les tempos, rythmes, instruments et danseurs sont des modules. Ce sont les tempos qui provoquent la propagation des notes. Ils emettent a intervalles reguliers des notes qui ne portent aucune information, comme des impulsions d'horloge. Ces impulsions sont transmises aux modules connectes a la sortie des tempos. Ainsi, ce sont les tempos qui encapsulent tout le traitement du temps reel. Les autres modules ne font que repondre aux sollicitations. Les rythmes, par exemple, se comportent comme des ltres pour les impulsions. Ils n'emettent sur leur sortie que certaines des impulsions qu'ils recoivent. Les instruments, quant a eux, emettent des notes plus evoluees en reaction aux impulsions qu'ils recoivent. Enn, les danseurs reagissent aux notes de maniere perceptible par l'utilisateur.

Avec les modules que nous avons decrits, le cheminement normal de l'information est donc le suivant : un tempo emet une impulsion, qui est ltree par un rythme ; si le rythme laisse passer l'impulsion, elle est recue par un instrument ; en reponse, l'instrument emet une note contenant une information ; cette note est alors recue par un danseur, qui agit en consequence en se deplacant, en se deformant, ou en produisant un son par exemple. Cette architecture ou les notes sont progressivement enrichies est vite limitee. En effet, on a souvent besoin de manipuler des notes, par exemple pour additionner deux positions. Certains systemes permettent une veritable programmation a base de ots de donnees. C'est le cas de nombreux systemes de programmation visuelle [Hils 91]. Fabrik, quant a lui, utilise ce modele pour programmer graphiquement des interfaces [Ingalls et al. 88].

Sans aller jusqu'a la programmation graphique, Whizz propose un certain nombre de modules destines a transformer et composer des notes. Certains modules permettent des operations sur des notes d'un même type. Ils effectuent par exemple des transformations geometriques sur des positions. D'autres modules realisent des conversions entre des types de notes differents. C'est avec ces modules que l'on peut transformer un nombre ottant en une position sur un segment, ou au contraire calculer un nombre a partir d'une position. Ces modules qui effectuent des calculs sont collectivement regroupes sous le nom de *modules de ltrage*.

5.2.2 *Scenes d'animation*

Les modules peuvent être organises en *scenes* d'animation.[1] Les scenes sont des groupements de modules connectes entre eux. De plus, les scenes sont elles-mêmes des modules, et peuvent être utilisees comme telles. Elles possedent donc des connecteurs vers l'exterieur. Ces derniers permettent d'\exporter" certains connecteurs des modules assembles dans la scene. On peut ainsi voir une scene comme un circuit integre, qui contient des composants relies entre eux et possede des connecteurs externes relies a certains de ses composants.

D'autre part, plusieurs instances d'une même scene peuvent être creees. Ainsi, les scenes permettent la fabrication et la reutilisation de representations ou de comportements complexes. La gure 5.3 montre comment on peut creer une representation

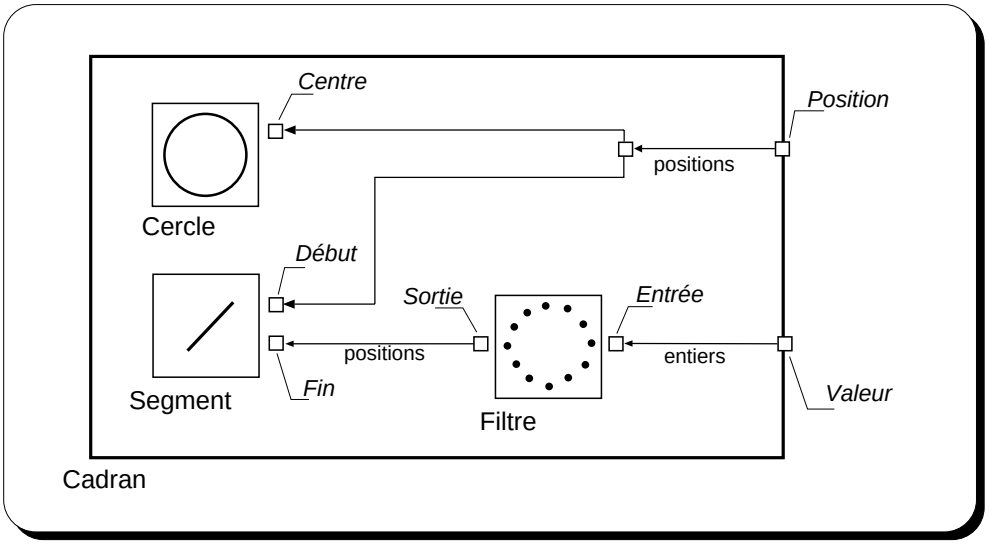

Figure 5.3 : Une scene representant un entier par un cadran. Le centre du cadran et le debut du segment sont confondus, et determines par l'entree Position ; la n du segment est calculee par un instrument qui transforme des entiers en positions sur un cercle. Le representation ainsi construite aura une apparence similaire au cadran de la gure 5.4.

complexe pour un entier, et l'isoler dans une scene. Il suft ensuite de creer une nouvelle instance de cette scene a chaque fois que l'on souhaite representer un autre entier. On pourra aussi l'inclure dans une autre scene, en lui adjoignant par exemple un module de ltrage qui limitera les valeurs entre 0 et 100.

[1] L'ambigute entre les *scenes* d'animation (en anglais \animation scenes") et les *scenes* d'X_{TV} sur lesquelles evoluent les acteurs (en anglais \stages") est involontaire.

Les scenes d'animation ont des utilisations variees, en fonction de la nature des modules qu'elles contiennent. Ainsi, l'exemple de la gure 5.3 est une scene destinee a representer des valeurs entieres sous la forme d'un cadran, et a faire varier cette representation en fonction des donnees recues. Une scene peut aussi servir a produire une animation transitoire, comme un rectangle associe a une trajectoire et un tempo. Dans ce dernier cas, l'instanciation de la scene produit l'apparition et le mouvement du rectangle. Les scenes de Whizz ont donc un caractere tres general, et nous verrons au chapitre 7 qu'elles sont utilisees de diverses manieres dans Witness.

5.2.3 Evenements

Le mecanisme d'evenements de Whizz tient une part importante dans la construction des interfaces animees. En effet, le cours d'une animation autant que l'interaction avec l'utilisateur sont ponctuees par divers evenements isoles dans le temps : collision entre danseurs, arrivee a la n d'une trajectoire, ou clic sur un danseur. Les evenements sont donc le complement indispensable du mecanisme de ot de donnees.

Les evenements sont des objets fabriques et diffuses pour rendre compte de circonstances particulieres ou de changements subits de l'environnement. Un evenement contient un type, un module d'origine, et des informations dependant de son type. Le type d'un evenement determine la nature de l'incident ou des conditions qui ont ete satisfaites : action de l'utilisateur, expiration d'un delai, egalite entre deux valeurs, etc. Il determine aussi le sens des autres informations transportees. Le module d'origine est celui qui a produit l'evenement, soit parce qu'une modication interne lui a fait remplir certaines conditions, soit parce que l'environnement a agi sur lui. Enn, les informations supplementaires precisent les circonstances de l'evenement.

Il existe de nombreux types d'evenements, que l'on peut regrouper en trois categories. La premiere categorie est celle des evenements produits par l'utilisateur, par exemple quand il clique sur un danseur graphique. La seconde categorie rassemble les evenements produits en cours d'animation, et lies a l'evolution des modules, comme par exemple la collision de deux danseurs. La derniere categorie est celle des evenements \synthetiques", qui sont le resultat de transformations d'evenements concrets, ou qui sont emis par le noyau fonctionnel, comme la creation de nouvelles donnees a representer.

Les evenements sont diffuses sur la base de l'abonnement. Un module peut demander a recevoir un type particulier d'evenements pour un module d'origine ou un groupe de modules d'origine possibles. Lorsqu'un tel evenement se produit, il est distribue a tous les modules interesses.

Lors de l'abonnement d'un module a un type d'evenements, il faut specier le

comportement associe a la reception d'un evenement de ce type. Tous les comportements sont possibles, depuis les changements d'aspects jusqu'aux instanciations de nouveaux modules ou les modications de connexions. Les modules de Whizz possedent tous des comportements predenis, en fonction de leur type. Par exemple, ils possedent tous un comportement d'auto-destruction. Cependant, il est souvent necessaire d'ajouter des comportements specialises pour chaque application developpee avec Whizz.

Lors du traitement des evenements, les modications du graphe de ots de donnees sont tres courantes. Lorsque le noyau fonctionnel emet un evenement, il faut pouvoir creer des objets graphiques ou declencher une animation. Cela passe par l'instanciation et la connexion de nouveaux modules. Quand un instrument atteint la n de sa trajectoire, on doit pouvoir instancier de nouveaux modules, mais aussi eliminer des connexions, voire detruire des modules. Ces tâches de reconguration, instanciation et destruction peuvent être prises en charge par un gestionnaire du graphe des modules. Cela permet de decrire les recongurations a effectuer, plutôt que de devoir les programmer. On peut ainsi decrire l'ensemble des modules a creer pour obtenir une representation animee d'une donnee lorsque l'evenement signalant la creation de cette donnee est recue.

Un tel gestionnaire de modules et de connexions est a l'etude, mais n'existe pas dans Whizz a l'heure actuelle. Sa realisation pose un probleme principal : l'acces aux modules deja crees. Supposons qu'en reponse a un evenement on veuille creer un clignotant, et le connecter a un tempo global, existant deja. Pour etablir la connexion, il faut pouvoir acceder a ce tempo, c'est a dire le nommer. Ce qui est facile a faire dans un programme l'est moins quand on denit le schema de gestion d'un evenement avec un outil interactif. Il faut pour cela disposer d'un mecanisme de nommage des objets (modules ou connexions) qui permette de les referencer facilement apres leur creation. De plus, il faut fournir des mecanismes d'indirection : si l'evenement declenchant la creation d'un clignotant contient une frequence, il faut trouver le tempo qui a cette frequence particuliere. De même, dans un systeme d'animation de programmes, une operation sur une variable pourra declencher une animation utilisant la representation de cette variable. Cela signie qu'il faudra referencer la representation a partir d'informations provenant de la variable. En raison de ces problemes, la gestion des instanciations et des recongurations du graphe est donc encore une tâche de programmation avec Whizz.

5.2.4 La propagation de l'information

La propagation des diverses informations tient une place centrale dans Whizz. Nous allons donc examiner en detail le mecanisme operatoire de cette propagation. Pour cela, il faut se souvenir qu'il existe deux types d'information a propager : les notes

et les evenements. Il faut aussi connaître les conditions dans lesquelles ces informations sont fabriquees et emises.

La premier cas d'emission d'un evenement est relativement simple. Il s'agit de l'emission declenchee par l'environnement, en dehors de toute propagation de note : action de l'utilisateur ou changement d'etat du noyau fonctionnel. Comme dans un systeme graphique, ces evenements sont traites de maniere asynchrone, mais leur traitement resulte en une action immediate. Le traitement d'un evenement est donc une operation atomique, que rien ne vient perturber.

L'autre cas d'emission provoquee par l'environnement est celui de l'emission spontanee d'une note. Cette note se propage alors de module en module, provoquant une vague de transmission de notes et d'actions en reponse. Au cours de la propagation de cette vague, des evenements sont susceptibles de se produire, comme par exemple la collision de deux danseurs. Cette emission d'evenements en cours de propagation de vague est le troisieme et dernier type d'emission d'information. Nous allons maintenant voir comment sont geres la vague de notes et les evenements qu'elle provoque.

Dans Whizz, l'hypothese est faite que la propagation et le traitement des notes sont instantanes. Whizz rejoint ainsi Esterel et son hypothese de synchronisme. Neanmoins, il faut distinguer plusieurs phases dans ce traitement, et cela pour deux raisons. Tout d'abord, la partie graphique de l'animation est un point critique. Lorsqu'une note provoque plusieurs modications sur la même partie de l'afchage, il faut eviter de repercuter ces modications independamment. Il arrive que la boîte a outils et le systeme de fenêtrage sous-jacents regroupent les ordres graphiques, ce qui evite des effets visuels deplaisants. Mais quoi qu'il arrive, la gestion des ordres de modication de l'ecran est coûteuse, et il faut eviter de multiplier les calculs. L'autre raison est liee au traitement des evenements produits en cours de propagation. Prenons l'exemple de la collision entre deux danseurs. Celle-ci est detectee au moment ou les danseurs recoivent la note qui les fait bouger. Or le traitement d'un tel evenement peut faire appel a l'etat d'autres objets. Imaginons, dans un jeu, que la collision qui vient de se produire double le score. Imaginons par ailleurs que ce score diminue tout seul au cours du temps, comme cela se produit souvent. Si l'impulsion qui a provoque le mouvement des danseurs et leur collision est aussi utilisee pour faire diminuer le score, on se retrouve face a un probleme d'ordre des traitements. En effet, il est different de doubler le score puis le diminuer, ou de le diminuer puis le doubler. Or l'ordre dans lequel deux modules percoivent la même note est inconnu. Pour assurer le determinisme, le traitement des evenements est repousse apres toute propagation de notes. Ainsi, dans notre exemple, le score est d'abord diminue en vertu de la note qu'il recoit, puis double en vertu de l'evenement de collision.

La gestion de la propagation est assuree par un objet particulier : le *chef d'orchestre*. C'est lui qui permet le bon sequencement des operations. Tout d'abord, une source de mouvement (tempo, valeur active ou action) fabrique une note et commence sa

propagation. Le chef d'orchestre est averti, et est alors pr^et a recevoir deux types de sollicitations. D'une part, certains modules recoivent une note, et demandent a ^etre prevenus de la n de la propagation. En particulier, les danseurs ayant une apparence graphique utilisent ce mecanisme pour se redessiner une seule fois. D'autre part, des evenements sont declenches au cours la propagation des notes : collisions ou ns de partition par exemple. Le chef d'orchestre enregistre ces evenements dans une le d'attente, et les distribue a leurs destinataires lorsque la propagation des notes est terminee, ce qui garantit un certain determinisme. Bien entendu, l'ordre des evenements est indetermine, mais ce non-determinisme est plus acceptable. Finalement, lorsque les notes sont propagees, et les evenements distribues et traites, le chef d'orchestre informe les danseurs qu'ils peuvent bouger, ce qui assure que l'afchage est modie une seule fois.

5.3 Les composants de base

Nous avons examine a la section precedente les deux mecanismes utilises par Whizz pour decrire des animations : les modules et les evenements. Nous allons maintenant etudier plus en detail les diverses categories de modules offerts par Whizz, ainsi que les types d'evenements les plus courants et la maniere dont ils sont emis.

5.3.1 Les sources du mouvement

Dans Whizz, les mouvements sont principalement provoques par la propagation de notes. Un certain nombre de modules sont a l'origine de cette propagation. Ils fabriquent des notes de leur propre initiative, ou plut^ot en fonction de l'environnement exterieur : temps, actions de l'utilisateur, etc. Ce sont ces differents types de modules qui permettent d'obtenir les differents types de comportement dynamique decrits au chapitre precedent.

Animation par le temps

La production de notes en fonction du temps est realisee par les *tempos*. Ces derniers sont des modules que l'on cree en speciant un intervalle de temps. Ils possedent une sortie, sur laquelle des impulsions sont emises periodiquement selon l'intervalle prescrit. Par ailleurs, les tempos possedent une entree qui permet de modier dynamiquement la frequence. De cette maniere, on peut par exemple associer la frequence a une variable entiere du noyau fonctionnel. Cela offre un moyen original de presenter des donnees (un clignotant a vitesse variable, par exemple).

Animation par l'utilisateur

Nous avons plusieurs fois mentionne l'interet de considerer l'activite de l'utilisateur comme une source d'animation. Ce souhait est pris en compte dans Whizz, sous deux formes. En effet, un des principes de Whizz est de distinguer phenomenes continus et evenements ponctuels, et cette distinction s'applique aux actions de l'utilisateur. Si cliquer sur une icone est une action ponctuelle, deplacer la souris en maintenant le bouton enfonce est une action qui s'inscrit dans la duree. Les diverses boites a outils nous ont habitue a voir cela comme une suite d'evenements : l'enfoncement du bouton, une serie de deplacements, et le relachement du bouton. L'optique choisie par Whizz est celle d'une action continue, avec un debut et une n. Les actions continues sont concretisees dans Whizz sous la forme de modules. Ces modules possedent une sortie, et emettent des notes pour rendre compte du deroulement de l'action. Dans le cas des actions cliquer-tirer, les notes contiennent des positions. Pour la rotation d'un potentiometre, elles contiennent des valeurs numeriques. Ces modules doivent etre instancies et connectes lorsque l'action debute. Ils sont detruits lorsqu'elle s'acheve, apres avoir emis un evenement pour en signaler la n.

Dans le cas d'une interface utilisant une souris et un clavier, on pense principalement aux actions de deplacement de la souris comme actions continues. Cependant, les actions de saisie au clavier peuvent rentrer dans le meme cadre. Toutefois, le debut et la n de l'action sont moins clairs que pour les actions cliquer-tirer. On peut decider qu'une action de saisie est une ligne de texte, delimitee par des caracteres \retour-chariot". On peut aussi decider de terminer une action de saisie quand un delai maximum entre deux pressions de touches a ete depasse. Le choix releve des techniques d'interaction de base, et Whizz repose pour cela sur les mecanisme fournis par la boite a outils graphique utilisee.

D'autres types d'animation provoquees par l'utilisateur sont appeles a prendre de l'importance. En effet, nous avons mentionne au chapitre 2 qu'une tendance se dessinait vers des peripheriques d'entree qui rendent compte de l'etat de l'utilisateur, sans action volontaire de sa part. Ces peripheriques fournissent des ots continus d'information. Par exemple, certains emettent la position de l'utilisateur ou d'une partie de son corps. On peut donc envisager d'utiliser ces ots continus de donnees pour initier des mouvements. Il suft pour cela de creer des modules specialises qui emettent ces informations sous forme de notes.

En plus du mecanisme qui fait correspondre des modules aux actions continues de l'utilisateur, Whizz permet de traiter des actions ponctuelles repetitives comme des impulsions. Pour cela, Whizz fournit des modules qui se comportent comme des tempos, mais sont actives par des actions ponctuelles de l'utilisateur plutot que par le temps. Une application interessante est la mise au point d'animations dirigees par le temps : plutot que de connecter un tempo, on connecte un module sensible aux clics

de la souris, et l'utilisateur maîtrise ainsi l'execution de l'animation.

Animation par les donnees : valeurs actives

Lorsque l'on parle d'animation continue dirigee par le noyau fonctionnel, il s'agit surtout d'evolution de variables. Et en effet, le noyau fonctionnel peut provoquer des animations grâce a un mecanisme deja connu : les valeurs actives, que nous avons rencontrees au chapitre 2. Dans Whizz, les *valeurs actives* se comportent comme des variables du côte du noyau fonctionnel et comme des modules du côte de Whizz. Lorsque la valeur est modiee, une note contenant la nouvelle valeur est emise. Le type du connecteur de sortie et celui des notes emises dependent du type de la variable. A l'heure actuelle, les types scalaires (entiers, nombres ottants, caracteres) sont proposes.

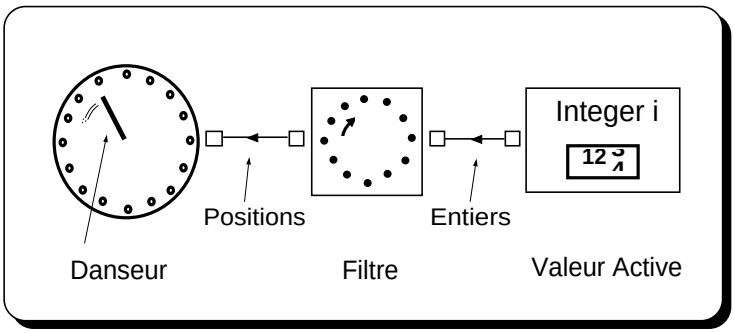

Figure 5.4 : Une valeur active commande l'animation de l'aiguille d'un cadran.

5.3.2 *Synchronisation*

Whizz permet la synchronisation entre differents mouvements. Cela signie par exemple qu'il est possible de faire se deplacer deux objets graphiques aux mêmes instants. Il est aussi possible de denir des synchronisations plus complexes, comme celles des aiguilles d'une horloge : la petite aiguille avance 60 fois moins vite que la grande. Le sens donne ici au mot \synchronisation" n'est pas le même qu'en programmation parallele. Ici, on ne resynchronise pas deux processus normalement asynchrones, mais on denit de quelle maniere ils sont synchrones.

La synchronisation est obtenue grâce a deux mecanismes. D'une part, le modele de ot de donnees permet a plusieurs modules de recevoir une note au même instant (moyennant l'hypothese de synchronisme). Cela permet le parallelisme exact entre deux mouvements. Les synchronisations plus complexes, quant a elles, sont obtenues grâce aux *rythmes*.

Les rythmes sont des modules qui ltrent leurs entrees. Ils sont surtout destines a ^etre utilises derriere des tempos ou d'autres modules qui produisent des impulsions. Un rythme possede une entree et une sortie. Il emet seulement une partie des notes qu'il recoit, sur un simple critere numerique. Par exemple, un rythme simple peut laisser passer une impulsion sur deux, de maniere alternative. Connectons lui un clignotant, c'est a dire un danseur qui change de couleur a chaque note recue. Connectons en parallele un autre clignotant, a travers un rythme qui laisse passer toutes les impulsions. Alors, le premier clignotant aura une frequence deux fois plus faible que le second (voir gures 5.5 et 5.6).

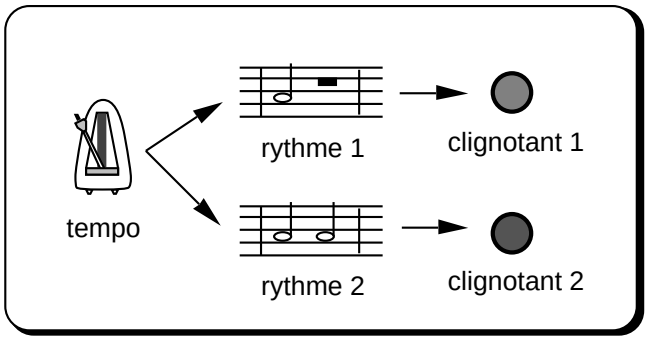

Figure 5.5 : Deux clignotants connectes au m^eme tempo a travers deux rythmes differents.

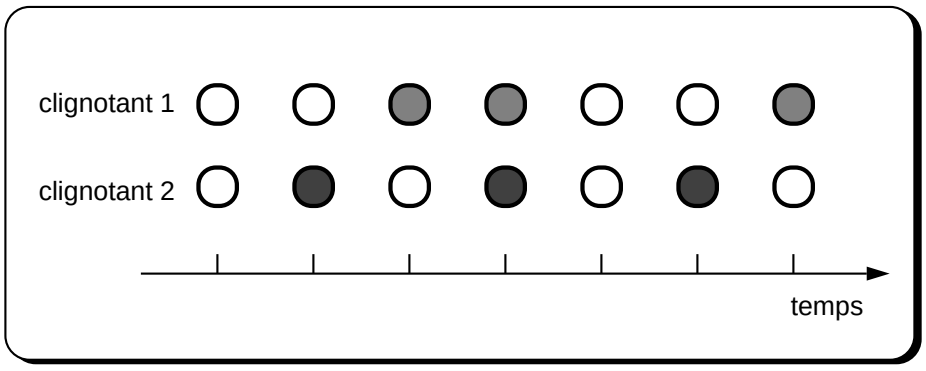

Figure 5.6 : Le premier clignotant est deux fois plus lent que le second.

5.3.3 Description du mouvement

La description du mouvement est la partie de l'animation qui a ete la plus etudiee dans les travaux anterieurs. C'est le centre d'inter^et principal dans l'animation des images de synthese, et les recherches sur la construction d'animations pour les

interfaces ont aussi suivi cette voie. Neanmoins, les solutions proposees dans ce dernier domaine ne sont pas totalement satisfaisantes. Les contraintes temporelles d'Animus, qui permettent surtout d'exprimer des lois physiques, semblent trop abstraites pour exprimer facilement des mouvements simples, et surtout construire graphiquement leur description. D'un autre côte, les chemins de Tango, denis point par point, offrent un faible niveau d'abstraction.

La solution proposee par Whizz est inspiree des objets graphiques des boîtes a outils. Elle repose sur la notion de trajectoire. Une trajectoire est une forme geometrique parametree. On l'utilise en donnant un parametrage, c'est-a-dire la maniere de se deplacer sur la trajectoire. Les boîtes a outils graphiques offrent des objets de base, tels que points, segments, rectangles ou polygones. De la même maniere, Whizz offre des trajectoires de base : rectilignes ou circulaires par exemple. Grâce a ces trajectoires abstraites, le programmeur dispose de formes simples, sufsantes dans la plupart des cas. De plus, des operations comme la deformation des trajectoires ou le changement de parametrage peuvent se faire sans calculs a l'echelle du point. De plus, Whizz possede des chemins denis point par point, similaires a ceux de Tango.

Les trajectoires de Whizz sont mises en uvre par des modules : les *instruments.* Ils emettent des notes contenant des positions lorsqu'ils recoivent des impulsions en provenance d'un tempo ou d'un rythme. On peut ainsi realiser une animation simple en connectant un tempo, un instrument, et un danseur. Par exemple, si l'instrument est du type *Rotor,* le danseur se deplacera sur une trajectoire circulaire. Les instruments possedent en general trois entrees : une pour avancer sur la trajectoire, une pour reculer, et une pour l'acces direct a une position donnee sur la trajectoire. Toutes ces entrees n'ont pas toujours de sens. Par exemple, si un instrument emet des positions qu'il lit une par une dans un chier, le retour en arriere et l'acces aleatoire sont delicats. Un mecanisme d'enregistrement de l'historique est souvent indispensable pour permettre le retour en arriere, mais n'a pas ete realise dans Whizz jusqu'a present.

Whizz propose des instruments pour des trajectoires rectilignes, elliptiques, sur des courbes, et des trajectoires denies point par point. Cependant, au cours d'experimentations sur la notion d'interface vivante evoquee au chapitre 4, l'interêt d'autres types de mouvements est apparu. Nous avons en effet mentionne que l'utilite des interfaces vivantes est d'exploiter la familiarite de l'utilisateur avec les lois physiques simples du monde reel. Pour cela, Whizz propose des instruments permettant de simuler de telles lois. Il ne s'agit pas, comme dans Animus, de produire des mouvements respectant de veritables lois physiques. Il s'agit d'obtenir des mouvements qui paraissent naturels, ce qui est souvent different. Whizz fournit des *attracteurs,* qui permettent de simuler la capture d'un danseur par un champ de gravitation. Les attracteurs sont par exemple utilises lorsqu'on lance un danseur vers un autre. Whizz offre aussi des mouvements freines, pour eviter que des danseurs lances partent trop loin, et des ressorts, utilises dans des jeux ou pour simuler la gravite vers le bas de l'ecran.

5.3.4 Realisation du mouvement

Aucun des modules que nous avons decrits jusqu'a present ne possede d'apparence graphique. Ils servent a provoquer ou decrire les mouvements, mais ne sont pas perceptibles par l'utilisateur. Les modules qui utilisent les notes pour produire un effet perceptible sont appeles *danseurs.* Un danseur n'a pas necessairement d'apparence graphique. Ainsi, nous rencontrerons par la suite des danseurs qui produisent des effets sonores. Cependant, les danseurs les plus courants sont des danseurs graphiques. Nous allons maintenant decrire leurs differents comportements.

Deplacement

Le comportement le plus evident est la possibilite de se deplacer. Chaque danseur graphique possede une entree a travers laquelle il recoit des notes contenant des positions. Ces positions determinent les deplacements successifs du danseur.

Changement d'aspect

D'autres comportements sont communs a tous les danseurs graphiques. Ce sont les changements d'apparence : couleur, epaisseur du trait, etc. Les danseurs graphiques possedent un connecteur d'entree pour chacune de ces caracteristiques. Par exemple, ils ont deux entrees de type couleur, pour la couleur du fond et celle du trait. L'arrivee d'une note sur l'une de ces entrees provoque le changement de couleur correspondant.

Deformation

Certains comportements dependent de la forme des danseurs. Ce sont toutes les deformations geometriques, comme l'agrandissement d'un rectangle, l'ajout d'un point a un polygone ou le deplacement d'une extremite d'un segment.

Chaque type de danseur graphique possede des entrees en fonction de sa geometrie. Ainsi, les segments ont une entree pour chaque extremite. Notons que ces entrees viennent s'ajouter a celle deja presente pour les deplacements. On peut donc choisir de deplacer l'une ou l'autre extremite, ou le segment dans son ensemble. De la même maniere, les rectangles, les ellipses et les rectangles arrondis ont un nombre determine de connecteurs d'entrees. La situation est plus complexe pour les polygones et les courbes. Ces derniers ont un nombre variable de sommets. Ils ont donc un nombre variable d'entrees pour modier la position de chacun des sommets. Les seules entrees predenies sont celles qui correspondent aux extremites, an de pouvoir realiser facilement l'animation d'un chemin par exemple. Par ailleurs, il faut un moyen pour modier le nombre de sommets, ce qui a l'heure actuelle est fait grâce a une entree

speciale. Un nouveau sommet est ajoute a la n de la courbe ou du polygone, a la position contenue dans chaque note qui parvient a cette entree.

Changement d'objet graphique

Les comportements que nous avons decrits jusqu'a present permettent de realiser une large palette d'animations, basees sur les transformations d'objets graphiques. Mais un autre mecanisme est necessaire, en particulier pour les animations les plus simples, obtenues par une succession d'images. En effet, ces animations ne rentrent pas dans le cadre de la modication d'un objet graphique. Ce sont en fait plusieurs objets (en general des icônes) qui sont visibles successivement. Ce type d'animation est obtenu grâce a des danseurs particuliers, qui contiennent une liste d'objets graphiques. Une entree permet de selectionner l'objet a afcher. C'est grâce a ces danseurs polymorphes que l'on peut facilement realiser une interface iconique animee, par exemple.

5.3.5 *Evenements lies a l'animation*

Nous avons vu au chapitre precedent que l'animation d'objets graphiques demandait que l'on laisse une certaine autonomie a ces objets. Mais on ne peut laisser les modules maîtres de leur etat et de son evolution que si l'on dispose d'un mecanisme pour être averti lorsque des circonstances particulieres sont reunies. Les evenements fournissent un tel mecanisme. Whizz permet d'associer a chaque module des verications a effectuer au cours de son evolution. Lorsque les conditions recherchees sont remplies, le module emet un evenement pour le signaler. Nous allons maintenant examiner les evenements susceptibles de se produire en cours d'animation.

Tout d'abord, il est necessaire d'introduire des types d'evenements rendant compte de l'evolution des danseurs. En effet, sans avoir besoin de connaître en permanence la position d'un danseur, il faut souvent savoir quand il entre ou sort de certaines zones de l'ecran, que nous nommerons *zones sensibles*. Ces zones peuvent être la surface de jeu, dans le cas d'un jeu de balle, ou le champ d'action d'un autre danseur, ou encore des objets graphiques statiques faisant partie du decor.

Divers types de zones sensibles sont fournies dans Whizz, ainsi que les evenements associes. On utilise ces zones en les sensibilisant a un ensemble danseurs. Par la suite, un evenement est emis a chaque fois qu'un de ces danseurs entre ou sort de la zone. Parmi les zones sensibles, Whizz fournit des lignes droites, ou des demi-plans, selon qu'on considere la frontiere ou la zone qu'elle delimite. Pour ces lignes, il est possible de selectionner les evenements de franchissement, et même de preciser le sens de franchissement interessant. Whizz fournit aussi des zones rectangulaires et elliptiques. Pour ces zones, l'entree, la sortie, ou le franchissement peuvent être

selectionnes. Les evenements de franchissement contiennent des informations sur le point d'intersection : par exemple dans le cas d'un rectangle, la face qui a ete traversee.

Parallelement a la gestion des zones sensibles, il faut aussi pouvoir rendre compte des collisions entre danseurs. Un mecanisme distinct de celui des zones sensibles est fourni, pour deux raisons. D'une part, plusieurs danseurs sont susceptibles de bouger au même moment, ce qui rend la gestion des collisions plus delicate que pour une zone sensible, dont la position est xe pendant le mouvement des danseurs. Ensuite, le fait que les danseurs bougent en même temps rend potentiellement tres grand le nombre des couples de danseurs a tester. Des algorithmes d'optimisation, lies a la repartition des danseurs dans une scene, peuvent être necessaires. Les collisions sont donc detectees par des objets particuliers, appele *choregraphe*, qui gerent une liste de danseurs actifs et une liste de danseurs sensibles. Cette distinction est introduite pour diminuer le nombre de couples a examiner. Lorsqu'un danseur actif rencontre un danseur sensible, un evenement est emis. Un autre evenement est emis lorsqu'ils se separent. Il est possible d'avoir plusieurs choregraphes a la fois, ce qui permet de gerer des groupes de danseurs sensibles les uns aux autres.

Enn, divers types d'evenements rendent compte de l'evolution des modules en general. Par exemple, on peut demander a un tempo d'emettre un evenement a une date donnee, ce qui permet d'introduire des delais. On peut ainsi interrompre une animation ou une action de l'utilisateur au bout d'un certain temps. D'autres evenements interessants sont les evenements *Fin*, qui sont emis lorsqu'un instrument ou un rythme atteint la n de sa partition. On peut selectionner cet evenement pour enchaîner sur d'autres animations. Par exemple, l'evenement *Fin* est utile lorsque l'utilisateur termine une action longue. Nous en verrons un exemple d'utilisation a la section 5.6.

5.3.6 Evenements lies a l'interaction

Un certain nombre d'actions de l'utilisateurs sont prises en compte dans Whizz a travers des modules : ce sont les actions que l'on peut considerer comme continues. Les actions ponctuelles, quant a elles, sont materialisees par des evenements. Dans le contexte habituel, ou l'on dispose d'une souris et d'un clavier, ces actions sont l'enfoncement d'un bouton de la souris, le double-clic s'il est deni par la boîte a outils graphique utilisee, et la pression d'une touche du clavier. Pour chacun de ces types d'evenements le module d'origine est le danseur graphique sur lequel l'action a ete effectuee.

5.3.7 *Evenements synthetiques*

Nous avons decrit dans les sections precedentes les evenements predenis de Whizz, qui sont associes aux actions de l'utilisateur ou au deroulement de l'animation. Les evenements de Whizz sont aussi destines a rendre compte de l'activite du noyau fonctionnel. Par exemple, on peut associer un module a une variable, et lui faire emettre des evenements lorsque des operations sont effectuees sur cette variable.

Les evenements en provenance du noyau fonctionnel ont des formes nombreuses, qui varient d'une application a l'autre. Pour cette raison, Whizz permet la creation de nouveaux types d'evenements. Outre la creation d'\evenements du noyau fonctionnel", ces evenements synthetiques permettent de traduire des evenements de l'interface dans une forme appropriee au noyau fonctionnel. Les modules qui gerent cette communication et cette traduction entre noyau fonctionnel et interface constituent un adaptateur du domaine, au sens du modele Arch.

5.4 *Whizz et X_{TV}*

Whizz a ete realise sous la forme d'une extension a la boîte a outils X_{TV}. Les deux systemes se completent pour permettre la construction d'interfaces animees. D'une part, X_{TV} permet la gestion de l'ecran et des entrees de l'utilisateur, ainsi que la partie statique des interfaces. D'autre part, Whizz permet d'ajouter un comportement dynamique aux objets fournis par X_{TV}. Nous allons dans cette section etudier les interactions entre les deux systemes. Ces interactions se produisent dans trois domaines. Les deux premiers domaines sont la \surface" de l'interaction homme-machine : la gestion du graphique et celle des actions de l'utilisateur. Le troisieme domaine est celui du sequencement des operations, qui conditionne l'integration de l'animation et de l'interactivite.

5.4.1 *Les danseurs graphiques*

Les seules entites de Whizz a posseder une apparence graphique sont les danseurs graphiques. Ce sont donc ces danseurs qui constituent le premier point de contact avec X_{TV}. Un tel danseur doit savoir evoluer en reponse aux notes qu'il recoit, ce que lui permet son statut de module. Il doit aussi savoir s'afcher, et maintenir la coherence entre son etat et son apparence. Les danseurs graphiques sont donc des acteurs d'X_{TV} aussi bien que des modules. Ils sont presents a l'ecran, et evoluent en fonction des modules qui leur sont connectes. On voit ainsi comment X_{TV} prend en charge l'apparence d'une interface, tandis que Whizz permet d'en denir le comportement : l'un gere la scene, et l'autre les coulisses.

La plupart des acteurs-danseurs sont tres similaires aux acteurs predenis d'X_{TV}. Ils sont composes d'un objet graphique et un environnement graphique. Le type de l'objet graphique determine le type du danseur et ses differents connecteurs d'entree : segment, rectangle, ellipse, polygone, etc. Les danseurs polymorphes, quant a eux, contiennent plusieurs objets graphiques, dont l'un est l'objet courant. Les changements d'apparence sont simplement des changements d'objet courant.

5.4.2 Les actions de l'utilisateur

Le second point de contact entre Whizz et X_{TV} est la gestion des actions de l'utilisateur. Dans ce domaine, Whizz utilise les services offerts par X_{TV} aussi bien pour les actions continues que pour les evenements ponctuels. Dans les deux cas, ce sont encore les danseurs-acteurs qui sont la cle de l'integration. En effet, ce sont des objets reactifs, auxquels des ltres d'evenements peuvent être associes. En particulier, un ltre leur est associe pour chaque type d'evenements ponctuels : clic, double-clic, enfoncement d'une touche. Ces ltres ont pour rôle de transformer l'evenement d'X_{TV} en un evenement de Whizz, qui est ensuite emis par le danseur.

Le mecanisme de propagation des evenements de Whizz est distinct de celui d'X_{TV}. Il y a plusieurs raisons a cela. Tout d'abord, les evenements d'X_{TV} sont fortement connotes par la notion d'evenement physique. La notion de dispositif emetteur, par exemple, a peu de sens pour la collision de deux acteurs animes, et on ne souhaite pas recevoir sans discernement tous les evenements de collision emis par un tel dispositif. Par ailleurs, X_{TV} impose l'existence d'une et une seule cible pour chaque evenement. Cette cible unique est signicative d'un systeme construit pour l'interaction graphique. Lorsqu'un instrument atteint la n de sa partition, il n'y a pas de cible privilegiee. La ou les cibles sont les modules interesses par cet evenement. En fait, comme nous l'avons suggere a la n du chapitre 3, les evenements d'X_{TV} sont adaptes au traitement des actions de l'utilisateur, mais moins a la communication entre les diverses composantes d'une application interactive. Les evenements dont nous avons besoin dans Whizz doivent permettre cette communication, dans la mesure ou ils peuvent servir aux couches proches du noyau fonctionnel a declencher des animations.

En ce qui concerne les actions continues, ou interactions longues, Whizz utilise les objets *Action* denis par X_{TV} et decrits a la section 3.5. Les modules qui rendent compte de l'interaction sont des Actions, dont le comportement a ete adapte pour emettre des notes. Ils sont instancies par des ltres associes aux danseurs-acteurs, qui detectent le debut des actions. Ainsi, des le premier evenement *MouseMoveDown* recu sur un danseur-acteur, un module-action *Drag* est instancie. Ensuite, a chaque fois qu'un evenement de deplacement de la souris est recu par la facette action, la facette module emet une note contenant la position de la souris.

5.4.3 *Integration de l'animation temporelle et de l'interactivite*

Nous allons etudier le fonctionnement des tempos, qui representent une partie fondamentale de Whizz. Ils permettent en effet de melanger animation et actions de l'utilisateur. L'integration de la gestion du temps et des autres sources d'evenements est obtenue grâce au mecanisme de gestion des evenements dans X_{TV}. En effet, ce mecanisme permet la creation de nouveaux types d'evenements, qu'ils soient synthetiques ou associes a de nouveaux peripheriques d'entree. Le temps est donc materialise par un nouveau type d'evenements, emis par un dispositif simule que nous nommerons *quartz*.

Chaque tempo possede un quartz associe. Ce quartz declenche l'emission d'evenements temporels a intervalles reguliers, selon sa frequence propre. Un tempo est un objet reactif, au sens de X_{TV}. Il est la cible des evenements emis par son quartz. Ainsi, la gestion des evenements temporels se fait avec le même mecanisme que celle des autres evenements. En particulier, ils sont traites dans la même boucle, ce qui elimine tout risque de blocage.

La gestion du temps que nous venons d'examiner souleve quelques questions. Tout d'abord, c'est une gestion asynchrone. Le traitement d'un evenement temporel par un tempo est synchrone, mais pas sa transmission au tempo. Entre le moment ou le quartz l'emet et celui ou le tempo le recoit, un evenement transite par une le d'attente. D'autres evenements eventuellement presents dans la le sont traites avant lui, ce qui introduit un delai qui peut être perceptible.

Ensuite, la question se pose de la base de temps qui est utilisee pour declencher les evenements temporels. Les systemes graphiques utilisant un modele distribue sont tres repandus. En particulier, l'implementation presente d'X_{TV} et de Whizz utilise le systeme de fenêtrage X. Les dessins et les entrees de l'utilisateur sont traites dans le serveur X, lequel communique avec un client qui contient l'application interactive. Le serveur X et son client ont chacun leur temps propre. Ils peuvent s'executer sur deux machines differentes, et les uctuations du debit de donnees peuvent poser des problemes de synchronisation. Dans le cas de Whizz, le temps utilise est celui du client, et pas celui du processus qui interagit directement avec l'utilisateur (le serveur X). Il peut donc se produire des decalages entre l'animation et les actions de l'utilisateur.

Prenons un exemple simple illustrant ce decalage. Un rectangle se deplace de maniere notable avec une periode *T*. Un utilisateur tente de cliquer sur ce rectangle. Le deplacement du rectangle est tel que s'il clique sur la position precedente, l'utilisateur le rate. Par ailleurs, le mouvement est sufsamment aleatoire pour qu'aucune anticipation ne soit possible. L'utilisateur doit donc attendre de voir le rectangle se deplacer. Puis il laisse s'ecouler un delai *x*, et clique. Supposons qu'il existe un delai de transmission *t* entre le serveur et le client. A chaque fois que *x* sera compris entre *T*-2*t* et *T*, l'utilisateur aura reussi, et le systeme lui dira qu'il a echoue.

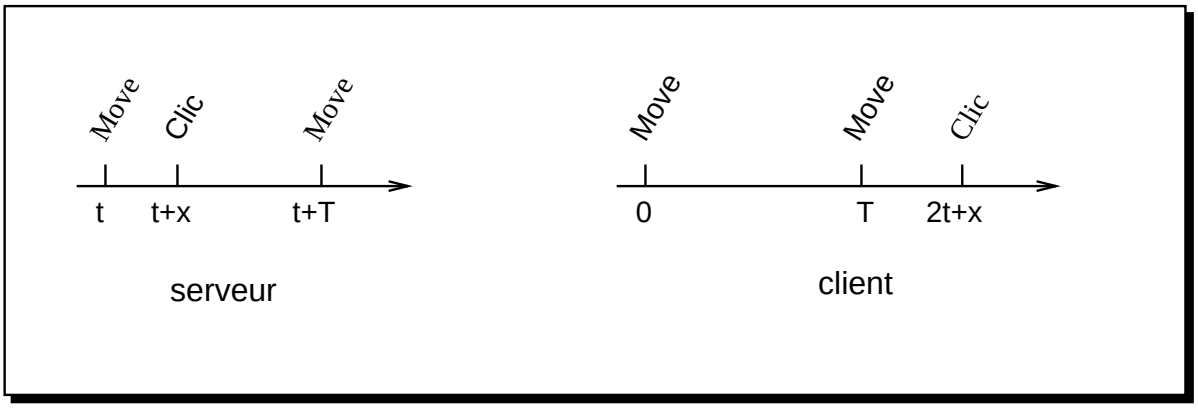

Figure 5.7 : Decalage temporel entre le serveur et le client

On peut repondre a cette objection de deux manieres. Tout d'abord, ce probleme se fait surtout sentir lorsque des evenements ponctuels se produisent. Pour un phenomene repetitif, le delai ne fait qu'introduire un dephasage, qui est rarement perceptible. C'est lorsqu'interviennent le debut ou la n d'un mouvement, ou une action de l'utilisateur, qu'on percoit le decalage dû a ce delai. Les variations du delai sont, elles, beaucoup plus perceptibles. Cependant, de telles variations peuvent se produire aussi pour d'autres raisons : seul un systeme d'exploitation prevu pour le temps reel permettrait de les eviter.

Ensuite, ce probleme n'est pas propre aux animations. Il apparaît des qu'une application interactive modie son apparence. Par exemple, un probleme similaire existe lors de la destruction d'une fenêtre. Si le delai entre l'ordre du client et la destruction par le serveur est trop grand, des evenements peuvent se produire dans une fenêtre que le client croit detruite. L'animation ne fait que systematiser le probleme, en lui fournissant de nombreuses occasions de se manifester. Or ce probleme n'a pas empêche la generalisation des systemes graphiques distribues. En effet, il se trouve que les delais de transmission sont en general negligeables devant les temps caracteristiques d'une application interactive. En particulier, ils sont negligeables devant les temps de reaction de l'utilisateur. On retrouve la la même situation que pour l'hypothese de synchronisme : l'important est que le phenomene paraisse synchrone a son observateur.

5.5 Extensions non graphiques

Whizz est principalement destine aux interfaces graphiques, et a l'animation d'objets graphiques. Toutefois, d'autres moyens de communication dynamiques peuvent apparaître dans les interfaces, et Whizz permet de les decrire. L'exemple le plus accessible sur les ordinateurs actuels, et le plus utilise dans les interfaces existantes, est le

son. La plupart des stations de travail possedent un haut-parleur et un petit synthetiseur de son. Selon les machines, le synthetiseur permet une qualite telephonique ou comparable a un disque compact. Face a cette diversite, le son est peu utilise : il sert surtout a ponctuer l'interaction par des bips ou des bruits divers. Sur le Macintosh, par exemple, il est possible de denir le son qui est emis lorsque l'utilisateur commet une erreur. Des systemes experimentaux comme le Sonic Finder [Gaver 89] utilisent differents sons pour transmettre des informations. Ainsi, lorsqu'on les selectionne, les icônes du Sonic Finder emettent des sons differents selon la nature et la taille des chiers qu'elles representent.

Des extensions ont ete faites a Whizz pour permettre la gestion de ce genre de sons. Helas, il n'existe pas encore d'equivalent du standard X Window System dans le domaine sonore. Le seul standard est le protocole MIDI utilise pour la communication entre synthetiseurs, mais il faut connecter un synthetiseur a l'ordinateur pour l'utiliser. Les extensions ont donc ete realisees en fonction du materiel sur lequel Whizz a ete developpe. Sur cet ordinateur, on peut demander au synthetiseur d'emettre des sons sur trois canaux independants. Un son y est deni par une frequence et une duree.

L'extension sonore de Whizz comprend donc trois danseurs qui representent les trois canaux. Ces danseurs ont chacun une entree de type entier. La valeur des notes qui parviennent a ces entrees est interpretee en Hertz. Lorsqu'une note arrive, le son correspondant est emis par le haut-parleur. Avec un module de ltrage qui traduit des notes de musique en frequences, il est ainsi possible de reproduire des partitions simples. Ces capacites sonores ont aussi ete utilisees pour presenter une valeur active entiere : un module de ltrage numerique transforme la valeur en une frequence.

L'extension que nous venons de decrire est tres modeste, et ne permet de decrire que des ajouts sonores simples a des interfaces graphiques. Neanmoins, il met en valeur le fait que Whizz n'est pas limite a la description du comportement d'objets graphiques, mais peut être adapte a d'autres processus dynamiques. Cela permet en particulier d'envisager l'utilisation de Whizz pour la description d'interfaces multimedias.

5.6 Exemples d'applications

Nous allons maintenant examiner en detail deux applications interactives realisees avec Whizz. Dans la mesure ou le chapitre 7 est essentiellement consacre a l'animation dirigee par les donnees, ces deux exemples illustrent plutôt le melange entre animation temporelle et animation par l'utilisateur. Le premier montre comment ces deux types d'animation peuvent coexister : c'est le classique jeu de casse-briques. Le second exemple illustre la combinaison de ces deux types d'animation, ainsi que la communication avec le noyau fonctionnel : c'est une interface iconique animee.

Le jeu de casse-briques

Le jeu de casse-briques est caracterise par deux comportements dynamiques independants : le mouvement de la balle, qui est autonome, et celui de la raquette, qui est dirigee par l'utilisateur. Nous allons ici considerer une version facile du jeu, ou la raquette repond instantanement aux sollicitations.

On peut isoler trois classes d'objets dans ce jeu : la raquette, la balle, et les briques. Un objet \jeu" contient le mur qui delimite la surface de jeu, et toutes les donnees concernant la gestion de la partie : nombre de points, de balles restantes, etc.

Dans sa partie visible, une raquette est un danseur (polygone ou icône), qui se deplace horizontalement avec les mouvements de la souris. Sa realisation est simple : une raquette est composee d'un danseur et d'un module qui ltre les positions en ne transmettant que les variations sur l'axe horizontal. La sortie du module de ltrage est connectee a l'entree du danseur qui commande son deplacement. Lors de l'instanciation, l'entree du module de ltrage est connectee a la sortie d'un module emettant les mouvements de la souris.

Les balles, quant a elles, se deplacent de leur propre chef. Ils nous faut donc denir un tempo. Nous considerons ici qu'il existe un seul tempo, partage par toutes les balles, pour peu que le jeu autorise plusieurs balles a la fois. Les balles sont des danseurs qui se deplacent chacune sur sa propre trajectoire rectiligne. Pour ajouter un effet visuel, on peut imaginer que les balles tournent sur elles-mêmes. Pour cela, il suft qu'une balle soit representee par plusieurs images entre lesquelles on alterne. La \rotation" peut se faire a un rythme different de la progression sur la trajectoire. Nous arrivons alors a la structure suivante : une balle est composee d'un danseur polymorphe, relie pour sa position a un glisseur (un instrument produisant des positions sur une trajectoire rectiligne), et pour la selection de son image, a un rythme (d'un temps sur trois par exemple).

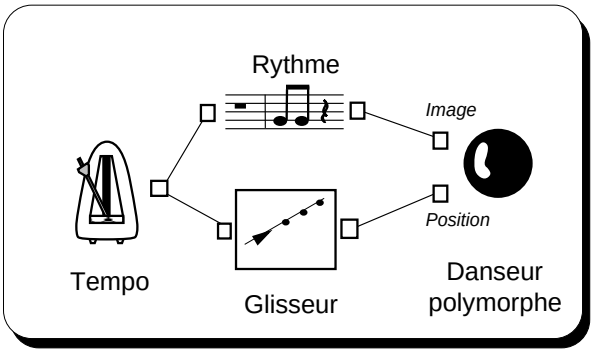

Figure 5.8 : La structure de la balle du casse-briques.

Au contraire des balles et de la raquette, les briques ne sont pas animees. En revanche, elles sont des obstacles pour les balles, et ont une apparence visible. Les briques sont donc composees d'une zone sensible rectangulaire et d'un acteur xe de la même forme (eventuellement une icône, pour un dessin plus soigne). Chaque zone sensible declenche un evenement des qu'une balle y penetre ou la traverse. Cet evenement contient l'information sur la face par laquelle la balle est entree. Les balles et les briques elles-mêmes sont abonnees a ces evenements. Lorsqu'une brique recoit un evenement de franchissement, elle s'auto-detruit. Les balles, quant a elles, modient le vecteur vitesse de leur glisseur, en fonction de la face qu'elles ont heurtee. Les balles ont un comportement similaire vis-a-vis de la zone sensible qui decrivent les limites de la surface de jeu : elles rebondissent lorsqu'elles heurtent le bord.

Une interface iconique animee

Nous allons ici etudier quelques aspects d'une interface utilisant des icônes animees. Nous montrerons de cette maniere comment les constructions proposees par Whizz peuvent servir de support a l'imagination.

Tout d'abord, les icônes doivent être des danseurs polymorphes, comme la balle du jeu de casse-briques. De cette maniere, les icônes peuvent être animees de maniere continue, si besoin est. On peut par exemple imaginer une icône representant un utilitaire de compression de chiers, ou encore une imprimante, et dont l'animation illustrerait la fonction. Mais il serait sans doute lassant de voir une icône animee en permanence, d'autant plus que de telles icônes risquent d'être nombreuses. Il est preferable d'utiliser l'animation pour representer une activite en cours (l'outil de compression est en train de comprimer un chier) ou un etat particulier (l'imprimante est a cours de papier).

Un autre point interessant est celui de l'interaction entre icônes. Dans une interface iconique, certaines operations peuvent être realisees en deposant une icône sur une autre. Par exemple, on depose l'icône d'un chier sur celle de l'imprimante pour lancer l'impression. An que l'utilisateur soit informe des operations possibles, les icônes susceptibles de reagir changent d'aspect lorsqu'on deplace une icône au-dessus d'elles. Il s'agit la d'un retour d'information semantique : le changement d'aspect se produit en fonction du type de l'icône, mais aussi de l'etat des donnees qu'elle represente. On peut imaginer que l'icône d'une imprimante en panne ne reagisse pas. La technique habituelle pour realiser ce type de comportement consiste a interroger le noyau fonctionnel pour determiner le retour d'information qui doit être fourni. Whizz permet d'imaginer une autre technique, ou le retour d'information serait entierement a la charge de l'interface. Pour cela, il suft de le realiser par une scene d'animation, instanciee lorsque l'evenement de collision est detecte. Si l'icône n'est sensible aux

collisions que lorsqu'elle a recu un evenement du noyau fonctionnel le lui ordonnant, le retour d'information dependra bien de l'etat des donnees representees.

Venons-en maintenant a l'idee emise au chapitre 4 : lancer des icônes. Dans les interfaces iconiques habituelles, toutes les icônes peuvent être deplacees. Si on les depose dans des zones actives ou sur d'autres icônes, des operations sont declenchees. Sinon, elles restent simplement la ou on les a amenees. Nous proposons d'etendre la metaphore utilisee en donnant la possibilite de lancer les icônes. Elles peuvent alors penetrer des zones actives et declencher des operations, ou simplement ralentir et s'arrêter. Pour illustrer cette proposition, considerons le lancer d'une icône dans la poubelle. On trouvera dans l'annexe B des extraits d'un programme mettant cet exemple en pratique.

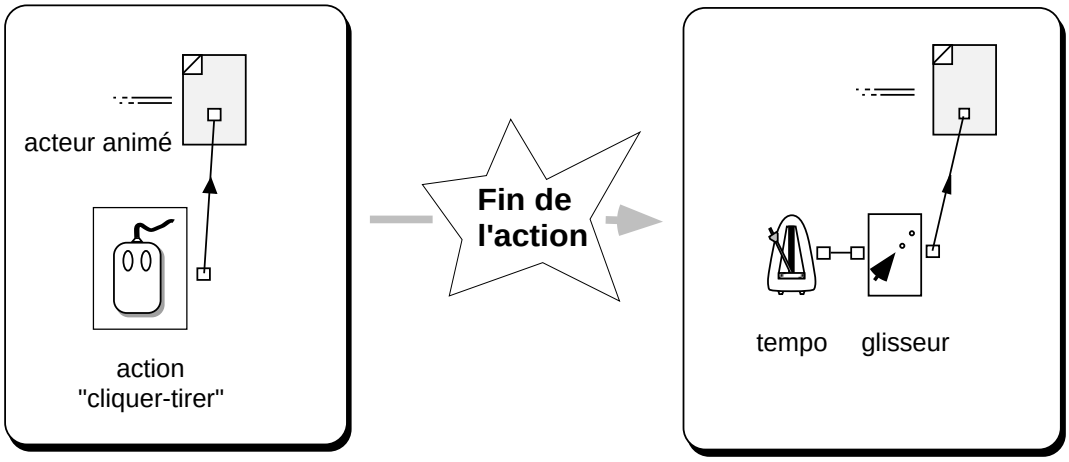

Figure 5.9 : Lancer une icône.

Etudions d'abord la phase du lancer. Dans un premier temps, l'icône est deplacee avec la souris. Ce deplacement est realise avec un des modules-actions decrits a la section 5.3.1, qui emet des notes rendant compte du deplacement de la souris durant l'interaction en cours. La sortie de ce module est connectee a l'entree qui determine la position de l'icône. Lorsque l'utilisateur relâche le bouton de la souris, l'interaction est terminee, et le module-action emet un evenement *Fin* avant de s'auto-detruire. Cet evenement est capte, et provoque l'instanciation d'un tempo et d'un instrument decrivant une trajectoire. Le tempo et l'instrument sont alors connectes a l'icône, qui poursuit le mouvement. Cette phase du lancer est illustree par la gure 5.9.

Examinons maintenant ce qui se passe lorsqu'une icône a ete lancee vers la poubelle. Il nous faut tout d'abord rendre la poubelle sensible aux collisions. L'evenement de collision avec une icône lancee provoquera alors la destruction de l'icône, et l'emission d'un evenement vers le noyau fonctionnel. Cependant, lancer une icône de maniere a ce qu'elle heurte la poubelle demande trop de precision. Une solution consiste a entourer

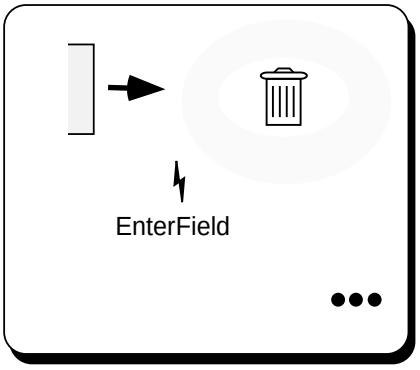

Figure 5.10 : La capture d'une icône par la poubelle.

la poubelle d'un \champ attractif". Pour cela, il suft d'utiliser une zone sensible, qui provoque l'emission d'un evenement lorsqu'une icône y penetre. On peut alors capter cet evenement, instancier un attracteur centre sur la poubelle, et y connecter l'icône qui vient d'être capturee, apres avoir detruit l'instrument qui lui communiquait son mouvement precedent. Ainsi, l'icône continue automatiquement son trajet vers la poubelle avant de la heurter, ce qui provoque sa destruction.

5.7 Whizz'Ed : un editeur de scenes

Whizz est une boîte a outils pour la construction d'interfaces animees. On peut donc utiliser les classes d'objets qu'elle fournit pour programmer de telles interfaces, et les deux exemples precedents donnent une indication sur la maniere de proceder. Mais une des raisons qui ont motive la realisation de Whizz est la possibilite de construire interactivement des objets animes. Le but est par exemple de construire des representations graphiques animees pour des donnees. De maniere plus generale, on souhaite disposer d'un outil similaire aux editeurs d'interfaces, mais permettant en plus d'exprimer le comportement dynamique des interfaces construites. Whizz'Ed est une premiere etape vers un tel outil. C'est un editeur interactif de scenes d'animation, qui combine l'edition de graphes de ots de donnees et l'edition de dessins.

Whizz'Ed comporte tout d'abord une palette, dans laquelle on peut choisir des modules et les ajouter a la scene en construction. Lorsqu'un module est cree, il est represente par une icône dependant de son type, entouree de petites icônes representant les connecteurs. L'utilisateur peut alors creer des liens entre connecteurs par des actions de cliquer-tirer. Certains modules comme les tempos ou les rythmes ne correspondent a aucune realite graphique. Ils ne sont donc rendus visibles que par leur icône, et des sous-editeurs specialises sont necessaires pour les modier. Par exemple, il existe un

editeur specialise pour les rythmes, inspire des editeurs de \bitmaps". En revanche, la plupart des danseurs ont une apparence graphique en plus de leur icône. L'icône sert a etablir les connexions, et l'apparence graphique a manipuler le danseur comme dans un outil de dessin. Ainsi, si l'on veut creer une animation constituee d'un carre qui se deplace, on prendra un acteur anime rectangulaire a qui on donnera la taille voulue, avant de connecter son icône a celle d'un instrument.

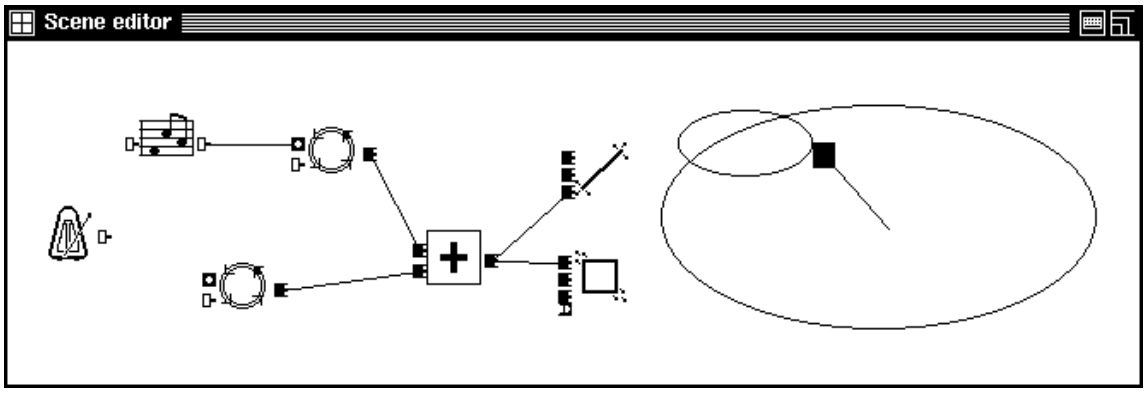

Figure 5.11 : La construction d'une scene avec Whizz'Ed.

Les instruments associes a des trajectoires benecient egalement de cette double representation. Leur apparence graphique represente leur trajectoire, modiable par manipulation directe. Cela permet ainsi de visualiser le trajet des danseurs graphiques. En effet, quand ces derniers sont connectes a d'autres modules, ils s'animent dans Whizz'Ed comme ils le feront lors de leur utilisation. Il est donc utile de montrer la trajectoire pour permettre les mises au point.

La gure 5.11 montre une scene en cours de construction avec Whizz'Ed. Elle comprend dans sa partie droite deux danseurs (un carre et un segment), qui vont s'animer selon une trajectoire denie par la composition de deux trajectoires elliptiques. La partie gauche contient les modules constituant le ot de donnees : un tempo, un rythme, les deux trajectoires elliptiques, un module combinant des positions, et les deux danseurs. Le rythme rend l'une des rotations plus lentes que l'autre. Il reste dans cette scene a connecter le tempo pour que l'ensemble s'anime, permettant ainsi de tester l'animation que l'on vient de construire.

Dans cet exemple, la simple connexion du tempo anime les danseurs de la scene. En realite, les icônes representant les modules s'animent egalement. Ces icônes animees sont revelatrices du fait que Whizz'Ed est realise avec Whizz. En particulier, la creation d'un module a partir de la palette est une veritable creation de module dans Whizz'Ed. Par exemple, l'ajout d'un tempo a la scene en cours de construction se traduit par la creation d'une icône, mais aussi d'un veritable tempo. L'icône est elle-même un danseur polymorphe, qui est connectee a la sortie du module qu'elle represente. Ainsi,

un tempo est represente par une icône a deux etats, representant un metronome dont le balancier se deplace au rythme des impulsions emises. Si l'utilisateur connecte ce tempo a d'autres modules, les icônes de ces derniers s'animeront aussi. De la même maniere, les danseurs sont de veritables danseurs de Whizz, et le dessin qui permet de les manipuler est leur propre apparence. Donc Whizz'Ed ne *represente* pas des danseurs, mais il les *rend visibles*, en les mettant sur une scene (au sens d'X_{TV}).

Whizz'Ed est utilise actuellement pour construire des representations de variables dans l'outil de mise au point Witness (voir chapitre 7). On peut ainsi envisager de construire des editeurs plus evolues, qui permettraient de construire interactivement des elements d'interfaces a manipulation directe.

5.8 Problemes ouverts

Le systeme Whizz et son modele tels que nous venons de les decrire sont utilisables pour realiser de nombreuses applications interactives animees. Cependant, leur mise au point et leur utilisation ont mis a jour un certain nombre de problemes et de directions nouvelles de recherche.

Les problemes les plus evidents sont sans doute ceux qui apparaissent dans Whizz'Ed. Tout d'abord, nous avons vu que certains modules sont representes deux fois, par une icône et une forme geometrique. Or il devient impossible de faire le lien entre ces deux representations des que le nombre de modules augmente. Un autre probleme de Whizz'Ed est l'absence de representation graphique pour les evenements, leur declenchement et leur traitement, ce qui limite son utilisation a des scenes simples. Ces deux lacunes mettent en evidence un probleme general : la creation de formalismes graphiques pour decrire des comportements ou des phenomenes dynamiques est delicate.

L'utilisation de Whizz pour decrire des applications interactives souleve des interrogations qui concernent plus son modele. En particulier, le caractere unidirectionnel de la propagation des notes est parfois desagreable. Il suft d'un exemple simple pour s'en convaincre. Si l'on veut representer une valeur active entiere avec Whizz, on peut utiliser un module de ltrage connecte a l'extremite de l'aiguille d'un cadran. Si l'on veut de plus que l'utilisateur puisse modier la valeur de l'entier en agissant sur l'aiguille, il faut ajouter un autre module de ltrage pour transmettre l'information vers la valeur active. Il serait pourtant tentant d'utiliser le même module de ltrage dans les deux sens, ce que Whizz ne permet pas. On rencontre la un dilemme bien connu des concepteurs de systemes a base de contraintes. La propagation bidirectionnelle est simple en apparence, mais elle pose deux problemes. Tout d'abord, toutes les transformations ne sont pas reversibles simplement, et il faut souvent recourir a des methodes de resolution numeriques coûteuses et complexes. Ensuite, si un module

possede deux entrees et une sortie, il faut choisir l'entree sur laquelle propager les modications venant en sens inverse. Il faut pour cela introduire un mecanisme de priorite ou de hierarchie [Borning et al. 87].

Un autre probleme pose par le modele de Whizz concerne l'impossibilite de lire des informations. En effet, les modules ont un comportement passif vis-a-vis du recueil d'informations. Ils ne sont actives que lorsqu'une note leur parvient, et n'ont a leur disposition que les informations transportees par cette note. Considerons par exemple un danseur qui doit interpreter les positions qu'il recoit comme des positions relatives par rapport a un autre danseur. Il ne dispose actuellement d'aucun moyen d'obtenir la position du danseur de reference, et les solutions possibles en utilisant le modele de Whizz sont inutilement compliquees. Une solution a ce probleme serait d'ajouter dans Whizz la possibilite de lire des capteurs, de la même maniere que dans Esterel.

5.9 Conclusion

Apres le chapitre 4, consacre a l'etude de l'animation dans les interfaces, ce chapitre nous a permis d'etudier le modele et les mecanismes de Whizz, une boîte a outils pour interfaces animees. Nous avons vu comment Whizz, grâce a son modele abstrait, permet de rendre compte des trois types de comportement dynamique d'une interface : ceux provoques par le temps, l'utilisateur ou les donnees. Nous avons aussi vu a travers des exemples comment il permet de melanger les sources de mouvement, et même de passer de l'une a l'autre. L'editeur Whizz'Ed prouve que Whizz peut être utilise pour denir interactivement le comportement d'objets animes. Il ouvre la voie au developpement d'outils plus puissants, qui permettront de fabriquer des interfaces a manipulation directe.

Enn, le modele de Whizz ouvre des perspectives dans le domaine de l'architecture des applications interactives. En effet, nous avons vu dans les chapitres precedents qu'un des problemes les plus importants est la communication entre les differentes parties de l'application. Or le modele de Whizz induit une communication avec le noyau fonctionnel a travers des modules specialises, que l'on peut considerer comme des adaptateurs du domaine au sens du modele Arch. D'un autre côte, l'usage de l'animation pour un meilleur retour semantique amene l'interface a contenir plus de semantique. Les communications avec le noyau fonctionnel prennent donc une importance croissante. Enn, l'identication des differents types de comportement dynamique pourrait être une nouvelle base pour un modele d'architecture. Toutefois, il est trop tôt pour proposer un tel modele.

Chapitre **6.**

L'animation de programmes et d'algorithmes

Un algorithme est la description d'une suite d'operations visant a resoudre un probleme ou accomplir une tâche en manipulant des donnees. Par exemple, on connaît de nombreux algorithmes de tri, qui manipulent un tableau d'entiers a trier, et utilisent parfois des structures de donnees intermediaires : un autre tableau, ou un arbre binaire [Froidevaux et al. 90]. Un programme est la concretisation d'un algorithme dans une forme utilisable par un ordinateur. Il reproduit plus ou moins delement l'algorithme, en tenant compte des contraintes du langage de programmation. Par exemple, peu de langages offrent des arbres binaires, et ces derniers sont alors construits par le programmeur a partir d'autres types de donnees. La lecture d'un programme n'est donc pas toujours revelatrice de l'algorithme sous-jacent. De maniere generale, la lecture d'un algorithme ou d'un programme ne permettent pas une comprehension facile des structures de donnees que l'on manipule, et un dessin est souvent necessaire.

Par ailleurs, programmes et algorithmes sont destines a être executes. Au cours de cette execution, les structures de donnees vont être modiees. Certaines vont même être creees, puis detruites. Le ot d'execution va passer par des tests et des boucles, repasser par le même point du programme avec des donnees differentes ... et vite saturer les capacites de memoire de celui qui tente de comprendre l'algorithme. Pour cette raison, l'explication d'un algorithme est une tâche ardue. Dans le cadre d'un cours au tableau noir, cette explication se fait en general avec un ensemble de dessins representant les donnees a des moments cles de l'execution. Ces dessins sont par ailleurs accompagnes de force gestes, pour indiquer l'enchaînement des operations. L'animation d'algorithmes consiste a remplacer ces dessins et ces gestes par une representation animee de l'execution de l'algorithme. On l'utilise pour l'enseignement, la recherche en algorithmique, et aussi pour la mise au point de programmes.

L'animation de programmes est un domaine relativement ancien, et l'une des premieres applications de l'animation en informatique. On considere qu'elle est apparue

au milieu des annees 1970 a l'Universite de Toronto, avec des travaux qui ont mene a la realisation du lm \Sorting out Sorting" [Baecker 81], presente a la conference SIGGRAPH'81. Depuis cette epoque, des travaux sont regulierement consacres a la maniere dont on peut creer des vues animees d'un programme ou d'un algorithme. Nous allons dans ce chapitre nous interesser aux differentes techniques mises en uvre dans ce domaine. Dans un premier temps, nous allons etudier les problemes generaux poses par l'animation de programmes et d'algorithmes. Nous examinerons ensuite les systemes existants les plus representatifs. Enn, nous etudierons les problemes speciques poses par la visualisation de programmes centree autour des donnees et de leurs operations.

6.1 Techniques d'animation

Nous allons dans cette section nous interesser aux aspects techniques de l'animation d'algorithmes. Tout d'abord, nous essaierons de determiner ce qui doit être represente pour animer un algorithme : quels phenomenes, quels objets. Nous examinerons ensuite les mecanismes qu'il faut fournir pour permettre cette animation. Enn, nous effectuerons une distinction entre animation d'algorithmes et animation de programmes.

6.1.1 Representer un algorithme

Pour determiner ce qu'il faut representer dans l'animation d'un algorithme, analysons la denition que nous en avons donnee. La premiere caracteristique d'un algorithme est qu'il est compose d'une suite d'operations, et que son execution est elle aussi composee d'une suite d'operations. On ne comprend un algorithme que si on comprend comment ces operations s'enchaînent. Pour cela, il faut d'abord determiner quelles sont les operations signicatives. En particulier, il existe rarement un lien direct entre operations de l'algorithme et instructions du programme associe. Parfois, une operation sera l'appel d'une fonction. Parfois, ce sera un ensemble d'affectations. Le systeme d'animation d'algorithme devra permettre de denir les operations interessantes. Il faut ensuite representer l'enchaînement des operations. La methode la plus simple est de construire une liste d'operations au cours de l'execution de l'algorithme. On obtient ainsi une *trace*, qui fournit deja de precieuses information : les traces sont encore la technique de mise au point favorite de bien des programmeurs. Une autre technique consiste a utiliser des informations supplementaires pour representer ces operations. Ainsi, dans un tri a bulle, les seules operations sont les echanges des valeurs situees aux positions n et $n+1$; on pourra representer l'execution du tri par un graphe representant le temps en abscisse, et la position de l'echange en ordonnee.

La seconde caracteristique d'un algorithme est la nature des donnees qu'il manipule. L'algorithme est construit autour de ces donnees, et il faut pouvoir les representer d'une maniere adequate. Ainsi, on imagine mal de visualiser un tri a bulle sans representer un tableau, ou un QuickSort sans un arbre binaire. Ne prejugeons pas ici de la nature de la representation. Selon les cas, on preferera une representation textuelle (plus precise) ou graphique (plus lisible). Par ailleurs, les donnees manipulees evoluent au cours de l'execution, et il faut montrer cette evolution. Pour comprendre un tri a bulle, il est utile de voir comment les valeurs les plus elevees migrent les unes apres les autres vers le haut du tableau. Pour cela, on peut d'abord mettre a jour la representation des donnees des qu'elles sont modiees. Dans certains cas, cette technique est sufsante : si l'algorithme compte les occurences d'un mot dans un texte, et qu'on est interesse par l'evolution de ce nombre, il sufra de representer cette variation. Mais le plus souvent, cette mise a jour n'est pas sufsante. En effet, si les donnees evoluent, c'est que des operations leur sont appliquees (oublions ici les erreurs de programmation). Or les operations ont une signication en elles-mêmes, et pas seulement par leurs consequences. Dans le cas du tri a bulle, il ne suft pas de voir que deux cases contigues ont change de valeur ; il faut aussi comprendre qu'elles ont *echange* leurs valeurs. Visualiser l'execution d'un algorithme passe donc par la visualisation des operations et des donnees, mais aussi des operations *sur* les donnees.

Enn, une derniere categorie d'informations a visualiser n'est pas directement deductible de la notion d'algorithme. Il s'agit des informations qui representent l'etat de son execution et de ses donnees, mais ne sont pas exprimees de maniere explicite. Ainsi, dans le cas du comptage des occurences d'un mot dans un texte, on voudra parfois representer la proportion de texte deja examinee. Pour le tri a bulle, on pourra montrer l'indice le plus eleve des cases bien ordonnees. Ces informations peuvent être obtenues a partir des donnees, mais elles sont le resultat d'une synthese, et sont donc d'une nature differente des donnees.

6.1.2 Aspects techniques

Apres avoir determine quelles informations devaient être visualisees pour animer un algorithme, examinons maintenant les techniques a mettre en uvre pour permettre cette animation.

Extraire les informations

Tout d'abord, nous avons mentionne que les informations a representer etaient les donnees de l'algorithme et ses operations. Avant de les representer, il faut obtenir ces informations. Or la methode la plus simple pour connaître le comportement d'un programme est de l'executer.

Considerons donc que l'algorithme a animer est mis en uvre par un programme sans erreurs. Les operations et l'evolution des donnees sont alors produites par l'execution de ce programme. Reste alors a extraire ces informations. La technique la plus generale consiste a modier le programme en lui ajoutant des instructions speciales. Elle presente l'inconvenient de manquer de souplesse, en particulier dans un environnement ou les programmes sont compiles. Les techniques qui permettent de ne pas modier le programme sont moins generales, et dependent de l'environnement de programmation. Pour un langage interprete, on pourra modier l'interpreteur. Pour un langage compile, on pourra utiliser des techniques traditionnelles de mise au point (ces technique sont detaillees au chapitre suivant), ou modier le programme au moment de sa compilation, soit en modiant le compilateur, soit en utilisant un preprocesseur.

Dessiner et animer

A l'oppose des techniques d'instrumentation de programme que nous venons d'enumerer, l'animation d'algorithmes necessite aussi des capacites graphiques. Tout d'abord, il faut representer les donnees. Pour cela, un systeme graphique a base d'objets est indique, tout comme pour les interfaces graphiques habituelles. Cela permet en effet de considerer la representation comme l'association d'une ou plusieurs donnees avec un ou plusieurs objets graphiques. Les evolutions des donnees sont alors visualisees par la modication des objets graphiques correspondants.

Ensuite, on peut utiliser le même systeme graphique pour donner une representation simple des operations, en associant une representation graphique a la trace du programme. Aussi etonnant que cela puisse paraître, nous rencontrerons a la section suivante des systemes d'animation de programmes qui n'ont pas de capacite d'animation pure, dirigee par le temps. Ces systemes se limitent le plus souvent a la representation des donnees et de leur evolution, en mettant a jour les representations graphiques lors des modications. Cependant, un systeme d'animation est necessaire des que l'on veut visualiser les operations sur les donnees. Ainsi, l'echange de valeurs dans le tri a bulle peut être montre par le deplacement des cases et de leur contenu a l'ecran.

Par ailleurs, il peut aussi être utile de representer de simples evolutions de valeurs par des animations, an d'eviter des sauts trop brutaux et d'attirer l'attention sur le changement en cours. Par exemple, un pourcentage represente par un cadran et passant de 0 a 100 pourra amener l'aiguille a se deplacer progressivement entre le 0 et le 100. Toutefois, il s'agit la d'une possibilite purement cosmetique, qui a pour inconvenient de placer la representation dans des etats non coherents avec les donnees, et doit donc être utilisee avec prudence.

Etablir les liens

Enn, pour animer des algorithmes il faut aussi etablir des liens entre d'un côte les representations et les animations, et de l'autre les donnees et les operations. Nous retrouvons la un probleme similaire a celui des communications entre interface et noyau fonctionnel, illustrant ainsi l'interêt de l'animation d'algorithme comme application pour des outils de construction d'interface. Cette association entre les objets du systeme d'animation et ceux extraits de l'algorithme depend beaucoup des formalismes utilises de part et d'autre. Nous rencontrerons plusieurs types d'associations plus ou moins souples dans les systemes etudies a la section suivante. On peut toutefois emettre quelques remarques d'ordre general.

Tout d'abord, l'association se fait souvent sous deux formes. D'une part, on associe objets graphiques et donnees, et d'autre part scenes d'animation et operations. Mais cette separation n'est pas toujours adaptee. En effet, nous avons mentionne precedemment l'utilite de representer des operations par des objets, an de conserver une trace de l'execution. De plus, si le systeme d'animation le permet, on peut aussi imaginer de representer une donnee par un objet anime : un booleen par un clignotant, par exemple. Un mecanisme d'association sufsamment general permettra donc tous les types d'associations entre objets de l'interface et informations de l'algorithme.

Un autre aspect critique de l'association entre le systeme d'animation et l'algorithme est la gestion de la dynamicite. Si l'algorithme cree de nouvelles donnees, ou encore si l'on cherche a representer chaque operation par de nouveaux objets graphiques, il faut que le systeme d'animation soit capable de creer de nouveaux objets graphiques. Il faut aussi que le mecanisme de description des liens permette de decrire comment cette instanciation doit se produire, et puisse etablir les liens entre le nouvel objet et des objets deja presents dans l'interface. Par exemple, si un algorithme ajoute des maillons a une liste chaînee, il faut pouvoir decrire la creation des nouveaux dessins de maillons, et surtout la creation des traits liant les nouveaux dessins aux anciens. Une solution extrême consiste a laisser au systeme d'animation la responsabilite de gerer son propre modele des donnees, an de savoir ou placer le nouveau dessin et comment le reiler aux autres. Cela signie que ce systeme n'a plus seulement des capacites graphiques, et cela complique sa realisation. L'autre extrême consiste a ce que le systeme qui manipule l'algorithme conserve la trace des objets graphiques deja crees an de les utiliser comme parametres pour la creation du nouvel objet. Cependant, conserver de telles references explicites sur des objets graphiques est un handicap pour changer de representation. Ce probleme est particulierement delicat, et ne semble pas concerner seulement le domaine de l'animation d'algorithmes. C'est en effet un obstacle majeur a la separation totale entre interface et noyau fonctionnel, dans la mesure ou le noyau fonctionnel a souvent besoin d'informations supplementaires pour provoquer la creation de nouvelles representations graphiques.

6.1.3 Programmes ou algorithmes ?

Jusqu'a present, nous avons employe indifferemment les termes d'animation de programme et d'animation d'algorithmes, en vertu du fait qu'un programme est la concretisation d'un algorithme. En particulier, un chercheur en algorithmique, s'il veut animer un algorithme, devra en faire un programme avant d'utiliser des outils pour l'animer. Cependant, il existe deux categories de preoccupations, qui permettent de separer animation d'algorithmes et animation de programmes.

Le premier besoin est la representation d'un algorithme dans un but d'enseignement ou de recherche. Dans ce cas, on sait a l'avance ce qu'on veut representer : c'est un algorithme particulier, sujet d'un cours ou d'un travail de recherche. On est alors interesse par la fabrication soigneuse de representations qui mettent en valeur le fonctionnement global de l'algorithme, ou certaines proprietes particulieres. Animer un algorithme est alors similaire a la preparation d'un cours ou la fabrication d'un lm, et on peut accepter d'y passer un certain temps, dans la mesure ou ce travail sera utilise de nombreuses fois. De la m^eme maniere, on acceptera d'ajouter des ordres d'animation a un programme, puisque ce programme aura ete ecrit precisement pour permettre l'animation de l'algorithme. Notons aussi qu'on peut faire l'hypothese qu'un programme mettant en uvre un algorithme sera relativement simple, et exempt d'erreurs, ce qui simplie un peu la t^ache du systeme d'animation. Enn, l'animation d'algorithme fait souvent appel a des representations synthetiques, et ne necessite pas une grande delite vis-a-vis de la structure des donnees.

Le second besoin est la representation d'un programme pour sa mise au point. Dans ce cas, le temps de creation de l'animation est beaucoup plus critique. En effet, le programmeur qui recherche une erreur ne sait pas a l'avance de quelles representations il aura besoin. De plus, il lui arrive de vouloir changer de representation au cours de son investigation, quand il a pu eliminer certaines hypotheses. La facilite d'utilisation est donc primordiale, et on souhaitera m^eme disposer de representations predenies pour les donnees. Pour les m^emes raisons, un programmeur ne desire pas modier son programme pour en permettre la visualisation. D'une part, c'est une perte de temps importante, et d'autre part, cela diminue sa conance envers les investigations qu'il va mener : il n'examine plus son programme, mais un programme hybride ou des erreurs peuvent avoir ete ajoutees, et d'autres masquees. En effet, les programmes que l'on cherche a animer sont le plus souvent errones. Ils contiennent l'erreur que l'on recherche, et d'autres que l'on ne soupconne pas. De plus, ce sont souvent des programmes de grande taille, avec des donnees complexes, dont on ne cherche a visualiser qu'une petite partie. Enn, la delite vis-a-vis des donnees est cruciale, et il faut pouvoir representer completement des structures de donnees complexes, pour permettre une investigation efcace.

L'animation d'algorithmes et l'animation de programmes representent donc des

besoins legerement differents. Dans la section suivante, nous allons examiner plusieurs systemes, dont les plus developpes sont plutôt des systemes d'animation d'algorithmes : Balsa, Tango, et Pastis. Cependant, l'animation de programmes, et en particulier une animation centree sur les donnees, est un enjeu important [Beaudouin-Lafon & Karsenty 87].

6.2 Les systemes existants

6.2.1 Balsa

Balsa est le systeme d'animation d'algorithmes le plus connu et le plus acheve a ce jour. Il a ete realise a l'Universite Brown par Marc Brown et Robert Sedgewick [Brown & Sedgewick 84 ; Brown 88], et il y est utilise pour l'enseignement d'algorithmique dans une \salle de classe electronique". Dans Balsa, l'accent est principalement mis sur la qualite des images obtenues, ainsi que sur leur pouvoir explicatif.

Balsa manipule des algorithmes transcrits dans un langage de programmation structure tel que Pascal. La correspondance entre les operations d'un algorithme et les instructions d'un programme n'est pas toujours claire : il n'est pas facile de voir un echange de valeurs dans les trois instructions *tmp=i; i=j; j=tmp*. Pour animer un algorithme en fonction de ses operations signicatives, Balsa introduit la notion d'*evenement interessant*, que l'on ajoute au programme. Ainsi, aux trois instructions realisant l'echange sera ajoute l'evenement *Swap (i,j)*. Par ailleurs, de nombreux algorithmes ont besoin de donnees en entree. Pour cela, Balsa utilise des evenements d'entree, que l'on ajoute de la même maniere que les evenements interessants. Le programme ainsi annote est traite par un preprocesseur, qui lui ajoute les differents composants necessaires a la representation. L'animation est ensuite obtenue par compilation, puis execution du programme resultant. Ce dernier a l'architecture representee a la gure 6.1 : les evenements interessants sont distribues a des *vues*, qui sont des composants logiciels capables de produire une representation graphique de l'algorithme. Plusieurs vues differentes peuvent être utilisees en même temps. Par ailleurs, les evenements d'entree sont traites par un *generateur d'entrees*, qui fournit des donnees enregistrees, ou obtenues de maniere interactive. Dans ce dernier cas, l'obtention des donnees se fait par echange de messages entre le generateur d'entrees et une vue.

Chaque vue est obtenue par un ensemble de procedures qui gerent l'animation, mais aussi traduisent les evenements en provenance de l'algorithme. En effet, Balsa essaie de permettre la reutilisation des composants graphiques, dont la realisation est complexe. Une vue est donc composee de trois parties : un adaptateur, un module de modelisation, et un module de rendu. Les adaptateurs comme les modules de modelisation peuvent être partages par plusieurs vues (gure 6.2). Le module de modelisation gere un modele

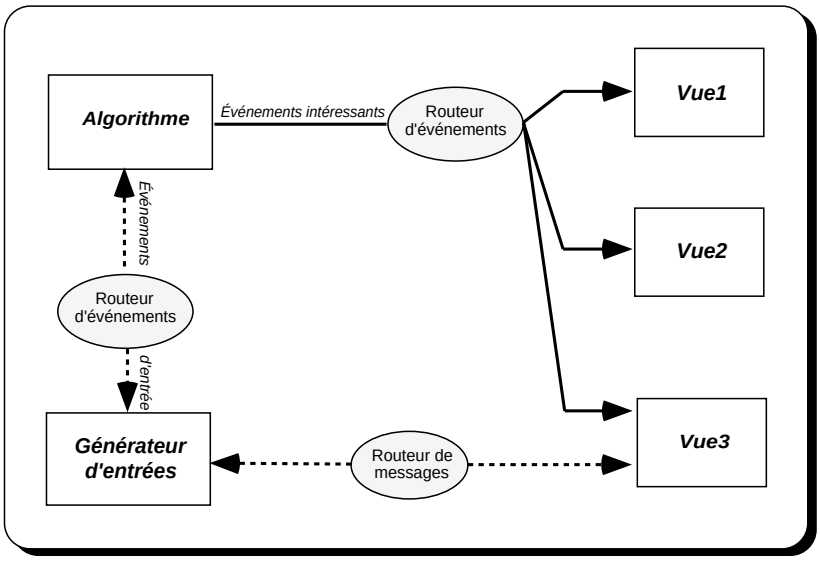

Figure 6.1 : L'architecture de Balsa.

interne qu'il met a jour en fonction des messages qu'il recoit. Ce modele est represente graphiquement par un ou plusieurs modules de rendu. Le rôle de l'adaptateur est de traduire les evenements interessants en ordres comprehensibles par le module de modelisation. De cette maniere, un ensemble compose de modules de modelisation et de rendu peut être utilise pour plusieurs algorithmes, en changeant simplement d'adaptateur.

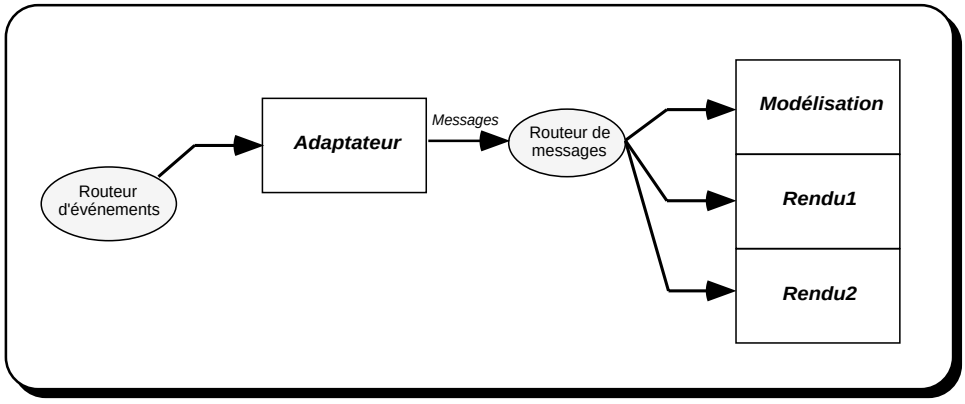

Figure 6.2 : Deux vues sur un même algorithme qui partagent un adaptateur et un module de modelisation.

6.2.2 *Tango*

Tango est un systeme d'animation d'algorithmes developpe par John Stasko a l'Universite Brown [Stasko 89]. Parmi les systemes d'animation d'algorithmes, Tango est le seul a ce jour dont la partie \animation" ait fait l'objet d'une formalisation. Nous avons detaille cette partie au chapitre 4. Nous allons maintenant nous interesser a la maniere dont Tango utilise l'animation pour representer des algorithmes.

Tango est soigneusement decompose en trois parties : l'animation, la manipulation des algorithmes, et la gestion des associations entre les deux. Pour Tango, un algorithme est modelise par des donnees et des operations. C'est avec les operations que l'on rend compte de l'execution et de l'etat de l'algorithme. L'animation de l'algorithme se fait en associant des scenes d'animation a ces operations.

```
int n, i, b, wtnum;
real wt;
real bin[100];

output("How many bins?");
input(n);                                          ⇐ Init(bin, n);

for (i=0; i<n; ++i)
    bin[i] = 0.0;

wtnum = 0;
output("Enter the weights (0.0 to quit)");
do {
    input(wt);
    if (wt == 0.0) break;
    b = 0;                                         ⇐ NewWeight(&wt, wtnum, wt);
    while (bin[b] + wt > 1.0) {
        b++;                                       ⇐ Failure(&wt, wtnum, bin, b);
    }
    bin[b] += wt;                                  ⇐ Success(&wt, wtnum, bin, b);

    wtnum++;
} while (1==1)
```

Figure 6.3 : Un programme annote decrivant un algorithme simple de remplissage de sacs a dos. Les quatre operations de l'algorithme sont identiees sur la droite du programme.

Pour Tango, un algorithme est concretise par un programme, ecrit en C par exemple. Les operations de l'algorithme correspondent a des evenements interessants rajoutes au programme, comme dans Balsa. L'implementation de Tango utilise l'environnement

de programmation FIELD developpe a Brown [Reiss 88]. Cela permet d'utiliser deux methodes d'annotations : soit les evenements sont simplement ajoutes sous forme d'instructions dans le programme, soit ils sont ajoutes sous forme de commentaires, et c'est FIELD qui se charge de les emettre en parallele avec l'execution. Les evenements interessants sont representes comme des appels de fonctions. Le nom de la fonction represente le type de l'operation, et les parametres servent a transmettre des donnees du programme. La gure 6.3, reprise de [Stasko 89] montre un programme annote representant un algorithme simple.

L'association entre un algorithme et une animation se fait de trois manieres. Tout d'abord, on peut associer des donnees de l'algorithme avec des objets du systeme d'animation : emplacements, images, chemins et transitions. En general, des valeurs signicatives sont associees a des images, et des emplacement en memoire (pointeurs ou indices dans un tableau) a des emplacements geometriques. Il arrive que certaines variables soient associees a des transitions, permettant ainsi de choisir un type de mouvement en fonction de l'etat de l'algorithme. Ensuite, il est possible d'associer des objets a des operations de l'algorithme. De cette maniere, on peut par exemple representer le nombre de permutations dans un tri a bulle, en afchant une image par operation. Enn, l'association la plus complexe est entre les operations d'un algorithme et les scenes d'animation. C'est cette association qui permet de visualiser la permutation de deux valeurs lors du tri d'un tableau.

Les trois types d'associations que nous venons de mentionner utilisent un mecanisme de tables d'association. Tango permet la creation de tables associant un objet du systeme d'animation a un nombre donne de cles. Les cles sont d'autres objets du systeme d'animation ou des donnees de l'algorithme. Par exemple, si l'on veut representer un tableau d'entiers, il sera utile d'associer une position geometrique a chacun des indices du tableau. On pourra le faire de la maniere suivante :

AssocStore (\ID", *array*, *i*, *pos_to_store [i]*);

On pourra par la suite retrouver la position par l'appel :

retrieved_pos = AssocRetrieve (\ID", *array*, *j*);

Dans cet exemple, la table *ID* est une table predenie de Tango. La gure 6.4 montre comment de telles associations peuvent être utilisees pour realiser une scene d'animation generique. La scene d'animation obtenue est illustree par la gure 6.5.

Ce mecanisme de tables d'association est sufsant pour les deux premiers types d'associations : entre donnees et objets, et entre operations et objets. La gure 6.4 illustre leur utilisation pour une scene de l'algorithme de sacs a dos. En revanche, les associations entre operations et scenes font intervenir un mecanisme plus complexe. En effet, Stasko insiste sur le fait qu'il n'existe pas toujours de relation simple entre une operation de l'algorithme et une scene d'animation. On peut vouloir faire correspondre a une operation la succession de plusieurs scenes d'animation, surtout quand les scenes ont ete denies a l'avance et qu'on veut les reutiliser. De maniere symetrique, il peut

```
void
AnimMoveTo(weight_id, bin_id)
   int weight_id, bin_id;
{
   Location   from, to;
   Image      rect, text;
   Path       base_path, exact_path;
   Transition move_rect, move_text, compose;

   rect = AssocRetrieve("ID", weight_id);
   text = AssocRetrieve("TEXT", rect);
   from = ImageLoc(rect, SouthWest);
   to = AssocRetrieve("ID", bin_id);
   base_path = PathMakeType(Clockwise);
   exact_path = PathExample(from, to, base_path);
   move_rect = TransCreate(Move, rect, exact_path);
   move_text = TransCreate(Move, text, exact_path);
   compose = TransCompose(move_rect, move_text);
   TransPerform(compose);
}
```

Figure 6.4 : La scene d'animation qui deplace un poids dans un sac a dos. On retrouve dans les tables d'association le rectangle et le texte associes au poids a deplacer. On retrouve ensuite la position geometrique de la destination dans le tableau. Il reste alors a fabriquer et executer une transition qui deplace le rectangle et le texte vers leur destination.

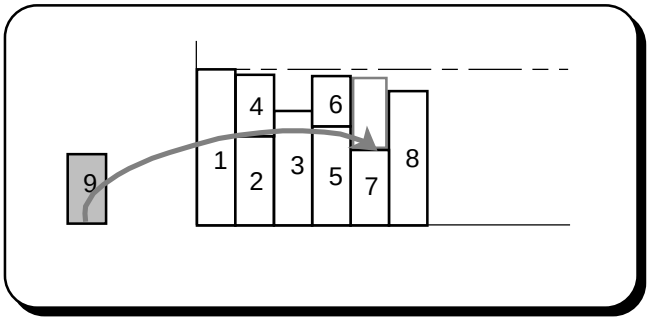

Figure 6.5 : L'execution de la scene d'animation : le rectangle et le texte viennent se mettre en place.

être utile de synthetiser une suite d'operations en une seule scene d'animation. Pour cela, Tango utilise une methode provenant de la theorie des langages. Il permet la construction d'un *automate de traduction,* qui permet de traduire une sequence de mots d'un langage (ici des operations) en une sequence de mots d'un autre langage (des scenes). Dans le cas de l'exemple que nous avons pris, trois des operations sont traduites chacune en une scene, et la quatrieme operation donne lieu a deux scenes enchaînees (gure 6.6).

```
Init -> AnimInit
NewWeight -> AnimNewWeight
Failure -> AnimMoveTo
Success -> AnimMoveTo AnimInPlace
```

Figure 6.6 : Les associations entre operations de l'algorithme et scenes d'animation pour l'algorithme de remplissage de sacs a dos.

6.2.3 Pastis

Pastis est un systeme d'animation d'algorithmes realise a l'Universite de Fribourg [Muller et al. 90a ; Muller et al. 90b]. Pastis permet d'animer un programme sans le modier, et utilise une architecture distribuee. A chaque programme peuvent être associes plusieurs processus realisant chacun une animation differente. Pour le lien entre les donnees du programme et les parametres des animations, Pastis utilise un modele original inspire du modele relationnel des bases de donnees.

Dans Pastis, l'interaction avec le programme est realisee a travers GDB, l'outil de mise au point de programmes du projet GNU [Stallman 89]. De cette maniere, il est possible d'interagir avec un programme executable sans modier le programme source, ce qui donne une plus grande souplesse d'utilisation. Pastis utilise les points d'arrêt de GDB, qui permettent d'interrompre l'execution du programme a l'entree d'une fonction ou simplement a un point donne d'un chier source. Avec GDB, il est possible d'associer des commandes a ces points d'arrêt. En general, on utilise cette possibilite pour afcher la valeur de certaines variables, puis reprendre l'execution. Pastis etend cette possibilite par des commandes de communication avec les processus d'animation. Ces commandes sont denies dans un *script de visualisation.*

Les animations sont decrites dans des scripts d'animation. Ce sont des programmes graphiques qui recoivent de GDB les ordres provenant des scripts de visualisation. Les scripts d'animations sont entreposes sous la forme d'une bibliotheque. On dispose ainsi d'un choix de representations, et il suft d'instancier ces representations dans les scripts de visualisation, puis d'y modier leurs parametres. En pratique, cela correspond a lancer de nouveaux processus d'animation et leur envoyer des ordres. Il est possible

d'ajouter de nouvelles representations, et Pastis comprend un outil pour fabriquer de nouveaux scripts d'animation. Cependant aucune publication a ce jour ne decrit en detail la maniere dont sont construits les scripts d'animation. Les animations en cours d'execution sont coordonnees entre elles par un serveur d'animation. Ce dernier sert d'intermediaire dans les communications avec GDB. Grâce a ce serveur d'animation, il est possible d'avoir plusieurs animations en même temps pour un même programme, voire d'animer plusieurs programmes a la fois. La gure 6.7 resume l'architecture de Pastis.

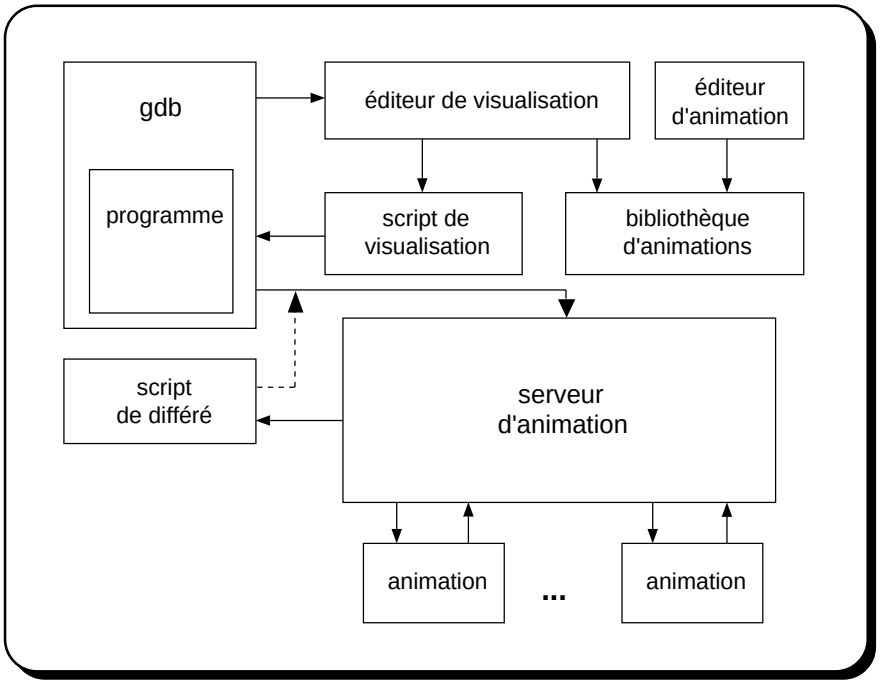

Figure 6.7 : L'architecture de Pastis, vue par ses auteurs.

Les scripts de visualisation sont la partie la plus originale de Pastis. Ils utilisent un modele relationnel pour decrire l'association entre les parametres de l'animation et les donnees du programme. Ces scripts sont composes de deux parties : la declaration des animations utilisees, suivie d'un ensemble de points d'arrêt et de leurs commandes, ou les parametres de ces animations sont modies. Il existe trois types d'animations : des *tuples*, des *relations*, et des *reseaux*. Les tuples sont composes d'un nombre ni de valeurs de types simples : entiers, nombres reels, ou chaînes de caracteres. Par exemple, une horloge dont on peut commander les deux aiguilles permettra de representer un tuple *(entier, entier)*. Une relation est un ensemble de tuples du même format, appeles des elements. Par exemple, une relation *(reel, reel)* pourra être representee par un ensemble de points a l'ecran. Enn, un reseau denit des connexions entre des relations. Les

elements des relations sont alors vus comme des sommets d'un graphe, entre lesquels on peut creer des arcs ou des aretes (les arcs sont orientes, et pas les aretes). Ici, les arcs et les aretes peuvent avoir plus de deux sommets. La gure 6.8, reprise de [Muller et al. 90a], contient la declaration d'un reseau represente par un arbre binaire. Dans la suite du script de visualisation, les animations precedemment declarees sont

```
type binary-tree = network
        node: vertex
            address: integer;
            information: string;
            leftson, rightson: integer;
            end;
        node>leftson, node>address: arc;
        node>rightson, node>address: arc;
        end;
```

Figure 6.8 : Un reseau declarant un arbre binaire.

utilisees par des commandes associees a des point d'arret dans le programme. Ces commandes, qui provoquent l'envoi de donnees vers les processus d'animation, sont des ordres de modication, d'insertion ou de destruction d'elements. Pour les tuples, seuls les modications sont possibles. Ainsi, l'ordre

update modify clock (long-hand=10) (short-hand=30)

provoquera la modication des deux elements du tuple represente par une horloge, et donc fera bouger les deux aiguilles. En general, les membres de droite des affectations sont des donnees extraites du programme en cours d'execution. La gure 6.9, reprise de [Muller et al. 90a], montre un script de visualisation complet.

6.2.4 *Autres systemes*

Incense

Incense [Myers 83] a ete developpe par Brad Myers a Xerox PARC a la n des annees 1970. Incense est un systeme de visualisation de structures de donnees pour le langage Mesa. Il n'a malheureusement jamais ete integre dans l'outil standard de mise au point de Xerox, en particulier pour des raisons de performance.

Incense n'est pas vraiment un systeme d'animation de programmes ou de donnees. Il afche des donnees de maniere graphique, mais seulement a la demande de l'utilisateur, un peu comme un outil de mise au point traditionnel le fait de maniere textuelle. Cependant, Incense merite d'etre mentionne pour les techniques d'afchage qu'il utilise. A chaque type de donnees, il associe un ou plusieurs *formats*, qui

```
# declaration d'une animation
type curve = tuple x,y,z: real; end;

# instanciation de l'animation
open curve-example curve displaying

# le point d'arrêt numero 1, a la ligne 34 du programme
break 34
command 1
silent
update modify curve-example (x=a) (y=b)
cont
end

# le point d'arrêt numero 2
break 53
command 2
silent
update modify curve-example (z=c)
cont
end
```

Figure 6.9 : Un script de visualisation utilisant une courbe en trois dimensions pour representer les variables *a*, *b*, et *c* extraites du programme. L'interface du script d'animation est un tuple, ce qui permet seulement de lui ajouter des points, sans pouvoir les referencer par la suite.

determinent la maniere dont les donnees de ce type peuvent être afchees. Dans sa derniere version, toutefois, Incense ne fournissait qu'un format par defaut, construit de maniere automatique. Un format contient un *artiste* et une *disposition.* C'est l'artiste qui a la responsabilite de l'afchage d'une donnee, en fonction d'une zone rectangulaire de l'ecran qui lui est allouee. Il realise cet afchage a partir de la valeur de la donnee si c'est une donnee simple. S'il s'agit d'un tableau ou d'une structure, il appelle recursivement les artistes associes aux types de ses composantes. Pour cela, il utilise la disposition, qui determine quel espace rectangulaire sera alloue a chaque composante. Avec cette technique, on est assure qu'une donnee n'excedera pas l'espace qui lui est xe. Bien entendu, cela limite le niveau de complexite des structures qui peuvent être afchees, mais il suft alors d'effectuer un zoom pour obtenir plus de details.

GDBX

GDBX est une couche graphique ajoutee au-dessus de DBX, l'outil de mise au point tres repandu sur les machines UNIX [Baskerville 85]. GDBX a ete developpe a Berkeley par David Baskerville, pour des stations Sun. Il permet l'afchage et l'animation de variables au cours de leur evolution. Les donnees modiees clignotent au moment de leur modication. Par ailleurs, il est possible de modier en partie la valeur des donnees. En particulier, la valeur d'un pointeur peut être changee en deplacant l'extremite de la eche qui le represente. En revanche, GDBX ne permettait pas la creation de representations par l'utilisateur. De plus, les representations offertes sont relativement primitives : principalement un ensemble de botes contenant des valeurs textuelles. Les structures imbriquees les unes dans les autres sont afchees avec la même taille. En consequence, les donnees occupent rapidement la totalite de l'espace, et le delement devient necessaire.

Provide

Provide est un systeme concu par Thomas Moher a l'Universite d'Illinois. Il permet de visualiser a posteriori l'evolution des donnees dans un programme, a partir d'une trace produite pendant l'execution [Moher 88]. L'utilisateur dispose d'un ensemble de representations predenies, auxquelles il peut connecter les donnees du programme. Ces representations comprennent des \camemberts", des courbes (x,y), ou des tableaux. Leur aspect est en principe reactualise apres chaque instruction, mais l'utilisateur peut regrouper des instructions, de maniere a eviter l'afchage d'etats incoherents. Par ailleurs, Provide permet de revenir en arriere au cours de l'animation, et même de revenir automatiquement au dernier point ou une variable a ete modiee, ce qui est utile pour la mise au point de programmes. Enn, Provide permet la modication des donnees par manipulation directe. Cependant, cette capacite est liee au fait que seules des representations predenies sont possibles. Par ailleurs, seules les donnees sont

representees, mais pas les operations : Provide ne contient pas de systeme d'animation temporelle permettant de le faire.

Garden

Garden est un environnement de programmation developpe par Steven Reiss a l'Universite Brown [Reiss 87b ; Reiss 86]. Son but est de permettre ce que Reiss nomme la \programmation conceptuelle", c'est-a-dire la possibilite de creer et mettre au point des programmes sur la base des representations mentales des programmeurs, et non d'un simple langage de programmation. Un tel objectif passe a la fois par la representation animee de programmes, mais aussi par la programmation visuelle, c'est-a-dire la construction graphique de programmes. Pour cela, Garden possede un systeme destine a la construction de representations de donnees, Gelo [Reiss 87a].

Gelo utilise une structure hierarchique pour representer des donnees. Des representations de base sont fournies pour les objets simples : un texte encadre d'un rectangle, un cercle, ou une autre forme. Ces objets graphiques de base peuvent ensuite être disposes dans des objets composites, qui permettent soit le pavage d'une zone rectangulaire, soit une disposition sous forme de graphe. L'utilisateur peut utiliser l'editeur graphique Apple pour denir une representation a partir de ces objets, puis indiquer l'association entre les champs d'un type de donnees et les objets graphiques de base qu'il a assembles. Garden est ainsi l'un des rares systemes a permettre la construction de representations. Cependant, Reiss reconnaît que les deux formes de composition d'images qu'il utilise limitent les possibilites de Garden a certains types de donnees.

6.3 L'animation de donnees typees

Les systemes que nous avons examines a la section precedente ont des approches differentes de l'animation de programmes. Certains cherchent a offrir une vue synthetique d'un algorithme, et permettent d'associer des operations a des scenes d'animation, en utilisant les donnees du programme comme parametres. D'autres sont plus centres sur l'evolution des donnees, mais ne permettent pas de representer les operations sur ces dernieres autrement que par les variations que ces operations provoquent. Ainsi, l'empilement d'un entier sur une pile sera visible parce que la pile aura change de taille, mais l'operation d'empilement ne sera pas explicite. Aucun des systemes rencontres ne permet de representer un programme comme une collection de donnees, sur lesquelles des operations sont effectuees. Par ailleurs, la plupart de ces systemes ne fournissent que des representations predenies, ou imposent une phase de programmation complexe pour creer de nouvelles representations ou de nouvelles animations.

Nous allons maintenant etudier les problemes souleves par la representation graphique et l'animation de variables d'un programme en fonction de leur type. Nous donnons au mot \type" le sens que lui donnent certains langages evolues comme ADA, CLU, Eiffel ou C++. Dans ces langages, on construit un systeme de types a partir de types de base, et de constructeurs de types. La denition d'un type comprend des donnees, qui sont en general cachees a l'utilisateur, et des fonctions (ou methodes) qui permettent de manipuler les objets de ce type. Le probleme est donc de fournir une representation pour les objets d'un type, pour leur evolution, et pour les operations qu'on leur applique. Le but recherche est que l'auteur d'un programme puisse construire de telles representations de maniere interactive, an de les utiliser pour la mise au point du programme. Il ne s'agit pas ici de denir les representations les plus adaptees aux differents types de donnees : nous entrerions alors dans le domaine de la semiologie graphique [Bertin 73], qui est complexe et eloigne de notre propos. Il s'agit plutôt d'envisager les diverses representations possibles, an de mettre en evidence les problemes techniques qu'elles soulevent.

6.3.1 Associer une representation a un type

La base de toute visualisation animee d'objets est la representation graphique que l'on associe a chaque type d'objets. Les evolutions et operations sur un objet seront ensuite visualisees par des animations centrees autour de sa representation graphique. Cette derniere fournit d'ailleurs la methode la plus simple pour montrer l'evolution d'un objet : il suft pour cela de la mettre a jour quand l'objet varie. Nous verrons cependant par la suite qu'il est parfois souhaitable de representer les evolutions de maniere plus explicite.

Les langages de programmation types distinguent types de base et types construits. Par exemple, les types \pointeur sur *pile*
" ou \tableau de *pile*" sont construits a partir du type *pile*. Les constructeurs habituels sont le pointeur, le tableau, la fonction (qui a par ailleurs un rôle particulier), et la structure, ou enregistrement. Ce dernier constructeur de types permet d'obtenir des types complexes fabriques en agglomerant des donnees de types plus simples (des *champs*), et en leur donnant des noms. Parfois, la representation graphique de ces structures depend fortement des types de base qui la composent. Une structure constituee d'un entier et d'une chaîne de caracteres sera ainsi representee par la juxtaposition de ces deux elements. C'est ce que permettent des systemes comme Garden ou Incense. Cependant, la generalisation de la programmation modulaire et de la programmation par objets a donne un rôle de plus en plus important a ces structures, tout en introduisant une distinction entre l'abstraction que represente une structure et la maniere dont elle la met en uvre.

Parfois, comme dans les langages evolues, on voudra oublier l'implementation

d'un type pour ne montrer que l'abstraction qu'il est cense representer. Par exemple, imaginons un systeme modelisant l'activite des clients dans un supermarche. On y trouvera un type *Client*, compose de diverses donnees representant l'âge du client, son sexe, ou le contenu de son chariot. On y trouvera aussi un type *File* pour representer les les d'attentes aux caisses, qui sera vraisemblablement compose d'un tableau de *Clients,* et d'un entier qui indiquera la longueur de la le. Supposons maintenant

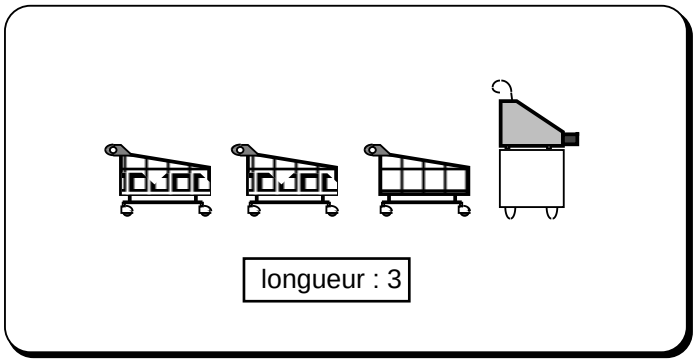

Figure 6.10 : Une le d'attente representee de maniere adaptee au contexte.

que l'on veuille visualiser l'evolution de la longueur d'une le. On utilisera alors une representation graphique telle que celle de la gure 6.10. Notons que des composantes du type *Client* disparaissent dans la representation : le sexe, ou l'âge des clients. Par ailleurs, des objets nouveaux apparaissent, comme la caisse enregistreuse.

Types de base et structures

Dans les langages actuels, les types de base sont des types scalaires : entiers, nombre ottants ou caracteres. On connaît un certain nombre de representations pour ces types. Elles comprennent entre autres la forme textuelle, le cadran, et le thermometre. Cependant, a part pour des applications qui mettent l'accent sur le calcul numerique ou la manipulation de texte, ces types sont rarement utilises en tant que tels. Ils sont souvent presents sous la forme de champs de structures. Pour cette raison, on peut les assimiler a des structures, un type de base etant une structure a un seul champ.

Nous l'avons mentionne plus haut, la decomposition d'une structure en champs est une chose, et l'abstraction qu'elle represente en est une autre. De plus, on ne souhaite pas toujours visualiser le même aspect de l'abstraction. On voudra donc creer des representations graphiques rendant compte des abstractions et de leurs differents aspects, comme nous l'avons fait pour la le d'attente de la gure 6.10. Cependant, la seule information concrete dont on dispose est la description en champs. Le principal

probleme pour representer une structure est donc la liaison entre les champs et les caracteristiques graphiques de la representation.

Considerons une structure composee de deux nombres, et representant un point dans le plan. La technique la plus simple consiste a associer une representation a chacune des deux coordonnees, et a juxtaposer ces deux representations. Cependant, ce n'est pas une maniere tres intuitive de visualiser un point. On preferera en general le representer par un point a l'ecran. Il faut alors etablir le lien entre les deux coordonnees de notre point abstrait et celles de sa representation graphique.

La situation se complique encore si l'on n'est pas interesse par le point lui même, mais par sa distance a un point de reference. Dans ce dernier cas, il faut effectuer un calcul a partir des coordonnees, et utiliser le resultat dans la representation graphique. On dispose pour cela de deux techniques. On peut d'abord se contenter d'extraire les coordonnees, et effectuer les calculs dans le systeme d'animation de programme. L'inconvenient est que cela complique notablement ce dernier, qui doit fournir un veritable langage de programmation. Par ailleurs, les programmes d'interpretation des donnees que l'on construira ne seront pas exempts d'erreurs. La seconde technique consiste a n'autoriser que les traitements qui sont fournis par le type a travers ses methodes, et a demander au programme que l'on anime de faire le calcul. Cela suppose d'abord que l'on ait les moyens de le lui demander, ce qui necessite parfois des prouesses techniques. Ensuite, l'inconvenient principal est que les methodes que l'on utilise ne doivent pas modier l'objet que l'on examine : ce sont ce que les specialistes du genie logiciel nomment des *observateurs*. Or rares sont les langages de programmation qui permettent de determiner si une methode modie ou non l'objet auquel elle est appliquee.

Les pointeurs

Les pointeurs sont des variables qui permettent de referencer des donnees sans les recopier. La maniere la plus intuitive de representer un pointeur sur un objet est une eche dirigee vers cet objet. Cette representation pose cependant plusieurs problemes. Tout d'abord, lorsqu'on represente les donnees d'une application ayant une forte structure de graphe, comme une application interactive, le nombre des pointeurs et leur complexite rendent vite la representation illisible. Il existe des algorithmes pour representer de tels graphes de maniere lisible, mais ces algorithmes sont coûteux, et donc incompatibles avec l'animation. Par ailleurs, on souhaite parfois regrouper les objets du programme en blocs syntaxiques representes par des fenêtres differentes. Or un pointeur est souvent dans un autre bloc que l'objet sur lequel il pointe. Et il est sinon impossible, du moins peu souhaitable de dessiner des eches d'une fenêtre a l'autre. Il faut donc pouvoir adopter des representations plus condensees que la eche. Plusieurs solutions sont possibles. Par exemple, on peut simplement indiquer si le

pointeur est valide, nul, ou invalide. L'objet pointe est alors mis en evidence seulement sur requête de l'utilisateur. On peut aussi representer seulement les extremites de la eche, la liaison etant suggeree par des etiquettes ou par un code de couleurs.

Par ailleurs, on ne souhaite pas toujours faire apparaître les pointeurs dans les representations. Il est parfois utile de visualiser un pointeur en tant que tel, en particulier pour la mise au point de programmes ; mais on est souvent plus interesse par les donnees referencees que par le pointeur lui-même. Dans l'exemple de la le d'attente, la le ne contient pas des pointeurs, mais bien des clients. Pour cela, il faut pouvoir automatiquement \dereferencer" un pointeur, et visualiser les donnees sur lesquelles il pointe. Ce dernier type de representation met en valeur un probleme de la representation de pointeurs : si on change la representation de l'objet pointe, celle du pointeur peut aussi changer. C'est vrai aussi pour les representations a base de eches : si on deplace l'objet, la eche doit suivre. Cela signie que tous les objets sur lesquels on represente des pointeurs doivent garder la trace de ces pointeurs, an de leur notier les modications eventuelles.

Les conteneurs

Les *conteneurs*" sont les objets qui permettent de stocker plusieurs objets appeles elements, et qui appartiennent le plus souvent a un même type. Rentrent dans cette categorie les tableaux, les listes, les arbres, les les d'attente, ou les piles. Les langages compiles traditionnels ne proposent que les tableaux comme types construits predenis. Il est cependant possible de simuler l'existence des autres conteneurs par des structures et des fonctions associees. Des langages comme C++ ou ADA permettent de rendre cette simulation tres dele grâce a leurs mecanismes d'encapsulation et de genericite.

Chaque type de conteneur a sa forme privilegiee : les tableaux sont le plus souvent vus comme des alignements de rectangles, et les listes chaînees comme des elements relies par des eches. Mais la situation n'est pas aussi simple. Si l'on prend l'exemple des tableaux, leur representation ne peut pas se limiter a une simple juxtaposition de leurs elements. En effet, il est rare que l'utilisateur souhaite voir au même moment les details de tous les elements. Tout d'abord, la taille est un facteur determinant : il est peu commode de representer dans sa totalite un tableau de 1000 entiers. Il faut donc permettre de ne montrer qu'une partie d'un tableau, le reste etant accessible grâce a une barre de delement. Ensuite, viennent des problemes d'echelle. Si la representation de chaque element a une taille importante, il est difcile d'en montrer beaucoup a la fois. Par ailleurs, le tableau est une abstraction en soi ; sa taille et la maniere dont il est rempli ont parfois plus d'importance que la valeur des elements qui le composent. Pour ces deux raisons, les elements doivent pouvoir être representes d'une maniere simpliee, de taille reduite. On peut utiliser pour cela le mecanisme d'Incense, consistant a fournir

un zone rectangulaire a chaque objet contenu, qui doit alors se dessiner dans cette zone. L'inconvenient de cette technique est qu'elle suggere plus des representations obtenues par des fonctions de dessin a qui on passe des parametres, que des representations construites interactivement. On peut imaginer des solutions moins generales, mais plus adaptees a la construction interactive. Par exemple, on peut envisager la construction de representations a certaines tailles cles, comme cela se fait pour les icônes du Finder.

Les iterateurs

Un iterateur est un objet qui permet d'enumerer les valeurs qui se trouvent dans un conteneur. Il possede generalement un operateur qui permet de lui demander l'element suivant. Dans les langages ou ils existent (CLU par exemple), les iterateurs permettent de simplier la plupart des constructions *for* ou *while*. Des langages comme C++ permettent de denir des iterateurs de la même maniere que des conteneurs : grâce a des mecanismes d'encapsulation des donnees. En C ou en Pascal, il n'y pas d'iterateurs, mais les boucles d'iteration existent cependant, et on souhaite parfois representer leur etat d'avancement.

La representation d'un iterateur depend beaucoup de celle des donnees sur lesquelles il itere. On pourra representer une boucle contrôlee par un simple entier en montrant la valeur de cet entier, mais en general les iterateurs sont plutôt utilises pour enumerer les elements d'un conteneur. On peut alors imaginer pour l'iterateur le même type de representation que pour un pointeur, qui permet de mettre en evidence l'element courant.

Par ailleurs, on peut utiliser le fait qu'un iterateur est un type de donnees, qui donc represente un etat, mais peut aussi être percu comme une fonction, qui sert a enumerer. Faire avancer l'iterateur d'un pas est alors equivalent a appliquer une fonction particuliere au conteneur sur lequel on itere.

Les fonctions

Les fonctions sont des types construits tres particuliers. Il ne faut pas confondre le type d'une fonction avec le type des valeurs qu'elle retourne. Ainsi, une fonction qui prend un entier et renvoie un entier n'aura en general pas le même type qu'une fonction qui renvoie un entier, mais ne prend pas d'argument. Le type d'une fonction est utilise lors des manipulations sur cette fonction, en particulier pour la referencer par un pointeur ou dans un tableau. Ces constructions n'existent pas dans tous les langages : Ada par exemple ne les autorise pas. Dans les langages ou elles existent, il est utile de les representer, ce qui se fera surtout avec le nom de la fonction ou eventuellement avec une icône. En effet, le moyen le plus synthetique de representer une fonction est encore de la nommer. On peut imaginer de la representer par une

animation demontrant son comportement, mais c'est une representation bien complexe pour une utilisation assez marginale.

Ce dernier point nous amene a ce qui est le plus interessant dans une fonction : la visualisation de son utilisation. On quitte alors le domaine de la representation des types pour entrer dans celui de l'animation des operations sur ces types.

6.3.2 Associer un comportement a un type

Une fois utilises dans des programmes, les divers objets pour lesquels nous venons d'envisager des representations vont evoluer, être modies, se voir appliquer des operations. On voudra donc visualiser ces evolutions, et en particulier en associant des comportements aux representations des objets. Examinons maintenant quels types d'evolutions sont envisageables, et quels comportements sont necessaires.

Modications

L'evolution la plus simple pour un objet est sa modication, realisee par une affectation : on change la valeur d'un entier, ou d'un champ d'une structure. C'est aussi l'evolution qui permet la representation la plus facile : il suft de detecter les modications, et de remettre a jour la representation de l'objet a chaque fois. C'est ce que permettent des mecanismes comme celui des valeurs actives, que l'on trouve dans certains generateurs d'interfaces. Il suft pour cela de savoir representer les objets, et de pouvoir detecter leurs modications.

La detection des modications pose toutefois des difcultes. En effet, une affectation est une operation tres simple, mais sufsamment primitive pour être indetectable. Il faut donc utiliser des techniques particulieres pour être averti des modications. On peut tout d'abord demander au programmeur d'appeler explicitement une fonction de mise a jour a chaque modication, mais cette solution manque de souplesse. On peut aussi \espionner" regulierement la valeur des variables, mais cela demande que le programme soit execute dans un environnement adequat, et cela pose parfois des problemes d'efcacite. Enn, on peut utiliser les possibilites des langages modernes qui permettent d'associer des operations aux types de donnees, et même parfois d'interdire les modications directes sur les donnees, obligeant le programmeur a utiliser les operations. On peut alors decider de ne verier la valeur des objets qu'apres de telles operations. Plus les donnees seront protegees des acces directs, et plus une telle technique sera efcace. Toutefois, dans de telles circonstances, on souhaite souvent visualiser les operations effectuees, et pas seulement la variation qui en resulte. Nous y reviendrons par la suite.

Un autre probleme lie a la representation des variations est le fait que la simple mise a jour de la representation statique n'est pas toujours sufsante. En effet, cette mise

a jour produit une animation rudimentaire. Lorsqu'elle se produit de maniere isolee, elle resulte en une variation brutale de l'afchage, trop breve pour que l'utilisateur puisse successivement diriger son attention dans la bonne direction et comprendre ce qui s'est produit. Lorsqu'elle se produit de maniere repetitive, elle fournit une animation de mauvaise qualite, composee de variations brutales, et donc difcilement comprehensible. Que l'utilisateur souhaite une animation lente, pour suivre toutes les variations en detail, ou rapide, pour avoir une vue d'ensemble, il ne se contentera pas d'une liaison directe entre etat et representation. Il faut alors pouvoir rendre compte de chaque variation par une animation, ce qui rapproche la visualisation des variations de celle des operations.

Operations

Nous avons vu qu'on pouvait visualiser une operation a travers la variation qu'elle produit sur l'objet associe. Cependant cette technique, si elle est simple, n'est pas satisfaisante. Nous avons vu qu'elle etait deja limitee pour les simples variations de valeurs. Pour les operations, elle a un inconvenient supplementaire : elle elimine de l'information. En effet, appliquer une operation sur un objet est une action particuliere, differente d'une simple modication ou d'une autre operation qui donnerait le même resultat. Si un entier vaut 2, lui affecter 6, le multiplier par 3 ou lui ajouter 4 sont des actions differentes. De plus, certaines operations n'ont pas de resultat visible : multiplier par 3 un entier qui vaut 0, par exemple. Une autre operation qu'on peut vouloir visualiser n'amene jamais de variations : c'est la lecture de la valeur d'un objet. Il est donc important de representer les operations directement, et pas seulement par leurs resultats. Pour cela, il faut utiliser l'animation.

Nous ne nous attarderons pas ici sur les types d'animation adaptes a la representation d'operations. On peut toutefois faire quelques remarques d'ordre general sur ces animations. Tout d'abord, l'animation d'une operation sera la plupart du temps liee a la representation de l'objet sur laquelle l'operation est effectuee. Par exemple, on pourra visualiser la lecture d'un entier par un bref changement de couleur de son image. Cela signie qu'il faut pouvoir se referer a la representation d'un objet lorsqu'on anime une operation. Cela pose des problemes techniques lorsqu'on declenche l'animation : comme pour la creation de nouveaux objets, evoquee a la section 6.1.2, il faut determiner ou ces references sont conservees. Cela pose aussi des problemes si l'on veut construire ces animations interactivement : il faut denir comment l'utilisateur doit designer les parametres de la representation utilises pour l'animation.

Une autre constatation pose des problemes similaires. C'est le fait que de nombreuses operations font intervenir plus d'un objet, parfois de maniere symetrique. On ne peut pas toujours voir cette operation comme faisant intervenir un objet et des parametres, surtout si les autres objets ont eux-aussi une representation graphique.

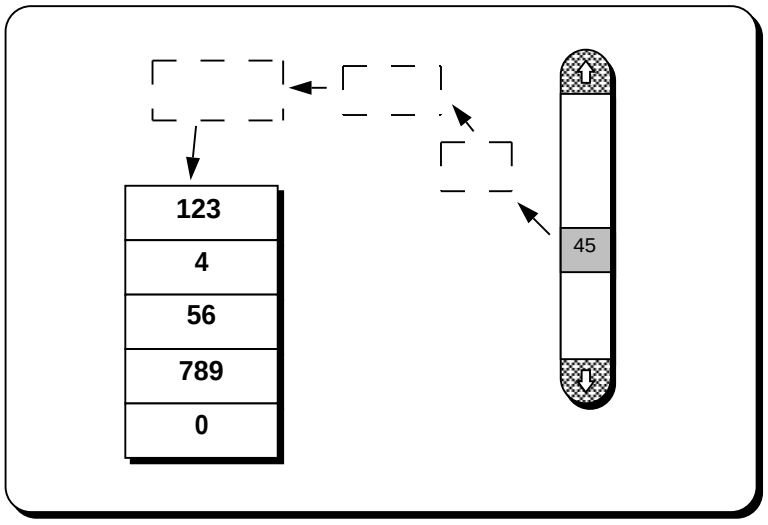

Figure 6.11 : Empiler un entier.

Ainsi, l'empilement d'un entier sur une pile fait intervenir deux objets (gure 6.11). L'addition de deux entiers en fait aussi intervenir deux, de maniere symetrique. Le modele des langages a objets elimine la difculte posee par la symetrie en associant un objet privilegie a chaque operation : le recepteur du message. Ainsi, pour additionner deux entiers a et b, il faut envoyer a a un message lui demandant de renvoyer le resultat de son addition avec b. Ce modele de message est parfois percu comme contraignant dans le cadre d'un langage de programmation. Pour la representation graphique, il est encore moins adapte. Toutefois, on pourra considerer que si le langage de programmation introduit une notion d'objet privilegie, c'est une information utile a representer : si la symetrie est deja rompue, montrons dans quel sens elle l'est.

Revenons a l'exemple de la pile et de l'entier, qui met en valeur un autre point interessant. En effet, on peut voir qu'au cours de l'empilement, un objet graphique transitoire apparaît, le temps pour lui d'aller de la representation de l'entier vers celle de la pile. Cela signie que les animations associees aux operations ne font pas toujours appels aux seules representations des objets impliques. Il faut donc que le systeme de denition d'animations permette de creer de nouveaux objets graphiques, et ne se limite pas a faire evoluer les objets existants.

6.4 Conclusion

Nous avons etudie dans ce chapitre l'application la plus repandue a ce jour de l'animation dans les interfaces : l'animation d'algorithme. Nous y avons identie trois tâches distinctes : l'instrumentation d'un algorithme, la construction de representations

animees, et l'etablissement de liens entre ces deux parties. L'etude de systemes existants nous a permis d'examiner les differentes solutions apportees a ces trois problemes. Nous avons en particulier constate qu'ils etaient rarement abordes sous l'angle des objets que l'on rencontre dans un programme. Nous avons donc examine les problemes techniques poses par la representation animee d'objets et de leurs operations. C'est sur la base de ces reexions que le chapitre suivant va presenter Witness, un systeme de mise au point de programmes utilisant l'animation des donnees, et reposant sur Whizz.

Chapitre **7.**

Witness

Witness est un outil de mise au point graphique de programmes. Il utilise Whizz pour visualiser les variables des programmes et les operations sur ces variables, selon des representations construites par les utilisateurs. En fait, Witness a servi de support au developpement de Whizz. En effet, l'animation de programmes est a peu pres la seule application de l'animation qui ait fait l'objet de travaux systematiques. C'est donc cette application qui a servi de base a la conception initiale de Whizz, et les considerations sur les interfaces animees que nous avons presentees au chapitre 4 en sont la consequence.

Un choix delibere a ete fait a l'origine du developpement de Witness : realiser dans la mesure du possible un outil utilisable dans des conditions reelles. Ce choix se justie par deux raisons. D'une part, une application reellement utile est un meilleur candidat qu'une maquette pour l'evaluation des idees qu'on y met en uvre. Cela permet d'attirer plus facilement des utilisateurs, et ces derniers peuvent alors juger si les innovations leur apportent quelque chose. D'autre part, le domaine des interfaces homme-machine est un domaine ou ce qui apparat comme un detail se revele souvent un probleme difcile a surmonter. Parfois, ces details sont ignores dans la realisation d'une \application-jouet", et apparaissent dans une veritable application. Cela a ete le cas pour Witness, ou de nombreux problemes sont apparus, qui avaient ete ignores dans une maquette precedente [Chatty 88]. Ces problemes nouveaux ont contribue a la conception de Whizz. Witness est donc un veritable outil de mise au point, utilisable sur des stations de travail utilisant UNIX, pour des programmes ecrits en C, C++, ou Fortran, l'animation de programmes n'etant disponible que pour C++.

Ce chapitre est organise comme suit. Nous examinerons dans un premier temps les techniques habituelles de mise au point et les recherches dans ce domaine. Nous presenterons les caracteristiques generales de Witness, pour nous attarder ensuite sur son interface de commande et sur la maniere dont il permet d'animer des donnees. Enn, la derniere section sera consacree a des problemes restant a resoudre dans Witness.

7.1 La mise au point de programmes

La mise au point de programmes est un domaine relativement mal considere. La plus grosse partie des efforts sur le genie logiciel est precisement consacree a rendre la phase de \debugging" sinon inutile, du moins la plus courte possible. Et il est vrai que pendant longtemps, les \debuggers" ont ete d'un niveau d'abstraction incompatible avec celui des autres outils dont disposent les programmeurs : placer des points d'arrêt dans le programme et imprimer le contenu de champs de la memoire ne sont pas des tâches du même niveau que la programmation modulaire ou la programmation par objets. Aujourd'hui les \debuggers symboliques" comme DBX ou GDB permettent d'acceder aux variables et aux fonctions par leur nom, et sont capables de montrer quelle ligne du programme source est executee. Mais il s'agit encore de manipulations a un niveau assez bas.

Pourtant, la recherche d'erreurs dans les programmes existe toujours, et existera encore longtemps. Selon les programmeurs, cette recherche se fait avec un \debugger" ou en inserant des ordres d'impression dans leur programme, pour determiner son ot d'execution et la valeur de certaines variables. C'est souvent une tâche longue et penible. Par ailleurs, la recherche d'erreurs n'est pas la seule tâche de mise au point. Toutes les methodes de developpement de programmes comportent le test des composants logiciels. Ces tests ne sont pas toujours faciles a mettre en uvre sans outil specialise. Enn, observer un programme en cours d'execution est un bon moyen pour comprendre son fonctionnement et l'ameliorer, voire y trouver des erreurs qui ne se sont pas encore manifestees. Pour toutes ces raisons, il semble utile de disposer d'outils de mise au point evolues, qui ne soient plus seulement des \debuggers" tels que nous les connaissons.

Techniques de mise au point

Les recherches sur la mise au point prennent deux directions. D'un côte, certains systemes recherchent automatiquement les erreurs. Nous ne nous interesserons pas a cette branche, dont on trouvera une presentation dans [Ducasse & Emde 89]. De l'autre côte, on trouve tous les systemes qui permettent au programmeur d'examiner lui-même son programme. Il peut alors remonter des symptômes d'une erreur jusqu'a sa cause, en developpant des heuristiques d'investigation [Araki et al. 91]. Nous allons etudier les techniques qui peuvent lui faciliter cette tâche.

Les informations que souhaite obtenir l'auteur d'un programme sont de deux types. Tout d'abord, il veut connaître la valeur de certaines donnees. En effet, les erreurs se manifestent souvent a travers des donnees produites ou visualisees par un programme. Il faut donc examiner les donnees pour tenter de trouver la cause de leur mauvaise valeur. D'autre part, le programmeur cherche a connaître le ot

de contrôle de son programme : quelles instructions sont executees, et dans quel ordre. Cette information peut lui servir a rechercher des defauts dans l'organisation du programme. Par exemple, une fonction peut être appelee a un moment imprevu, a la suite d'une erreur. Savoir qu'elle a ete appelee est un pas important dans la recherche de l'erreur. Les informations sur le ot de contrôle servent aussi de base a des strategies de recherche de l'erreur dans le cas ou une donnee est erronee. On peut en effet tenter de localiser le moment ou elle prend sa valeur erronee. Enn, il arrive que le symptôme de l'erreur soit l'echec du programme, qui interrompt son execution. Il faut alors determiner ou l'execution s'est arrêtee, avant de rechercher les causes du probleme.

La plupart des systemes d'aide a la mise au point reposent sur des mecanismes destines a determiner la valeur des variables et le ot d'execution. Dans des environnements ou le programme est interprete, en LISP par exemple, il suft de changer le moteur qui interprete le programme, an qu'il s'arrête pour chaque expression ou chaque ligne de programme. On peut alors tracer l'execution, et obtenir la valeur des variables grâce au dictionnaire de symboles de l'environnement. Pour des langages compiles, l'observation passe par la manipulation d'un processus en cours d'execution. On commence par y introduire des points d'arrêt, c'est-a-dire des instructions qui provoquent l'interruption. Les donnees sont ensuite obtenues en lisant la memoire du processus interrompu. Le fonctionnement de ces mecanismes dans Witness est detaille dans l'annexe D.

Les \debuggers" offrent des fonctions pour ajouter des points d'arrêt, et lire et ecrire en memoire. Les plus primitifs ne permettent de le faire qu'en termes d'adresse en memoire, tandis que les \debuggers symboliques" permettent de manipuler des numeros de lignes, des noms de fonctions et des noms et types de variables. C'est le cas de DBX et GDB. Certains de ces systemes offrent ponctuellement des services plus sophistiques. Ainsi, DBX peut espionner en permanence le contenu d'une variable, et avertir l'utilisateur quand elle change, au prix d'une grande lenteur d'utilisation. GDB, quant a lui, permet d'associer des commandes a un point d'arrêt, ce qui permet entre autres d'automatiser des tâches d'impression de valeurs.

Des outils plus recents permettent des recherches a un niveau d'abstraction plus eleve. Ainsi, Dalek [Olsson et al. 91] contient un veritable langage de programmation pour exprimer les actions a executer lorsqu'un point d'arrêt est rencontre. Avec ce langage, on peut par exemple imprimer le contenu d'une liste chaînee implementee en C, ce qui est fastidieux avec les outils habituels. On peut aussi utiliser ce langage pour verier que certaines proprietes des donnees sont bien conservees au cours du temps. On pourra par exemple verier regulierement que la somme de deux nombres est bien nulle. Dalek permet ainsi de realiser lors de la mise au point un mecanisme d'invariants similaire a celui qu'offre le langage Eiffel lors de l'ecriture des programmes. Par ailleurs, Dalek permet de structurer l'investigation grâce a

un systeme d'evenements. Le langage de commande offre la possibilite d'emettre un evenement lorsqu'on detecte un fait interessant. L'evenement est alors traite, et propage dans le systeme, declenchant eventuellement d'autres evenements selon un graphe de ots de donnees. Des evenements de haut niveau peuvent être denis comme etant produits par l'occurrence successive d'autres evenements. Dalek permet ainsi de suivre le ot de contrôle a un niveau plus abstrait que les points d'arrêt. On peut par exemple decider qu'on est interesse par l'evenement *f_puis_g*, qui est emis quand les fonctions *f* et *g* sont appelees successivement. On retrouve la l'idee mise en uvre dans les \expressions de chemins[1]" [Bruegge & Hibbard 83], qui permettent de decrire des evenements interessants sous la forme d'expressions regulieres contenant des evenements de base. Grâce a ces mecanismes d'exploration systematique, Dalek montre que la mise au point peut quitter le stade de l'artisanat pour devenir une tâche plus raisonnee.

En complement de ces progres dans l'analyse du ot d'execution, des recherches sont menees sur la maniere de presenter les informations a l'utilisateur, et en particulier sur la mise au point graphique de programmes. Les techniques d'animation de programmes, que nous avons decrites au chapitre precedent et qu'utilisent des systemes comme Provide, Incense ou GDBX, ne permettront sans doute jamais une analyse aussi efcace que des langages comme celui de Dalek. En revanche, elles sont prometteuses pour la premiere phase de l'investigation, lorsqu'il s'agit de localiser grossierement un erreur et de recueillir les premiers indices. Pour ces tâches ou la vitesse est predominante, les representations graphiques permettent de presenter des informations a un plus grand debit. On peut donc imaginer qu'un outil de mise au point ideal contiendrait a la fois un mecanisme de representation graphique pour l'exploration et un langage pour l'analyse precise et systematique.

Dans ce contexte, l'outil de mise au point Witness, developpe par l'auteur et presente dans la suite de ce chapitre, met l'accent sur la presentation graphique des informations. Il utilise pour cela les services de Whizz, et permet la representation animee des donnees d'un programme.

7.2 *Witness : caracteristiques generales*

Witness est un outil de mise au point graphique utilisable pour des programmes executes dans l'environnement UNIX. Il possede deux groupes de fonctions. Tout d'abord, Witness dispose des capacites habituellement rencontrees dans des \debuggers". En particulier, il permet l'examen statique d'un programme, de sa structure, ses types, ses fonctions et ses variables. Il permet aussi de placer des points d'arrêt, et d'examiner l'execution du programme, et cela pour tout programme ecrit en C,

[1] path-expressions

C++ ou Fortran et compile avec l'option habituelle pour la mise au point. L'interface de Witness pour ces fonctions est originale : elle utilise le paradigme des fenêtres et des icônes. De cette maniere, Witness applique le principe de la manipulation directe a l'exploration de programmes, en utilisant la forte organisation hierarchique des programmes structures. L'interface iconique de Witness est mise en uvre grâce au serveur d'icônes WISh.

La seconde partie de Witness concerne la visualisation des variables, de leur evolution et des operations qui leur sont appliquees. Cette partie permet d'associer des representations animees aux objets du langage C++. L'originalite de Witness a ce sujet est multiple. Tout d'abord, il organise l'animation d'un programme autour de la notion de type, et en particulier de classe du langage C++. Ensuite, il permet la creation interactive de representations animeesan de faciliter l'utilisation par un auteur de programmes. Enn, Witness permet de representer par des animations non seulement les operations sur une variable, mais aussi son etat. On peut ainsi representer une variable par un clignotant, ou lui associer la vitesse d'un projectile. Ces fonctions d'animation de donnees sont realisees grâce a un serveur d'animation construit avec Whizz. Par ailleurs, la communication entre Witness et le serveur d'animation utilise le modele d'evenements et de ots de Whizz. Witness sert ainsi de terrain d'experience pour l'etude des relations entre noyau fonctionnel et interface.

Une architecture distribuee

Witness est organise selon une architecture repartie, sous la forme de processus communicant entre eux (gure 7.1). D'une part, les animations sont gerees par un serveur d'animation construit avec Whizz. D'autre part, le ou les programmes observes sont geres par Witness, qui est un client du serveur d'animation. Enn, Witness est aussi un client du serveur d'icônes WISh, pour ce qui concerne son interface de commande iconique. Ainsi, Witness ne possede lui-même aucune capacite graphique. Ses competences se limitent a la mise au point de programmes, l'interaction avec l'utilisateur etant realisee a travers les serveurs Whizz et WISh.

L'architecture repartie favorise une separation nette entre les differentes composantes de Witness. Elle permet aussi de resoudre certains problemes techniques. En effet, nous avons vu qu'une des caracteristiques originales de Witness est la representation animee de variables. Ainsi, il est par exemple possible de representer un booleen par un clignotant. Le clignotant fonctionne quand le booleen est vrai, et s'arrête quand il est faux. Or la partie de Witness qui permet d'examiner et d'executer les programmes est incompatible avec l'animation. En effet, il est possible de commander l'execution des programmes grâce a des ordres de manipulation de processus. En particulier, il est possible de modier leur code executable de maniere a être prevenu lorsque l'execution passe par un point de ce code. Or, sous UNIX, l'attente d'un tel evenement

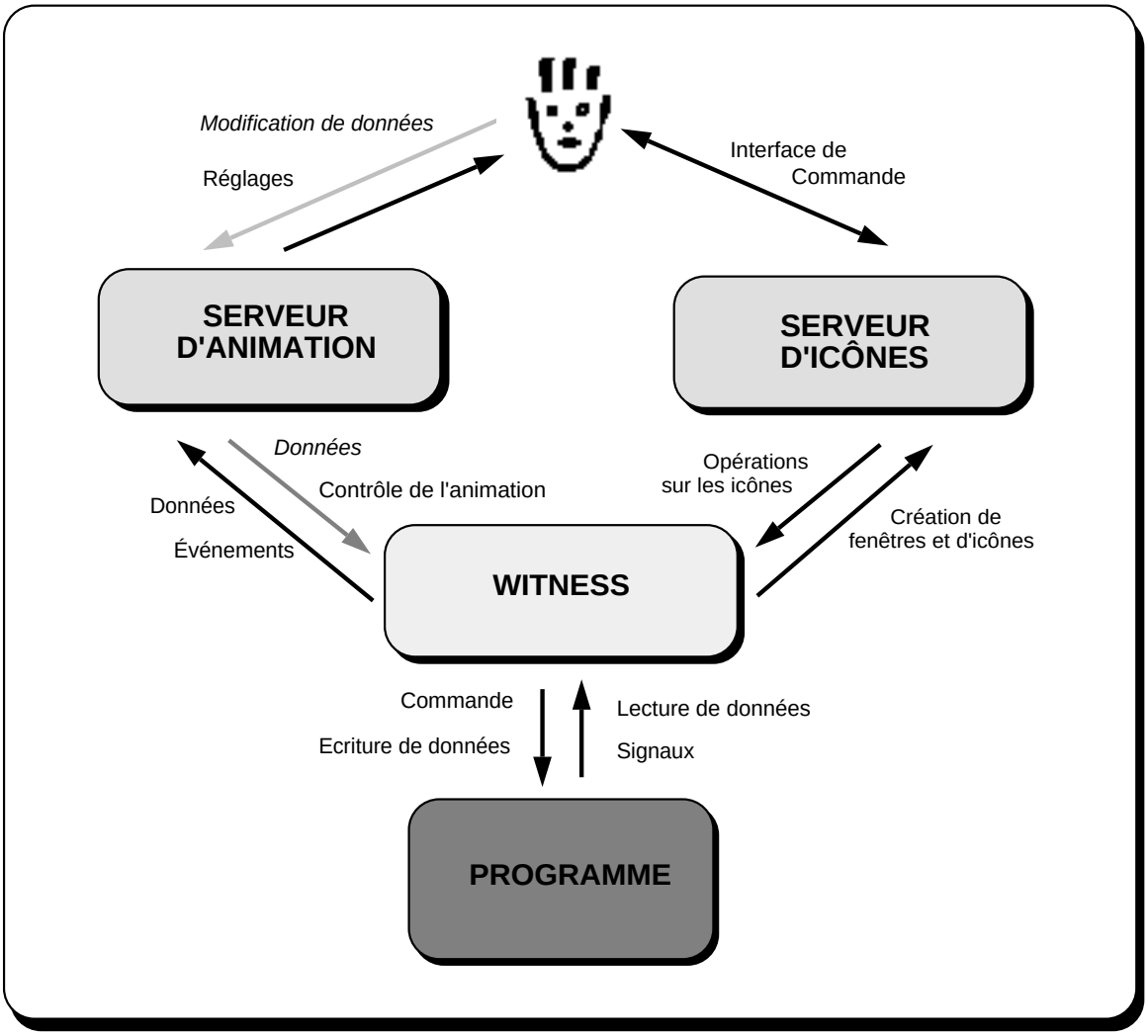

Figure 7.1 : Architecture generale de Witness. Les eches grises et les libelles en italique correspondent a des developpements prevus.

est bloquante. Cela signie que le processus qui commande l'execution donne l'ordre de s'executer au processus trace, puis attend qu'il se produise quelque chose. Aucune action n'est possible durant cette attente. En particulier, le clignotant representant notre booleen serait articiellement arrete pendant ce temps. En fait, il serait tres souvent arrete, puisque l'execution du programme examine est malgre tout le moteur principal de l'observation. Le fait que le serveur d'animation et Witness soient deux processus distincts permet d'eviter ce probleme, le serveur d'animation n'etant pas perturbe par les blocages dans Witness.

La suite de ce chapitre est consacree a la maniere dont Witness utilise les services de WISh et de Witness pour son interface de commande et la visualition de donnees. Les details concernant l'interaction avec les programmes mis au point seront trouves a l'annexe D.

7.3 L'interface de commande

Les outils de mise au point traditionnels ont une interface textuelle. Outre le fait qu'elle est plus agreable a manipuler, une interface iconique presente des avantages fonctionnels, similaires a ceux que presente le Finder du Macintosh vis-a-vis des interfaces textuelles de systemes d'exploitation. Le principal avantage est que l'utilisateur dispose en permanence d'un grand nombre d'informations immediatement accessibles. Ainsi, toutes les variables et fonctions d'une application sont visibles a chaque instant, de meme que les points d'arret et leur etat, ou eventuellement l'arbre d'execution du programme. Nous allons etudier la maniere dont l'utilisateur peut acceder a ces informations et agir sur elles, apres avoir examine les services offerts par le serveur d'icones WISh.

Le serveur d'icones WISh

WISh [Beaudouin-Lafon 88] est un serveur d'interface iconique. En reponse aux requetes de ses clients, il cree et gere des fenetres et des icones. Les actions de l'utilisateur sont interpretees en fonction d'un chier de conguration, et des evenements de haut niveau sont retransmis vers les clients. La gestion que WISh prend en charge correspond a toute la partie traditionnelle des interfaces iconiques : tri et disposition des icones dans les fenetres, selection et deplacement des icones par l'utilisateur, ou encore iconication d'une fenetre.

Le chier de conguration contient des denitions de classes de fenetres et d'icones. Pour chaque classe, on denit des parametres visuels (geometrie par defaut, images des differents etats de l'icone) et une table associant des actions de l'utilisateurs et des operations a effectuer. Les classes peuvent heriter de classes prealablement

denies. Il en est de même pour les tables d'association. Les operations peuvent être internes a WISh (ouvrir la fenêtre correspondant a une icône, ou lancer un programme). Elle peuvent aussi être retransmises vers les clients. Il existe des operations predenies, qui correspondent en general a la notication au client d'operations effectuees par WISh : ouverture d'une fenêtre, par exemple. Le chier de conguration peut aussi contenir des declarations de nouveaux noms d'operations, correspondant a des capacites particulieres de l'application. La gure 7.2 contient un fragment du chier de conguration utilise pour Witness. On y voit comment les operations *SET* et *UNSET* peuvent être envoyees a l'application, ce qui permet de positionner ou inhiber un point d'arrêt depuis son icône.

```
# declaration de nouvelles operations
operation SET;
operation UNSET;

# la table d'association de base entre actions et operations
map BasicWitIconMap (BasicIconMap) {
    key 'i'  in icon => INFORMATION icon;
}

# les icones representant des points d'arrêt
icon type WitBreak {
    icon list = "bp_set","bp_notset";
    rank = 60;
    map (BasicWitIconMap) {
        key 's'  in icon => SET icon;
        key 'u'  in icon => UNSET icon;
    }
}
```

Figure 7.2 : Un extrait du chier de conguration de WISh, qui denit les icônes representant les point d'arrêt. Si l'utilisateur tape la touche 's' sur une telle icône, Witness devra executer l'operation SET sur l'objet correspondant.

Acces a l'information

La base de l'interface de Witness est un arbre de fenêtres et d'icônes, similaire a celui du Finder qui gere le systeme de chiers du Macintosh. La structure de cet arbre mime la structure syntaxique du programme. En effet, cette derniere consiste en une hierarchie de conteneurs imbriques, dans lesquels on trouve des symboles. Tout d'abord, un programme executable est organise en chiers sources. Chaque chier contient des variables globales et des fonctions. Chaque fonction comprend elle-même

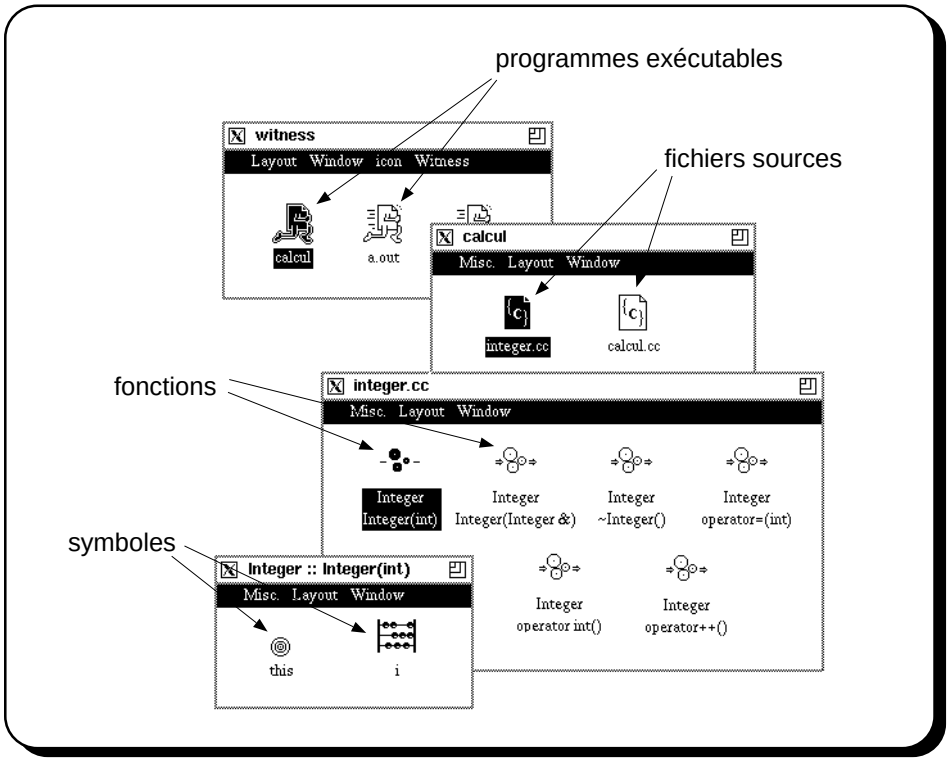

Figure 7.3 : La hierarchie des blocs syntaxiques est representee par un arbre de fenêtres contenant des icônes. Le contenu des chiers *integer.cc* et *calcul.cc*, qui ont servi a fabriquer le programme *calcul*, sont fournis dans l'annexe D. Le premier contient une classe *Integer* pour manipuler des nombres entiers. Le second utilise cette classe dans un calcul simple.

des variables statiques ou automatiques, et des blocs syntaxiques correspondant a des structures de contrôle comme *if*, *for* ou *while*. Chacun de ces blocs peut a nouveau contenir des variables locales et d'autres blocs, a l'inni. Witness associe donc une icône et une fenêtre a chaque bloc, fonction ou chier source. Cette fenêtre contient une icône pour chaque symbole de variable appartenant au bloc.

Au-dela de la structure syntaxique du programme, d'autres informations peuvent être representees par des icônes. Ainsi, un certain nombre de types sont denis dans chaque chier source. Ce sont les types predenis, entiers, ottants, ou caracteres, mais aussi les types denis par le programmeur : classes, pointeurs ou tableaux. Witness represente ces types par des icônes dans une fenêtre. Enn, Witness represente aussi les points d'arrêt par des icônes, et utilise deux etats de l'icône pour representer les points d'arrêt actifs ou inhibes (gure 7.4).

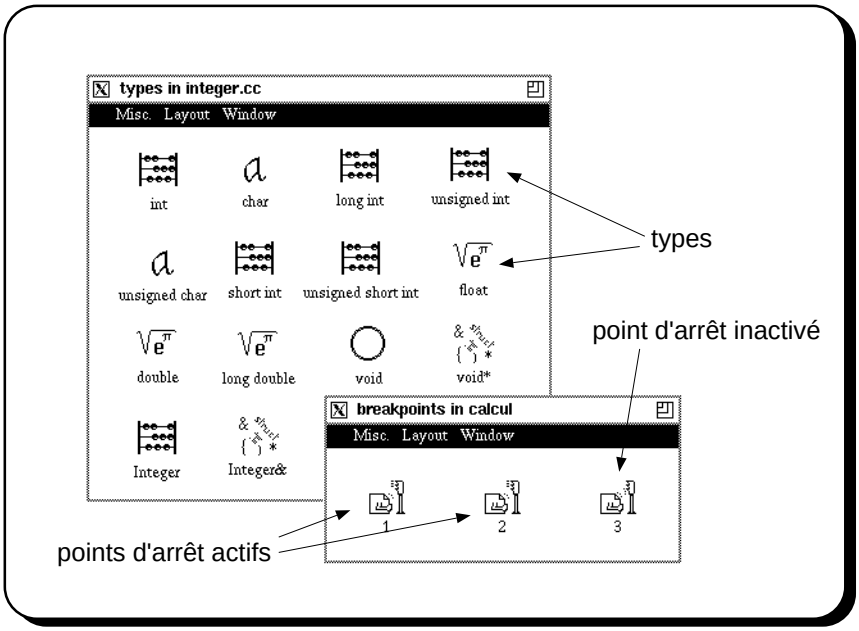

Figure 7.4 : Les points d'arrêt dans un programme et les types dans un chier source. Aux differents etats d'un point d'arrêt correspondent les differents etats des feux de circulation.

Manipulation

Les icônes sont utiles pour visualiser des informations. Ainsi, dans Witness, elles representent des blocs, des symboles ou encore des points d'arrêt. Mais les icônes peuvent aussi servir de support a l'interaction. Les utilisateurs du Macintosh sont habitues a effectuer des double-clics sur des icônes, ou encore a les selectionner

avant de choisir des commandes par des touches du clavier ou des menus. De la même maniere, WISh permet l'interaction a travers les icônes, et Witness utilise cette possibilite.

Quatre categories d'interaction sont possibles :

- cliquer ou double-cliquer sur une icône
- selectionner une ou plusieurs icônes, puis choisir une commande dans un menu.
- positionner le curseur de la souris sur une icône et enfoncer une touche du clavier.
- deplacer une icône avec une action cliquer-tirer, et l'amener dans une autre fenêtre.

Dans les quatre cas, l'action de l'utilisateur est soit ignoree, soit transformee en une operation, en fonction du chier de conguration. Dans certains cas, WISh peut demander des complements d'information sur l'action effectuee, en ouvrant une boîte de dialogue.

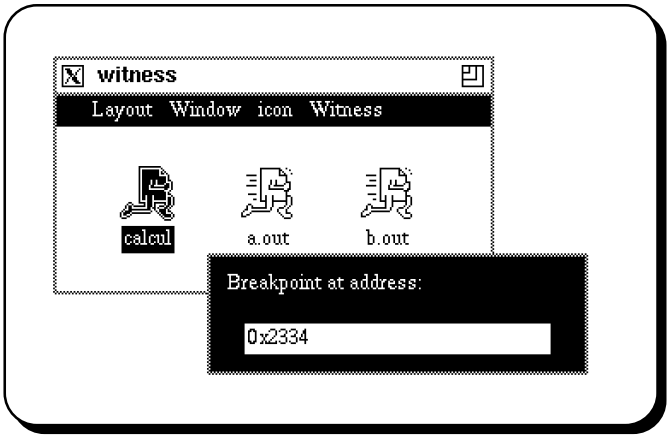

Figure 7.5 : L'utilisateur a enfonce la touche 'b' au dessus de l'icône du programme *calcul*. Une boîte de dialogue apparaît pour saisir un argument numerique, et Witness recevra l'operation *BREAK* avec cet argument.

Witness ne percoit donc les actions de l'utilisateur que sous la forme de couples (operation, icône), avec eventuellement des arguments numeriques ou textuels supplementaires. C'est lui qui donne un sens aux actions, chaque operation sur une icône se traduisant en une operation sur l'objet represente. Ainsi, Witness sait repondre aux operations *RUN, BREAK* et *BREAKLIST* sur les icônes representant des programmes. *RUN* se traduit par l'execution du programme. *BREAK*, qui doit être accompagne d'un argument numerique, declenche l'insertion d'un point d'arrêt a l'adresse speciee. Enn, *BREAKLIST* declenche l'ouverture de la fenêtre des points d'arrêt. Avec le chier de conguration de WISh utilise par l'auteur, ces operations sont produites par l'enfoncement des touches 'r', 'b' et 'l' au-dessus de l'icône d'un programme. L'enfoncement

de la touche 'b' est suivi de l'apparition d'une boîte de dialogue pour saisir l'adresse du point d'arrêt (gure 7.5).

Dans Witness, toutes les fonctions rencontres habituellement dans un \debugger" sont accessibles par des operations sur des icônes. Nous venons de voir que le lancement de l'execution ou la pose d'un point d'arrêt a une adresse sont realisees a travers l'icône du programme. Les autres fonctions, quant a elles, peuvent être rattachees a des entites plus precises du programme. Elles sont donc realisees par des actions sur les icônes representant ces entites. Ainsi, la pose d'un point d'arrêt a une ligne d'un chier source et la visualisation des types denis dans ce chier se font a travers son icône. A travers l'icône d'une fonction, on peut placer un point d'arrêt au debut de la fonction ou provoquer son appel. Un point d'arrêt peut être active, desactive ou detruit grâce a son icône. De maniere plus generale, l'utilisateur peut demander des informations sur une entite grâce a une operation valable pour toutes les icônes : description d'un type, adresse et parametres d'une fonction, type et valeur d'une variable, etc. Ces informations sont actuellement presentees sous forme textuelle. Enn, une operation permet de designer les symboles dont on veut visualiser les valeurs. Dans ce dernier cas, l'operation se traduit par la creation d'une representation animee de la variable.

7.4 L'animation des variables

Witness permet l'animation des objets appartenant a des classes C++, mais pas des autres objets : types de base, tableaux ou pointeurs. Ce choix se justie par des raisons techniques, detaillees dans l'annexe D. Il se justie aussi parce que les classes C++ constituent des ensembles complets de donnees et de traitements, et representent le niveau auquel les programmeurs concoivent l'essentiel de leurs programmes. Le but de Witness est donc de permettre a un programmeur de construire une representation pour une classe C++, an de visualiser ses instances et leur evolution.

Les informations disponibles

On peut decomposer en trois categories les informations disponibles pour animer les instances d'une classe C++. Tout d'abord, au cours de l'analyse statique du programme, Witness recueille une description complete et precise de la structure des classes. En particulier, il connaît le nom et le type des divers champs de la classe. C'est la valeur de ces champs dans les instances de la classe qui determine leur etat. Les representations graphiques sont donc construites sur la base de ces champs. Par exemple, si la description d'une classe est constituee d'un entier et d'une chaîne de caracteres, on pourra lui associer une representation constituee d'un cadran et d'une chaîne de caracteres, ou encore d'un cadran seul.

La deuxieme categorie d'informations est a rapprocher des operations de l'animation d'algorithme. Ici, Witness considere que les operations a visualiser sont les appels de methodes des classes examinees. Il detecte le debut et la n de l'appel de ces methodes, et utilise ces informations de deux manieres. D'une part, pour des representations simples de donnees, Witness permet de suivre leur evolution au cours de l'execution. Pour cela, il est important de surveiller leur valeur de la maniere la plus reguliere possible, surtout lorsqu'on est a la recherche d'une modication accidentelle. Witness met a jour les representations des variables au debut et a la n des methodes, ce qui permet d'identier plus facilement une methode suspecte. Par ailleurs, pour des representations plus evoluees, Witness permet de visualiser les operations elles-mêmes, en utilisant le debut des methodes et leurs parametres.

La troisieme categorie d'informations est techniquement proche de la seconde, mais elle revêt une importance cruciale : il s'agit des evenements de creation et de destruction d'instances. En effet, les programmes ecrits en C++, comme dans les autres langages a objets, ont une forte nature dynamique. Les objets y sont frequemment crees explicitement (et moins souvent detruits helas), voire automatiquement, lorsque ce sont des objets locaux a une fonction ou un bloc syntaxique. Witness detecte donc ces creations pour pouvoir representer les nouvelles instances.

Toutes ces informations permettent une animation dele des donnees en fonction de leur evolution. C'est aussi sur la base de ces informations que l'utilisateur peut choisir, voire construire, la representation d'une classe. C'est en effet lui qui doit decrire l'association entre classe et representation. Cette description se fait actuellement dans un chier de conguration ou, pour chaque classe a representer, l'utilisateur donne l'association entre d'un côte les champs a observer et les methodes a animer, et de l'autre des objets decrivant l'animation. Nous allons maintenant nous interesser a cette description de l'animation.

La description de l'animation

L'animation dans Witness utilise un serveur d'animation bâti avec Whizz. Ce serveur permet trois types de communication : avec des ots et des evenements, selon le modele de Whizz, et avec des requêtes, pour creer des modules et des connexions. On peut en particulier utiliser des ordres de creation de scenes, qui permettent de charger dans le serveur des animations et des objets animes deja construits.

L'animation dans Witness est realisee en instanciant des scenes dans le serveur d'animation, et en etablissant des connexions entre les donnees du programme et ces scenes. Ce choix permet de decomposer les responsabilites : d'une part, la scene construite a l'avance est responsable de la representation des informations qu'elle recoit ; d'autre part, Witness est responsable de l'etablissement des connexions en fonction de la conguration de la classe. Ainsi, l'utilisateur intervient a deux endroits :

d'une part, il construit une scene avec l'editeur a manipulation directe Whizz'Ed, et d'autre part il decrit dans un chier de conguration la maniere dont cette scene est utilisee pour representer des informations.

```
type Integer {
 icon WitInt;
 repr integer;   # represent an Integer by the scene \integer"
 observ Value;   # connect Value to the rst slot
};
```

Figure 7.6 : Un extrait du chier de conguration de Witness. Le type *Integer* est represente par la scene *integer*, le champ *Value* etant associe a la premiere entree de la scene.

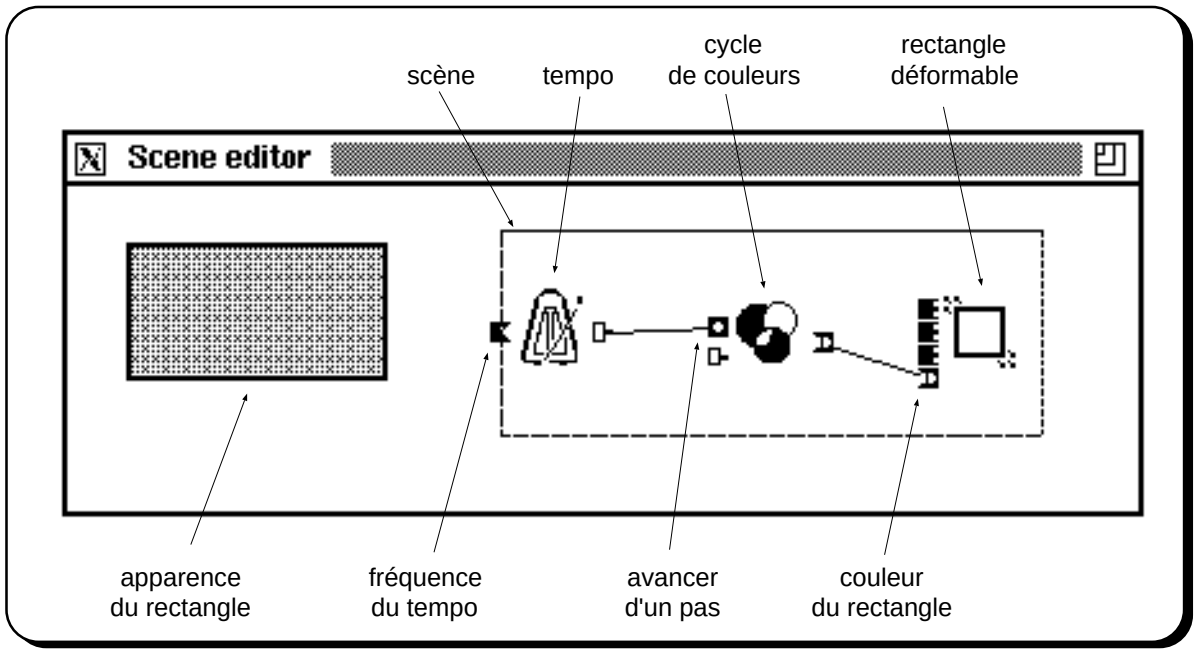

Figure 7.7 : La construction d'une scene pour representer un entier par la frequence de clignotement d'un rectangle.

Witness met ainsi en uvre deux utilisations principales des scenes. Tout d'abord, l'utilisateur associe une scene a une classe. Cette scene contient la representation de la classe, et ses divers points d'entree sont utilises pour contrôler l'apparence de la representation en fonction de l'etat de l'objet represente. Pour cela, Witness utilise le mecanisme des valeurs actives. Pour chaque champ interessant de la classe, une valeur active est creee et connectee a la scene, selon le modele de ots de Whizz. On peut

ainsi creer tres facilement des representations complexes pour des classes. Prenons l'exemple simple d'une classe *Integer* ayant pour unique champ une valeur entiere appelee *Value*. Il suft pour la representer de creer une scene ayant un connecteur d'entree de type entier, et de declarer la connexion entre ce connecteur et le champ *Value* dans le chier de conguration. La gure 7.7 montre une scene ou la valeur de l'entier determine la frequence de clignotement d'un rectangle. Le gure 7.8 contient une scene ou l'entier commande le deplacement de la colonne dans un thermometre.

La seconde utilisation des scenes est liee a la visualisation des operations. Pour representer une operation par une animation, il suft de construire une scene d'animation et de lui associer le nom de la methode dans le chier de conguration. Lorsque la methode est invoquee, la scene est instanciee, provoquant l'animation desiree.

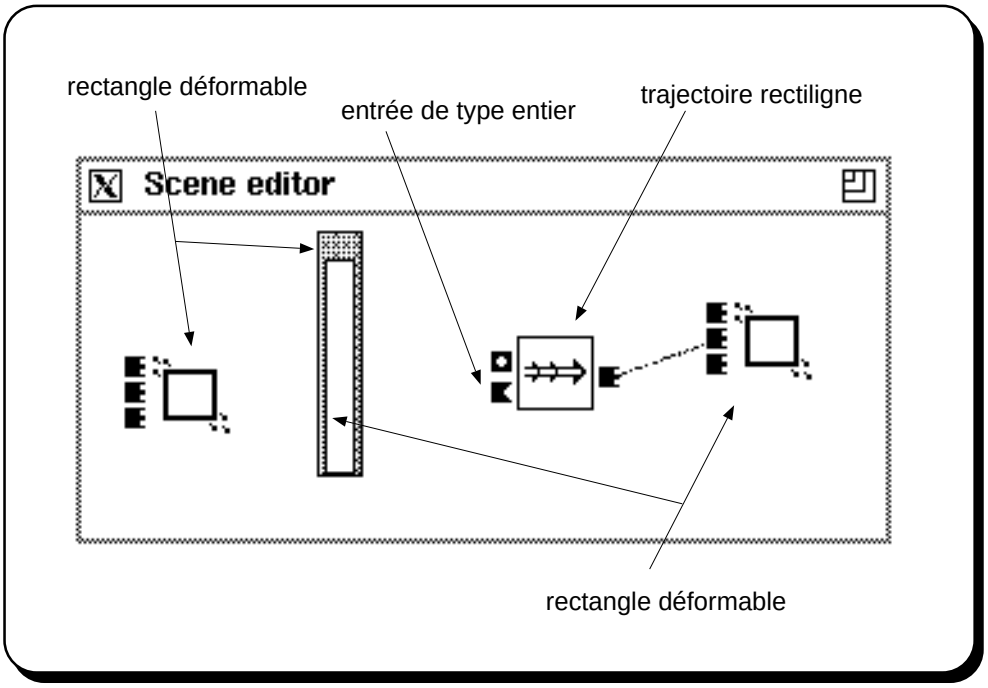

Figure 7.8 : Une scene pour representer un entier par un thermometre. Un rectangle reste xe, tandis que le coin superieur gauche de l'autre se deplace sur une trajectoire rectiligne. La visualisation graphique de la trajectoire est ici masquee par le rectangle xe.

On peut realiser cette instanciation selon deux techniques. La premiere technique, utilisee actuellement, consiste a analyser dans Witness les informations recueillies lors de l'appel de la methode. En particulier, Witness determine pour quel objet la methode a ete appelee, ce qui permet d'instancier la scene avec les bons parametres. C'est alors un ordre de creation de scene qui transite entre Witness et le serveur d'animation. Cette

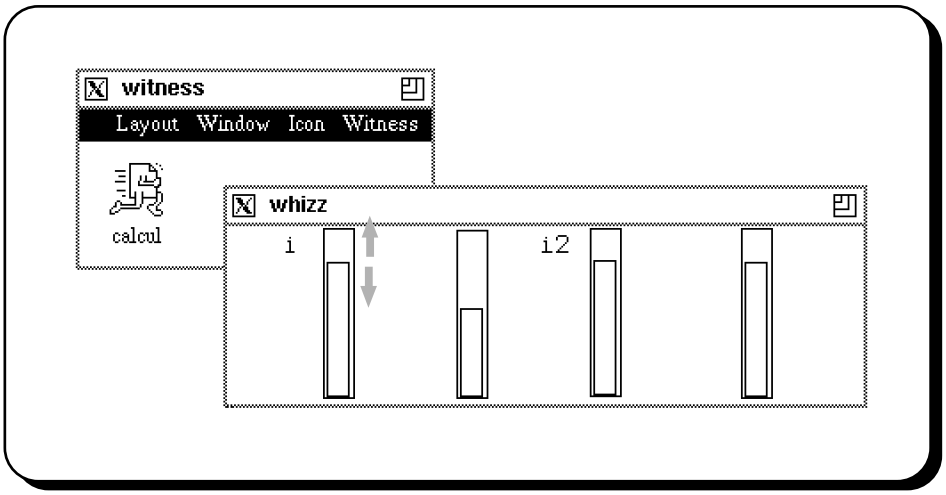

Figure 7.9 : La scene contruite a la gure 7.8 est utilisee pour visualiser le programme *calcul*. On peut y suivre les valeurs de la variable *i*, mais aussi decouvrir que deux instances anonymes de la classe *Integer* ont ete creees.

technique n'est pas totalement satisfaisante, car elle ne respecte pas un bon decoupage entre le noyau fonctionnel (Witness) et l'interface (Whizz). En effet, pour fournir les parametres a la scene de la methode, Witness doit connaître des informations de bas niveau sur la scene de l'objet, ce que l'on cherche en general a eviter. L'autre technique consisterait a ne pas instancier la scene dans Witness, mais plutôt a utiliser le modele de Whizz, et a envoyer un evenement vers le serveur d'animation. C'est alors ce serveur qui pourrait instancier la scene en fonction du contenu de l'evenement, et de la maniere de le traiter qu'on lui aurait precedemment indiquee. Helas, Whizz ne permet pas encore ce type de comportement (voir sections 5.2.3 et 5.8).

7.5 Problemes ouverts

La representation animees des donnees est encore au stade de prototype dans Witness, et un certain nombre de problemes subsistent.

Tout d'abord, comme nous venons de le mentionner, Whizz ne permet pas encore l'instanciation automatique de scenes d'animation. Il serait souhaitable que Witness signale les appels de fonction par des evenements, et que le serveur d'animation reagisse en instanciant des scenes d'animation. Or il manque un mecanisme pour decrire l'instanciation des scenes, et surtout la maniere de les connecter aux modules existants. En l'absence de ce mecanisme, la visualisation des operations sur les variables est restee embryonnaire dans Witness.

Ensuite, Witness utilise Whizz pour creer des representations de variables, mais aucun service n'est fourni pour gerer la disposition de ces representations a l'ecran. A l'heure actuelle, Witness juxtapose simplement les representations, ce qui s'avere insufsant lorsque les variables representees deviennent nombreuses. Un outil reellement utilisable devra fournir des mecanismes de gestion de l'espace, comme le fait Incense par exemple.

Nous avons vu par ailleurs qu'une nouveaute de Witness est de permettre la creation interactive de representations de variables. Cet atout se transforme en inconvenient lorsqu'il n'existe pas d'autre moyen de creer des representations. En effet, un programmeur n'est pas toujours dispose a depenser du temps pour creer une representation de ses donnees. Il souhaite aussi disposer de representations fabriquees automatiquement, par exemple sous la forme de botes contenant du texte, comme dans Incense. Une solution intermediaire consiste a fournir de nombreuses representations predenies, parmi lesquelles on pourra choisir la plus adaptee. Mais il est probable que la construction automatique de representations \sur mesure" sera necessaire dans un systeme operationnel.

Enn, Witness utilise des representations graphiques animees pour visualiser l'evolution des donnees, mais il ne permet pas la manipulation de ces reprsentations par l'utilisateur. Pourtant, il serait plaisant de pouvoir modier la valeur d'une variable en agissant sur sa representation. Il sufrait pour cela de specier plus completement le comportement dynamique de la representation, et de denir ses reactions aux actions de l'utilisateur. Le modele de Whizz permet d'envisager de tels developpements, qui n'ont cependant pas encore ete etudies.

7.6 Conclusion

Ce chapitre nous a permis d'examiner les differents aspects de Witness, un outil de mise au point de programmes utilisant l'animation de donnees. Nous avons vu la maniere dont il utilise une interface iconique pour permettre l'acces aux informations recherchees. Nous avons aussi etudie la maniere dont Witness utilise Whizz pour representer les donnees et leurs operations, autorisant ainsi la construction interactive de representations animees. Witness est encore un systeme experimental. Les mecanismes d'animation de donnees y sont encore rudimentaires, et demanderont de nombreuses ameliorations avant d'etre reellement utilisables sans connaissance prealable du systeme. Cependant, Witness demontre comment Whizz peut etre utilise a la fois pour representer des donnees de maniere animee et pour etablir le lien entre le noyau fonctionnel et l'interface d'une application. De plus, il ouvre la voie a des outils graphiques de mise au point et d'exploration de programmes plus facilement utilisables.

Chapitre **8.**

Conclusion

Nous avons etudie dans cette these divers aspects de l'animation dans les interfaces homme-machine. S'agissant d'un domaine nouveau, il nous a d'abord fallu envisager les applications possibles. Nous avons vu qu'il en existait de nombreuses, depuis les utilisations purement cosmetiques jusqu'a celles qui melangent animation et manipulation directe pour donner de nouveaux types d'interaction. Ces remarques, couplees a des observations sur les mecanismes des interfaces, nous ont permis de denir des besoins plus generaux que l'animation au sens strict. Il apparat en effet utile de decrire dans les mêmes termes les trois types de comportement dynamique d'une interface : son evolution avec le temps, avec les actions de l'utilisateur, ou en reponse a l'evolution des donnees. Nous avons par ailleurs examine les aspects techniques mis en jeu dans une telle description.

Sur la base de ces reexions, nous avons presente Whizz, un systeme destine a la construction d'interfaces animees. Whizz est une extension de la boe a outils X_{TV}, dont nous avons examine les principaux mecanismes. Son modele, qui distingue evolutions continues et ponctuelles, decrit un systeme anime comme un ensemble d'objets communiquant par des ots de donnees et des evenements. Nous avons vu comment, grâce a ce modele, Whizz pouvait être utilise pour melanger animation et interaction.

Nous nous sommes enn penches sur le domaine de l'animation de programmes. Nous avons vu comment Whizz permet la creation de representations animees de donnees, utilisables pour suivre l'evolution d'un programme. Cette possibilite est exploitee dans Witness, un outil de mise au point graphique de programmes. Witness, outre la presentation animee des donnees, possede une interface iconique originale. De plus, il permet d'experimenter le modele de Whizz pour la communication entre interface et noyau fonctionnel.

Ces diverses etudes et resultats ouvrent un certain nombre de perspectives. D'un point de vue concret tout d'abord, les travaux sur Witness permettent d'envisager la creation d'outils de mise au point entierement graphiques, ou l'animation des variables

et des operations serait une tâche aisee. L'apparition de tels outils sur le marche constituera un progres appreciable pour de nombreux programmeurs, similaire a celui qu'ont apporte les \debuggers symboliques" il y a quelques annees.

Par ailleurs, la construction de representations animees de donnees a de nombreuses applications autres que la mise au point de programmes. Ainsi, tous les systemes du type \tableau de bord", utilises pour contrôler des centrales nucleaires ou d'autres processus industriels, font appel a de telles representations. L'exemple de Whizz'Ed montre qu'on peut envisager le developpement d'outils pour faciliter leur creation.

Venons en maintenant aux perspectives de recherche ouvertes par ces travaux. Tout d'abord, l'une des raisons qui ont motive la realisation de Whizz est la conviction que l'introduction de l'animation dans les interfaces est beneque. Cette conviction etait difcile a etayer tant que la construction d'interfaces animees etait une tâche complexe de programmation. Une boîte a outils comme Whizz, en facilitant cette construction, permettra d'engager des recherches sur le sujet. Desormais, les interfaces animees peuvent devenir un sujet d'etude pour les ergonomes et les psychologues.

Par ailleurs, de nouveaux types d'interfaces font leur apparition. D'une part, certains peripheriques permettent maintenant de suivre l'etat de l'utilisateur en permanence, et plus seulement ses actions volontaires. D'autre part, le collecticiel, qui permet a plusieurs utilisateurs d'agir ensemble, doit les aider a mettre en commun leurs actions, en tenant compte de leurs etats respectifs. On peut esperer modeliser ces nouvelles interfaces, qui utilisent de multiples sources d'information et font une place importante a la communication, grâce a des systemes comme Whizz. En effet, ses capacites a decrire des processus dynamiques et a faire communiquer des agents, qu'ils soient objets graphiques, peripheriques d'entree, ou pourquoi pas utilisateurs, en font un bon candidat pour decrire de telles interfaces.

Enn, une autre contribution de Whizz semble riche de developpements futurs : c'est la meilleure comprehension du comportement dynamique de l'interface et de sa description. Pendant longtemps, on a decrit des interfaces en donnant leur aspect graphique et leurs reponses aux actions de l'utilisateur. Whizz a permis de comprendre que l'aspect reactif d'une interface n'est qu'une partie de son comportement dynamique, ce qui ouvre la voie a l'etude de ce comportement dans son ensemble. De plus, identier et comprendre le comportement dynamique d'une interface permettra d'en donner une description declarative, par opposition a la situation actuelle ou il est reparti dans le programme sous forme procedurale. Cette description declarative permettra ensuite de realiser des outils pour construire interactivement ce comportement.

Whizz et Whizz'Ed representent un premier pas dans cette direction, qui paraît avoir de nombreux prolongements. Dans un premier temps, on peut esperer que de tels outils permettront la construction interactive d'interfaces a manipulation directe. Par la suite, ces outils devraient accompagner l'apparition d'interfaces encore plus evoluees.

En particulier, Whizz permet d'envisager des interfaces vivantes, composees non plus d'objets inertes que l'on manipule, mais d'objets vivants avec qui on interagit. Dans la mesure ou l'homme, et en particulier son il, reagit mieux a un environnement en mouvement qu'a une scene immobile, on peut esperer que ces interfaces seront a la fois plus intuitives et plus efcaces, c'est-a-dire meilleures.

Annexe **A.**

Le jeu de cartes

Nous presentons ici des extraits du programme de jeu de cartes realise avec X_{TV}, et prsente au chapitre 3. Seules les fonctions les plus signicatives sont presentees. Ce sont toutes des methodes des classes *TransatStage* et *CardActor*.

Regardons d'abord le fonctionnement de la scene-jeu (TransatStage). C'est a la fois une scene (STAGE) et un jeu de patience (Transat). Elle se voit associer un ltre pour les evenements au clavier, un pour les cliquer-tirer, correspondant aux deplacements de cartes avortes, et un pour les clics, utilise pour deselectionner les cartes.

```
TransatStage :: TransatStage ()
: STAGE (),
  Transat ()
{
   KeyHandler = new FILTER (KeyboardDn, &TransatStage::CommandOperation);
   AbortDragFilter = new FILTER (DragEnd, &TransatStage::AbortDragOperation);
   DeselectHandler = new FILTER (MouseButtonDn, &TransatStage::DeselectOperation);
   AbortDragFilter→Add (*this);
   DeselectHandler→Add (*this);
   KeyHandler→Add (*this);
   Selection = 0;
}
```

La fonction qui cree une nouvelle carte est redenie pour creer des acteurs-cartes, et les faire apparaître sur la scene.

```
Card* TransatStage :: CreateCard (CardRank r, CardColor c)
{
   CardActor* card = new CardActor (r, c, CardRank (r), c);
   card→Appear (*this);
   return card;
}
```

La repartition des evenements utilise le calcul de la carte en fonction de la position.

```
REACTIVE* TransatStage :: Dispatch (EVENT& ev)
{
   CardActor* card = PositionToCard (ev.Position ());
   if (card) return card; else return this;
}
```

La fonction de reafchage utilise la disposition en tableau, pour assurer un ordre de reafchage agreable a l'il.

```
void TransatStage :: Redisplay (VIEW& v)
{
   for (CardRank r = Ace; r ≤ Empty; r++)
      for (CardColor c = Hearts; c ≤ Clubs; c++) {
         CardActor* card = (CardActor*) Layout [r][c];
         card→Redisplay (v);
      }
}
```

Examinons maintenant les cartes. Ce sont des acteurs specialises (ACTOR), ainsi que des cartes (Card). Elles utilisent un environnement graphique, et se voient associer un ltre correspondant aux operations de selection, et au debut des actions de cliquer-tirer. Les cases vides sont des cartes speciales, avec une representation invisible et un ltre pour la n des actions cliquer-tirer, qui declenche les permutations.

```
CardActor :: CardActor (CardRank r, CardColor c, CardRank rpos, CardColor cpos)
: ACTOR (),
  Card (r, c)
{
   Env = new GENV (XtvRed, Replace, 1, XtvWhite, Replace);
   SelectHandler = new FILTER (MouseButtonDn, &CardActor::SelectOperation);
   EndDragFilter = new FILTER (DragEnd, &CardActor::SwapOperation);
   PosRank = rpos;
   PosColor = cpos;
   POINT pos = ((TransatStage*) Stage)→CardPosition (rpos, cpos);
   if (r == Empty) {
      Icon = new ICON (Mask, Mask, pos);
      EndDragFilter→Add (*this);
   } else {
      pPATTERN img = new PATTERN (card_mask_width, card_mask_height, Images [c][r]);
      Icon = new ICON (img, Mask, pos);
      SelectHandler→Add (*this);
   }
   Selected = FALSE;
}
```

Le reafchage d'une carte. Un seul environnement est utilise, dont on change les couleurs selon la couleur de la carte et le fait qu'elle est ou non selectionnee.

```
void CardActor :: Redisplay (VIEW& v)
{
   int red_card = (Color == Hearts) || (Color == Diamonds);
   if (Selected) {
      Env→Set (FillColor (XtvBlack));
      Env→Set (BorderColor (red_card ? XtvRed : XtvWhite));
   } else {
      Env→Set (FillColor (Wheat));
      Env→Set (BorderColor (red_card ? XtvRed : XtvBlack));
   }
   Icon→Draw (v, *Env);
}
```

La fonction appelee quand on clique sur une carte. Si elle est deja selectionnee, cela provoque sa deselection. Dans le cas contraire, la carte est selectionnee, et une action de cliquer-tirer est commencee.

```
void CardActor :: SelectOperation (EVENT& ev)
{
   if (Selected) {
      ((TransatStage*) Stage)→ChangeSelection (NIL);
   } else {
      ((TransatStage*) Stage)→ChangeSelection (this);
      new MoveAction (ev, *Stage, *Ghost, Position () - ev.Position ());
   }
}
```

La fonction appelee quand une action cliquer-tirer se termine sur une case vide. On verie que la permutation est legale, et on l'effectue.

```
void CardActor :: SwapOperation (EVENT& ev)
{
   TransatStage* s = (TransatStage*) Stage;
   CardActor* sel = s→GetSelection ();
   s→AbortDragOperation (ev);
   if (s→MaySwap (sel→PosRank, sel→PosColor, PosRank, PosColor))
      s→Swap (sel→PosRank, sel→PosColor, PosRank, PosColor);
   else
      s→BadOperation ();
}
```

Annexe **B.**

Le lancer d'une icône

Les extraits de programme suivants sont les parties signicatives de la capture d'une icône par la poubelle dans l'exemple d'interface iconique animee de la section 5.6.

Le module qui gere le lancement d'une icône : il effectue des statistiques pendant que l'utilisateur la deplace, puis gere la suite du mouvement.

```
Launcher* TheLauncher;
```

La fonction appelee quand une icône recoit une note.

```
void Icon :: Hear (const Note& n, InPlug& in_plug)
{
   if (&in_plug == &FacePlug) /* changement d'aspect */
      NextObj ();
   else if (&in_plug == &MovePlug) /* deplacement */
      NextPos = ((PointNote&) n).GetValue ();
   Heard ();
}
```

La fonction appelee quand une icône est capturee.

```
void Icon :: Captured (const TriggerEvent& ev)
{
   /* Calcul du centre d'attraction */
   Box* box = (Box*) ev.Source ();
   RECT zone = box→GetFrame ();
   POINT center = (zone.TopLeft + zone.BotRight) / 2;
   /* Creation de nouveaux modules */
   Attractor *a = new Attractor (Position (), TheLauncher→GetSpeed (), center);
   Tempo* t = new Tempo (TheLauncher→GetInterval ());
   /* Reconguration des connexions */
   MovePlug.Isolate ();
   MovePlug.Connect (a→Out);
   a→Step.Connect (t→Out);
}
```

La fonction appelee quand l'utilisateur commence a deplacer une icône.

```
void Icon :: NewDrag (EVENT& ev)
{
   /* Creation du module qui rend compte de l'action */
   DragInstr* a = new DragInstr (ev);
   /* Reconguration des connexions */
   TheLauncher→In.Connect (a→Out);
   MovePlug.Isolate ();
   MovePlug.Connect (TheLauncher→Out);
}
```

La methode qui sensibilise une icône active (la poubelle) a une autre icône.

```
void ActiveIcon :: Sensitize (Icon& i)
{
   Field.AddTarget (i);
   Field.AddClient (i, (ModuleCallback) (&Icon::Captured));
}
```

Le programme principal.

```
main ()
{
   XtvOpen ();
   /* Creation de la scene et la vue */
   STAGE s;
   VIEW v (s);
   /* Creation de la poubelle */
   ActiveIcon trash ("trash");
   trash.Appear (s);
   /* Creation d'une icône */
   Icon le ("cprog");
   le.Appear (s);
   trash.Sensitize (le);
   /* Connexion de l'icône au lanceur */
   TheLauncher = new Launcher;
   le.MovePlug.Connect (TheLauncher→Out);
   /* Creation du ltre qui detecte les clics de l'utilisateur sur la souris */
   HANDLER h (MouseButtonDn, le, Icon::NewDrag);
   /* Lancement d'X_TV. */
   XtvMainLoop ();
   XtvClose ();
}
```

Annexe **C.**

Le programme de calcul

Le chier *integer.cc* dont nous reproduisons ici le contenu denit la classe *Integer*, qui simule dans une classe le fonctionnement d'un nombre entier. Elle s'utilise comme le type entier de base, mais permet a Witness de representer ses instances.

```
class Integer {
protected:
   int Value;
public:
   Integer (int = 0);
   Integer (Integer&);
   ~Integer ();
   Integer& operator = (int);
   Integer& operator = (Integer&);
   operator int ();
   Integer& operator ++ ();
   Integer& operator -- ();
};

Integer :: Integer (int i)
{
   Value = i;
}

Integer :: Integer (Integer& I)
{
   Value = I.Value;
}

Integer :: ~Integer ()
{
}
```

```
Integer& Integer :: operator = (int i)
{
   Value = i;
   return *this;
}

Integer& Integer :: operator = (Integer& i)
{
   Value = i.Value;
   return *this;
}

Integer :: operator int ()
{
   return Value;
}

Integer& Integer :: operator ++ ()
{
   Value++;
   return *this;
}

Integer& Integer :: operator -- ()
{
   Value--;
   return *this;
}
```

Par ailleurs, le chier *calcul.cc*, contient un calcul simple effectue avec la classe *Integer* :

```
main ()
{
   Integer i = 5;
   i = i - 2;
   i++;
}
```

Annexe **D.**

Mecanismes de mise au point

D.1 L'analyse statique des programmes compiles

La maniere dont sont stockes les programmes prêts a être executes est a peu pres independante du langage dans lequel ils ont ete ecrits. Mais elle varie selon les systemes d'exploitation. Même sous Unix, les variantes sont nombreuses. Cependant, les differences sont surtout syntaxiques, et on retrouve heureusement les mêmes structures de base. De maniere generale, un programme executable et une partie de programme resultant de la compilation d'un chier source ont la même structure. Ils sont principalement compose d'instructions en langage machine, d'un segment de donnees contenant les valeurs initiales de certaines variables, et d'une table des symboles. Cette derniere est utilisee par les editeurs de liens pour assembler les parties de programmes provenant de compilations separees. Elle est facultative dans les executables, et est souvent eliminee pour economiser de la place.

A partir d'un executable sans table des symboles, il est tres difcile d'examiner un programme. On ne dispose que d'instructions et de donnees, sans aucune trace des noms des fonctions et des variables qui composent le programme d'origine. Ces noms sont fournis par la table des symboles, avec en regard leur adresse. Ainsi, il est possible de trouver l'adresse d'une variable quand on connaît son nom, et donc d'acceder a sa valeur une fois que le programme est charge en memoire et execute.

Cependant, le nom et l'adresse des variables et des fonctions sont de bien pauvres informations si l'on considere des langages comme Pascal, C++ ou ADA. Il manque deux categories d'informations indispensables. D'une part, il manque des informations sur la structure du programme. La plupart des langages modernes permettent de decrire des programmes sous la forme de blocs syntaxiques imbriques les uns dans les autres. Ces blocs determinent en particulier la visibilite et la signication des identicateurs. Par exemple, les variables locales d'une fonction peuvent porter le même nom que celles d'une autre fonction, sans interference. Avec les informations fournies par la table des symboles, on ne peut pas savoir si la variable *i* situee a

l'adresse[1] *ff7e* est une variable locale de la fonction *f* ou de la fonction *g*. Ensuite, la table des symboles ne renseigne pas sur les types des objets. Le systeme de types a une importance predominante. Il n'a pas la même place cruciale dans l'executable que dans le programme source, ou une affectation entre deux variables est illegale si elles n'ont pas le même type. Mais c'est le type d'une variable qui determine sa taille. C'est aussi lui qui determine la signication de cette variable. Est-ce la representation d'un nombre ottant, d'un rectangle, d'un tableau de caracteres ? Cela fait une difference pour le format dans lequel on va imprimer sa valeur. Si en plus, il s'agit de representer graphiquement les variables en fonction de l'objet abstrait qu'elles representent, le type devient indispensable. Dans des langages a objets comme C++, c'est aussi le type qui determine les fonctions susceptibles de modier une variable. Bien entendu, il reste les modications accidentelles, mais nous y reviendrons.

Donc pour observer et representer convenablement un programme, il faut des informations qui dependent du langage, et qu'on ne trouve pas dans la table des symboles. Heureusement, il se trouve que les langages les plus classiques ont des structures similaires. Ainsi, il a ete possible de denir des formats communs pour representer ces informations. Un format courant est celui de DBX. Lorsqu'une option particuliere du compilateur est choisie, les informations sont representees dans un segment supplementaire du programme executable. Le format DBX permet de representer correctement des programmes ecrits en Pascal, en C, et en FORTRAN, ce dernier langage ayant une structure plus simple. Pour C++, des extensions sont necessaires pour representer les classes, et dans l'avenir la genericite et les exceptions. Le compilateur GNU C++ denit une telle extension, utilisee par GDB et maintenant par Witness. Nous allons maintenant identier les informations disponibles dans un programme executable.

La structure

Tout d'abord, on trouve des informations concernant la structure du programme. Un programme est organise en blocs syntaxiques imbriques qui forment un arbre. Le bloc le plus grand est le programme lui-même. Il est divise en unites de compilation, qui correspondent le plus souvent a des chiers compiles separement. Une unite de compilation contient des fonctions. Les procedures sont ici assimilees a des fonctions qui retournent un type vide. Les fonctions contiennent elles-mêmes d'autres fonctions, ou des blocs syntaxiques de base. En C et en C++, les fonctions imbriquees n'existent pas. Les blocs syntaxiques de base correspondent a des instructions composites. En C, ce sont des blocs d'instructions entre accolades. On les trouve souvent dans des boucles ou des tests, comme dans l'expression `if (test) { ...} else { ...}`. Ces blocs d'instructions peuvent être imbriques a volonte. Les informations recueillies dans le

[1] en fait, les variables locales n'ont pas une adresse, mais une position relative dans la pile d'execution.

programme executable permettent de reconstruire toute la hierarchie. Elles permettent pour chaque bloc de retrouver son nom, son type, et les adresses de debut et de n dans le programme en langage machine.

Les symboles

Par ailleurs, a chaque niveau de la hierarchie, on trouve des symboles representant des variables. Les symboles au niveau du programme sont des variables globales visibles partout. Au niveau des unites de compilation, on trouve des variables globales visibles seulement a l'echelle de l'unite. Enn, dans les fonctions et les blocs d'instructions, on trouve des variables locales, qu'elles soient statiques (ce sont des variables globales visibles seulement localement), ou automatiques (elles sont allouees et desallouees a chaque passage dans le bloc). Certains symboles representent aussi les fonctions, qui sont donc visibles en tant que symboles et en tant que blocs. Pour chaque symbole, il est possible de retrouver son bloc, son espece (variable ou fonction, globale ou locale), et son adresse ou sa position relative dans la pile d'execution. Enn, on peut aussi connaître le type auquel il appartient.

Les types

Enn, on trouve aussi toutes les informations concernant le systeme de types. Pour chaque type, on trouve le nom, la taille et son espece : il existe des types de base, des types \fonction", des types \pointeur", des types \structure". On trouve enn toute la description du type. Pour les pointeurs, on sait quel est le type pointe. Pour les structures, on a le detail des champs, avec leur nom, leur type et leur position relative. Dans le cas des classes C++, on a de plus la liste des classes de base, le detail des champs publics ou prives, et l'ensemble des methodes. Sous certaines conditions, il est possible pour chaque methode de trouver le symbole de la fonction qui l'implemente. Pour cela, il faut connaître le mecanisme d'encodage des noms de methodes. Cet encodage permet d'utiliser le même nom pour plusieurs methodes prenant des arguments de types differents : pour chaque methode, un nom code est fabrique et utilise comme symbole pour la fonction correspondante. L'encodage consiste generalement a ajouter le nom des types des arguments au nom de la methode, mais aucun mecanisme n'est specie dans le langage.

Fait interessant a noter, il n'y a pas un unique systeme de types par programme. Chaque unite de compilation a le sien. En effet, le compilateur ne garde aucune information entre deux compilations. Même dans le cas ou les types proviennent du même chier de declarations inclus dans deux unites de compilation, le compilateur ne le detecte pas. La situation pourrait être differente avec un environnement de programmation sophistique comme celui d'ADA, mais les denitions des langages C,

C++ ou Pascal ne disent rien sur le sujet. Par ailleurs, on peut verier qu'il est legal dans ces langages de donner une denition pour un nom de type dans une unite de compilation, et une autre denition incompatible dans une autre unite. C'est d'ailleurs une cause frequente d'erreurs, pendant la phase de developpement d'un programme : un champ est rajoute a un type, et les deux versions du type sont utilisees dans un même programme. La difference de taille mene vite a une erreur a l'execution. De plus, cette erreur est difcile a interpreter lorsqu'elle se produit.

En ce qui concerne Witness, le choix a ete fait de correler les types provenant des differentes unites de compilation. Nous verrons plus tard que la principale raison est liee a la representation des donnees. Mais cela permet aussi de detecter des erreurs possibles sans même avoir a executer le programme. La correlation effectuee est relativement simple, comme nous allons le voir.

Habituellement, les langages de programmation utilisent l'une des deux techniques suivantes pour determiner l'equivalence de deux types. Certains utilisent l'equivalence par nom : deux types sont equivalents s'ils ont le même nom. D'autres utilisent l'equivalence par structure : deux types sont equivalents s'ils ont le même nom, ou si leurs composantes sont equivalentes. Par exemple, si pA est un type \pointeur sur A" et ptrA un type \pointeur sur A", pA et ptrA sont des types equivalents. Cette regle s'applique de maniere recursive. Si B est une structure contenant deux pA, et C une structure contenant deux ptrA, alors B et C sont equivalents. Il a ete montre que l'equivalence par structure permet des equivalences non souhaitees, et elle est generalement abandonnee. Cependant, nous venons de voir que l'equivalence par nom est insufsante dans le cas de la compilation structuree. Witness utilise donc la notion d'equivalence suivante :

Deux types sont equivalents si :

1. ils ont le même nom, et
2. leurs composantes sont equivalentes.

Il est evident que la relation ainsi denie est une relation d'equivalence. Cette denition permet de garantir la coherence entre deux unites de compilation. Lorsque deux types ont le même nom et ne sont pas equivalents, une erreur est probable.

D.2 L'analyse dynamique

Pour observer le comportement d'un programme et l'evolution de ses donnees, il faut pouvoir l'executer tout en le contrôlant. Les techniques de base sont celles utilisees par les traditionnels \debuggers". Nous allons les passer en revue avant de voir comment elles permettent d'observer un programme.

Manipulation de processus

Tout d'abord, pour suivre l'execution d'un programme, il faut l'executer dans un processus. Des fonctions d'usage courant du systeme d'exploitation permettent de lancer un processus \ls" depuis un processus pere. D'autres fonctions plus speciques sont necessaires pour contrôler l'execution du processus ls. Le systeme Unix regroupe ces fonctions sous l'appel systeme *ptrace*, qui est speciquement destine a la mise au point. Le principe consiste a appeler *ptrace* avec un premier argument qui determine le type de manipulation a effectuer, et le sens des arguments suivants. Certaines de ces fonctions permettent une communication de tres bas niveau entre processus, mais elles ne sont pas adaptees a une utilisation generale. Ces fonctions peuvent être regroupees en deux categories.

Le premier groupe de fonctions concerne l'execution du processus ls. Elles permettent de lancer ou de continuer l'execution. Il est parfois possible de demander l'execution d'une seule instruction en langage machine. L'execution se fait en fonction de l'etat courant du processus, qui peut avoir ete modie par son pere. Par exemple, il est possible de choisir l'endroit ou l'execution va reprendre.

Le second groupe de fonctions permet de manipuler l'etat du processus ls. Elles permettent de lire et de modier la memoire et les registres. C'est de cette maniere qu'il est possible d'examiner et de modier les variables. C'est aussi comme cela qu'on peut interferer avec l'execution, comme nous allons le voir.

Les points d'arrêt

Les fonctions de manipulation du processus ls ne sont utilisables que lorsque ce dernier est arrête et prêt a recevoir des ordres. Pour suivre son execution, il faut donc le forcer a s'arrêter en des points determines a l'avance. Ces points d'arrêt sont realises en modiant le code qui est en train d'être execute. Il faut reperer l'endroit ou l'on veut s'arrêter, recopier les instructions qui s'y trouvent, et les remplacer par des instructions particulieres. Ces instructions \parasites" ont pour effet de forcer l'interruption du processus, comme s'il avait recu un signal. Le processus pere, apres avoir lance l'execution, est reste en attente. Lorsqu'un point d'arrêt est rencontre, il est prevenu et peut de nouveau examiner et manipuler le processus ls.

L'endroit ou placer des points d'arrêt est determine grâce aux informations recueillies dans le chier executable. Par exemple, il est possible de poser un point d'arrêt au debut de chaque fonction. Ainsi, l'execution se deroule comme un dialogue entre le pere et le ls, un peu comme le font des coroutines.

Cependant, il n'est pas toujours souhaitable de s'en tenir au cours normal de l'execution du ls. On veut parfois appeler une fonction particuliere an d'examiner son comportement en detail, par exemple avec une serie de parametres differents. Pour

ce faire, la technique consiste a simuler l'appel de la fonction. Il suft d'allouer de la memoire supplementaire au processus ls, d'y inscrire la sequence d'instructions correspondante, et de lancer l'execution a cet endroit.

L'acces aux donnees

L'ensemble des techniques que nous venons d'evoquer permet d'executer a volonte telle ou telle partie d'un programme, tout en suivant son execution. Voyons maintenant comment on accede aux donnees. Nous avons vu qu'on peut lire et ecrire en tous points du processus ls. Il suft donc de savoir ou le faire. Pour les variables globales, l'adresse est connue grâce aux informations recueillies a la lecture du chier executable. Pour les variables locales, on ne connaît que leur position relative. En effet, il n'est pas possible de determiner a l'avance quand une fonction ou un bloc d'instructions va être execute. Une fonction peut même être active plusieurs fois a un instant donne : c'est la recursivite. Pour permettre cela, les variables locales sont stockees dans un *contexte* cree a l'entree du bloc et detruit a la sortie. Pour acceder aux variables locales, il faut donc trouver le contexte correspondant. En general, l'adresse du contexte actif a un moment donne est disponible dans un registre. Par ailleurs, les contextes sont chaînes : le contexte d'un bloc contient l'adresse du contexte du bloc appelant. On peut ainsi acceder a toutes les valeurs des variables locales des blocs actifs. Notons qu'a un symbole representant une variable locale correspond un nombre de valeurs qui varie selon le moment de l'execution du programme. S'il appartient a une fonction qui n'est pas active, il n'a pas de valeur. Si la fonction a ete appelee et s'est ensuite appelee elle même, il a deux valeurs, jusqu'a la n de l'execution de cette fonction.

Par ailleurs, selon les langages, toutes les donnees n'ont pas de symbole associe. En effet, il est generalement possible de creer dynamiquement des donnees en memoire. En C, le programmeur s'alloue des morceaux de memoire et les utilise a volonte. En C++, il cree des objets ou des tableaux d'objets. Ces donnees dynamiques ne correspondent a aucun symbole. Il arrive même qu'elles ne soient plus referencees d'aucune maniere. Or ces donnees dynamiques representent souvent la grande majorite des donnees d'un programme, en particulier avec un langage a objets comme C++. Il importe qu'elles puissent être representees au même titre que les autres donnees. Il faut donc dans la mesure du possible conserver un moyen d'y acceder, et obtenir leur type. Nous verrons par la suite comment cela a ete realise pour C++ dans Witness.

Les informations et les possibilites que nous venons de decrire sont utilisees dans Witness comme dans un \debugger" classique. Elles sont aussi utilisees pour la representation animee de variables. Nous allons maintenant decrire comment.

L'observation des objets

Dans sa version actuelle, Witness permet de representer les instances de classes C++. Dans ce langage, les classes sont des types comme les autres. Mais nous avons vu que la notion de type est mal denie dans un programme executable. Dans la suite de ce texte, nous appellerons donc \classe" l'ensemble des types denissant une classe C++ dans leurs unites de compilation respectives, et equivalents entre eux. Les objets animes seront les instances de ces classes. Il existe deux raisons a ce choix de la classe et de ses instances comme cible de la representation animee.

La premiere raison est technique. Nous avons vu qu'il est delicat de garder trace des donnees creees dynamiquement. Or les instances d'une classe C++ ont une caracteristique commune, qu'elles correspondent a des donnees globales, locales, ou dynamiques : tous les objets passent par un constructeur et un destructeur. Witness exploite cette caracteristique. Lorsque les objets d'une classe doivent être visualises, un point d'arrêt est place dans chaque constructeur de la classe, et un dans le destructeur. Ces points d'arrêt ne sont normalement pas visibles de l'utilisateur nal. Ils servent a Witness pour gerer la liste des instances d'une classe. Lorsqu'un objet est construit, il est ajoute a la liste. S'il correspond a un symbole, Witness le detecte et le memorise. Lorsqu'un objet est detruit, il est elimine de la liste. En revanche, Witness est incapable de garder trace des objets de classes sans constructeurs et sans destructeurs, et a plus forte raison des objets des types de base, entiers, nombre ottants ou caracteres.

La deuxieme raison pour faire reposer la representation des donnees sur la notion de classe est moins technique. En effet, le but de Witness n'est pas seulement de representer l'etat des donnees. Il est aussi de visualiser les operations sur ces donnees. Les classes de C++ fournissent un cadre ideal pour cela. Les donnees sont les objets, et les operations sur ces donnees sont les methodes de la classe. De la même maniere qu'il est possible de mettre des points d'arrêt dans les constructeurs et destructeur d'une classe, il est possible d'en mettre dans toutes les methodes de cette classe. Lorsque l'une de ces methodes est appelee, Witness le detecte et trouve l'objet pour lequel elle a ete appelee. Il est ainsi possible de visualiser l'operation sur l'objet.

Bibliographie

[Adobe Systems Inc. 85]
Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[Aho et al. 86]
A.V. Aho, R. Sethi, et J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Anderson & Kuivila 90]
David P. Anderson et Ron Kuivila. A system for computer music performance. *ACM Transactions on Computer Systems*, 8(1):56{82, 1990.

[Apple Computer Inc. 85]
Apple Computer Inc. *Inside Macintosh*. Addison-Wesley, 1985.

[Apple Computer Inc. 86]
Apple Computer Inc. *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, 1986.

[Araki et al. 91]
Keijiro Araki, Zengo Furukawa, et Jingde Cheng. A general framework for debugging. *IEEE Software*, pages 14{20, may 1991.

[Baecker & Buxton 87]
Ronald M. Baecker et William A.S. Buxton. *Readings in Human-Computer Interaction*. Morgan-Kaufmann, 1987.

[Baecker 81]
Ronald Baecker. Sorting out sorting. 16 mm color sound lm, 25 mn, 1981.

[Baecker et al. 91]
Ronald Baecker, Ian Small, et Richard Mander. Bringing icons to life. *Proceedings of the ACM CHI'91*, pages 1{6. Addison-Wesley, Mai 1991.

[Barth 86]
Paul S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2), Avril 1986.

[Baskerville 85]
David B. Baskerville. Graphic presentation of data structures in the DBX debugger. Rapport Technique UCB/CSD 86/260, University of California at Berkeley, 1985.

[Bass & Coutaz 91]
Len Bass et Joelle Coutaz. *Developing Software for the User Interface*. The SEI Series in Software Engineering. Addison Wesley, 1991.

[Bass et al. 88]
L. Bass, E. Hardy, K. Hoyt, R. Little, et R. Seacord. The Serpent run time architecture and dialogue model. Rapport Technique SEI-88-TR-6, Carnegie Mellon University, Janvier 1988.

[Baudel 90]
Thomas Baudel. GAG : un generateur d'analyseurs gestuels. Rapport de DEA, Universite de Paris Sud, 1990.

[Beaudouin-Lafon & Karsenty 87]
Michel Beaudouin-Lafon et Solange Karsenty. Graphical debugging in object-oriented environments. Internal Report 357, LRI, Universite de Paris-Sud, France, Juin 1987.

[Beaudouin-Lafon & Karsenty 91]
Michel Beaudouin-Lafon et Alain Karsenty. Transparency and awareness in a real-time groupware system. Rapport Technique 704, LRI, 1991.

[Beaudouin-Lafon 88]
Michel Beaudouin-Lafon. User interface support for the integration of software tools: an iconic model of interaction. *Proceedings of the ACM Symposium on Software, Boston*, pages 187{196, 1988.

[Beaudouin-Lafon 91]
Michel Beaudouin-Lafon. User interface management systems: Present and future. *Eurographics'91*, Septembre 1991.

[Beaudouin-Lafon et al. 90]
Michel Beaudouin-Lafon, Yves Berteaud, et Stephane Chatty. Creer des applications a manipulation directe avec X_{TV}. *IHM'90, Biarritz, France*, 1990.

[Beaudouin-Lafon et al. 91]
Michel Beaudouin-Lafon, Yves Berteaud, Stephane Chatty, Thomas Baudel, et Jean-Daniel Fekete. X_{TV} version 2.0 programmer's manual. Rapport technique, Laboratoire de Recherche en Informatique, 1991.

[Berry et al. 87]
Gerard Berry, Philippe Couronne, et Georges Gonthier. Programmation synchrone des systemes reactifs : le langage ESTEREL. *Technique et Science Informatique*, 6(4):305{316, 1987.

[Bertin 73]
Jacques Bertin. *Semiologie Graphique*. Mouton-Gauthier-Villard-Bordas, 1973.

[Bono et al. 82]
Peter R. Bono, Jose L. Encarnacao, F. Robert A. Hopgood, et Paul J. W. ten Hagen. GKS { the rst graphics standard. *IEEE Computer Graphics and Applications*, 1982.

[Borning & Duisberg 86]
Alan Borning et Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on graphics*, 5(4):345{374, 1986.

[Borning et al. 87]
A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, et M. Woolf. Constraint hierarchies. *Proceedings of the ACM OOPSLA*, pages 48{60, Octobre 1987.

[Brown & Sedgewick 84]
Marc H. Brown et Robert Sedgewick. A system for algorithm animation. *Computer Graphics: SIGGRAPH '84*, 18(3):177{186, 1984.

[Brown 88]
Marc H. Brown. *Algorithm Animation*. ACM Distinguished Dissertations. The MIT Press, 1988.

[Bruegge & Hibbard 83]
Bernt Bruegge et Peter Hibbard. Generalized path expressions: A high-level debugging mechanism. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Mars 1983.

[Card et al. 83]
Stuart K. Card, Thomas P. Moran, et Alan Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[Cardelli 87]
Luca Cardelli. Building user interfaces by direct manipulation. Rapport Technique #22, Digital Systems Research Center, 1987.

[Chadabe & Neyers 78]
J. Chadabe et R. Neyers. An introduction to the Play program. *Computer Music*, 2(1), 1978.

[Chatty 88]
Stephane Chatty. Une maquette de debugger graphique. Rapport de DEA, Universite de Paris Sud, 1988.

[Cointe & Rodet 84]
Pierre Cointe et Xavier Rodet. Formes: an object and time oriented system for music composition and synthesis. *Proceedings of the ACM Conference on Lisp and Functional Languages*, 1984.

[Cournarie & Beaudouin-Lafon 91]
Eric Cournarie et Michel Beaudouin-Lafon. Alien: A prototype-based constraint system. *Second Eurographics Workshop on Object Oriented Graphics*, pages 93{114, Juin 1991.

[Coutaz 87]
Joelle Coutaz. PAC, an implementation model for dialog design. *Proceedings of the Interact'87 Conference*, pages 431{436. North Holland, Septembre 1987.

[Coutaz 90]
Joelle Coutaz. *Interfaces homme-ordinateur. Conception et realisation.* Informatique. Dunod, 1990.

[Dannenberg 84]
Roger B. Dannenberg. Arctic: A functional language for real-time control. *Proceedings of the ACM Conference on Lisp and Functional Languages*, pages 96{103, 1984.

[Ducasse & Emde 89]
M. Ducasse et A.-M. Emde. A survey of automated bug location. Rapport Technique LP-31-23, ECRC, Arabellastr. 17. D-8000 Munich 81, Decembre 1989.

[Duisberg 86]
Robert A. Duisberg. Animated graphical interfaces using temporal constraints. *Proceedings of the ACM CHI'86*, pages 131{136, 1986.

[Edmonds 81]
E. A. Edmonds. Adaptive man-computer interfaces. M. J. Coombs et J. L. Alty, editors, *Computing Skills and the User Interface*. Academic-Press, 1981.

[Fantôme 91]
Fantôme. Les fables geometriques. Video-cassette, Canal+ Video, 1991.

[Foley et al. 90]
James D. Foley, Andires van Dam, S. K. Feiner, et J. F. Hugues. *Fundamentals of Interactive Computer Graphics, 2nd edition.* Addison-Wesley, 1990.

[Froidevaux et al. 90]
Christine Froidevaux, Marie-Claude Gaudel, et Michele Soria. *Algorithmes et Types de Donnees*. Mac Graw-Hill, 1990.

[Fry 91]
C. Fry. Flavors band: A language for specifying musical style. Stephen Travis Pope, editor, *The Well-Tempered Object*. The MIT Press, 1991.

[Gascuel & Puech 91]
Marie-Paule Gascuel et Claude Puech. Dynamic animation and deformable bodies. *Eurographics'91*, Septembre 1991.

[Gaver 89]
William W. Gaver. The SonicFinder: an interface that uses auditory icons. *Human-Computer Interaction*, 4(1), 1989.

[Gaver 91]
William W. Gaver. Technology affordances. *Proceedings of the ACM CHI'91*, pages 79{84. Addison-Wesley, Mai 1991.

[Goldberg 84]
Adele Goldberg. *SMALLTALK-80, the Interactive Programming Environment*. Addison-Wesley, 1984.

[Harel 87]
D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231{274, Juin 1987.

[Hartson & Hix 89]
H. Rex Hartson et Deborah Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21:5{92, 1989.

[Hils 91]
Daniel D. Hils. Data ow visual programming languages. *Journal of Visual Languages and Computing*, Decembre 1991.

[Hoare 78]
C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666{677, 1978.

[Hullot 86]
Jean-Marie Hullot. SOS Interface, un Generateur d'interfaces homme-machine. *Actes des Journees Langages Orientes Objet*, pages 69{78. AFCET Informatique, 1986.

[Ingalls et al. 88]
Dan Ingalls, Scott Wallace, Yu-Ying Chow, Franck Ludolph, et Ken Doyle. Fabrik: A visual programming environment. *OOPSLA'88 Proceedings*, pages 176{190, Septembre 1988.

[ISO 86]
ISO. Programmer's hierarchical interface to graphics functional description. Rapport Technique ISO DP9592, International Standards Organisation, 1986.

[Jaffe & Boynton 91]
David Jaffe et Lee Boynton. An overview of the Sound and Music Kits for the NeXT computer. Stephen Travis Pope, editor, *The Well-Tempered Object*. The MIT Press, 1991.

[Karsenty 87]
Solange Karsenty. *Grafti: un Outil Interactif et Graphique pour la Construction d'Interfaces Homme-machine Adaptables*. These de Doctorat, Universite de Paris-Sud, France, 1987.

[Kasik 82]
D. J. Kasik. A user interface management system. *ACM Computer Graphics*, 16(3):99{105, Juillet 1982.

[Kleyn & Chatravarty 89]
Michael F. Kleyn et Indranil Chatravarty. EDGE { a graph based tool for specifying interaction. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, pages 1{14, Octobre 1989.

[Knuth 84]
Donald E. Knuth. *The TEX Book*. Addison-Wesley, 1984.

[Levy 91]
Pierre Levy. *L'ideographie dynamique*. Le Concept Moderne, Geneve, 1991.

[Lieberman 86]
Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *Proceedings of the ACM OOPSLA*, pages 214{223, Septembre 1986.

[Linton & Vlissides 87]
Marc A. Linton et John M. Vlissides. The design and implementation of InterViews. *Proceedings of the USENIX C++ Workshop*, Novembre 1987.

[Linton et al. 89]
Marc A. Linton, John M. Vlissides, et Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8{22, Fevrier 1989.

[Loy & Abbott 85]
Gareth Loy et Curtis Abbott. Programming languages for computer music synthesis, performance and composition. *ACM Computing Surveys*, 17(2):235{265, 1985.

[Magnenat-Thalmann & Thalmann 85]
Nadia Magnenat-Thalmann et Daniel Thalmann. *Computer Animation: Theory and Practice.* Springer, 1985.

[Maloney et al. 89]
John H. Maloney, Alan Borning, et Bjorn N. Freeman-Benson. Constraint technology for user-interface construction in ThingLab II. *OOPSLA'89 Proceedings*, pages 381{388, Octobre 1989.

[Masini & al. 89]
Gerald Masini et al. *Les Langages a Objets*. InterEditions, 1989.

[Meyer 90]
Bertrand Meyer. *Conception et Programmation par Objets*. InterEditions, 1990.

[Moher 88]
Thomas G. Moher. Provide: A process visualisation and debugging environment. *IEEE Transactions on Software Engineering*, 14:849{857, 1988.

[Muller et al. 90a]
Heinrich Muller, Jorg Winckler, Stefan Grzybek, Matthias Otte, Bertram Stoll, Frederic Equoy, et Nicolas Higelin. The program animation system PASTIS. Rapport technique, Universitat Freiburg, Institut fur Informatik, 1990.

[Muller et al. 90b]
Heinrich Muller, Jorg Winckler, Stefan Grzybek, Matthias Otte, Bertram Stoll, Frederic Equoy, et Nicolas Higelin. PASTIS - program animation using X. *The European X Window System Conference (EX'90)*, pages 104{111, 1990.

[Myers 83]
Brad Myers. Incense: A system for displaying data structure. *Computer Graphics: SIGGRAPH '83*, 17:115{125, 1983.

[Myers 89a]
Brad Myers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15{23, 1989.

[Myers 89b]
Brad A. Myers. Encapsulating interactive behaviors. *Proceedings of the ACM SIGCHI*, pages 319{324, Mai 1989.

[Myers et al. 90]
Brad A. Myers et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71{85, Novembre 1990.

[Olsen & Dempsey 83]
Dan R. Olsen, Jr et Elizabeth P. Dempsey. Syngraph: A graphical user interface generator. *ACM Computer Graphics*, pages 43{50, Juillet 1983.

[Olsen 86]
Dan R. Olsen, Jr. MIKE: the menu interaction kontrol environment. *ACM Transactions on Graphics*, 5(4):318{344, Octobre 1986.

[Olsson et al. 91]
Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, et Christopher E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, pages 27{36, may 1991.

[Pfaff 85]
Gunther E. Pfaff, editor. *User Interface Management Systems*. Eurographics Seminars. Springer-Verlag, 1985.

[Reiss 86]
Steven. P. Reiss. Visual languages and the GARDEN system. *Visualization in Programming*, number 282 in Lecture Notes in Computer Science, pages 178{192. Springer-Verlag, Mai 1986.

[Reiss 87a]
Steven P. Reiss. Displaying programs and data structures. *Proceedings of the 20th HICSS*, 1987.

[Reiss 87b]
Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, pages 16{27, Novembre 1987.

[Reiss 88]
Steven P. Reiss. Integration mechanisms in the FIELD environment. Rapport Technique CS 88-18, Brown university, Octobre 1988.

[Robertson et al. 91]
George G. Robertson, Jock D. Mackinlay, et Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. *Proceedings of the ACM CHI'91*, pages 189{194. Addison-Wesley, Mai 1991.

[Rogers 85]
David F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, 1985.

[Scheier & Gettys 86]
Robert W. Scheier et Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79{109, 1986.

[Shmucker 86]
K. J. Shmucker. MacApp: An application framework. *Byte Magazine*, pages 189{192, Août 1986.

[Shneiderman 83]
Ben Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, pages 57{69, Août 1983.

[Shneiderman 87]
Ben Shneiderman. *Designing the user-interface. Strategies for Effective Human-Computer Interaction*. Addison Wesley, 1987.

[Singh & Green 91]
Gurminder Singh et Mark Green. Automating the lexical and syntactical design of graphic user interfaces: The UofA* UIMS. *ACM Transactions on Graphics*, 10(3), Juillet 1991.

[Stallman 89]
Richard M. Stallman. GDB user's manual. Rapport technique, Free Software Foundation, 1989.

[Stasko 89]
John T. Stasko. *TANGO: A Framework and System for Algorithm Animation*. These de Doctorat, Brown University, 1989.

[Tanner & Buxton 83]
Peter P. Tanner et William A.S. Buxton. Some issues in future interface management systems. *Proceedings of the IFIP WG 5.2 Workshop on UIMS*, Novembre 1983.

[Thalmann 90]
Daniel Thalmann. *Scientic Visualization and Graphics Simulation*. Wiley, 1990.

[UID91]
The Arch model: Seeheim revisited. Presented at the ACM SIGCHI, Avril 1991.

[UID92]
A metamodel for the runtime architecture of an interactive system. A paraître dans ACM SIGCHI Bulletin, 1992.

[Ungar & Smith 87]
David Ungar et Randall B. Smith. Self: The power of simplicity. *Proceedings of the ACM OOPSLA*, pages 227{241, Octobre 1987.

[Vercoe 86]
Barry Vercoe. Csound. a manual for the audio processing system and supporting programs. Rapport technique, MIT Media Lab, 1986.

[Vlissides & Linton 88]
John M. Vlissides et Mark. A. Linton. Applying object-oriented design to structured graphics. *Proceedings of the Usenix C++ Conference*, Octobre 1988.

[Wasserman 85]
A. I. Wasserman. Extending state transition diagrams for the specication of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11:699{713, Août 1985.

[Weinand et al. 88]
Andre Weinand, Eric Gamma, et Rudolf Marty. ET++ { an object-oriented application framework in C++. *OOPSLA'88 Proceedings*, Septembre 1988.

[Weinand et al. 89]
Andre Weinand, Eric Gamma, et Rudolf Marty. Design and implementation of ET++, a seamless object-oriented framework. *Structured Programming*, 10(2), 1989.

[Weiser 91]
Mark Weiser. Les reseaux de l'an 2000. *Pour la Science*, pages 64{72, Novembre 1991.

[Wellner 89]
Pierre D. Wellner. Statemaster: A UIMS based on Statecharts for prototyping and target application. *Proceedings of the ACM SIGCHI*, pages 177{182, Mai 1989.

[Woods 70]
W.A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, pages 591{606, Octobre 1970.

[Zeleznik & al. 91]
Robert C. Zeleznik et al. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics*, 25(4):105{113, Juillet 1991.

Index