Fondements de l'Interaction Homme-Machine

3 - Outils de développement

Michel Beaudouin-Lafon, mbl@lri.fr

Introduction

Les outils de développement d'applications graphiques interactives se classent en plusieurs catégories selon le niveau d'abstraction qu'ils fournissent à l'utilisateur, chaque niveau utilisant les services du niveau en dessous.

Au niveau le plus bas on trouve les *systèmes graphiques* qui fournissent les services élémentaires de dessin et de gestion des entrées. Au-dessus on trouve les *systèmes de fenêtrage* qui permettent de séparer l'écran en zones indépendantes (les fenêtres) afin que plusieurs applications puissent se partager les périphériques d'entrée et de sortie. Dans de nombreux cas, le système et de fenêtrage et le systèmes graphique sont intégrés dans un seul module logiciel.

Les *boîtes à outils* fournissent au programmeur un niveau d'abstraction supérieur, celui d'objet interactif ou "widget". La plupart des applications actuelles sont développées avec des boîtes à outils d'interface (ou "toolkits"). Pour faciliter la programmation, il existe des *générateurs d'interface* qui permettent de construire une partie de l'application de façon interactive, en "dessinant" l'interface.

Enfin, les *UIMS* et les *UIDE* sont des environnements de développement d'applications interactives qui fournissent un ensemble d'outils plus ou moins intégrés.

Librairies graphiques

Une librairie graphique fournit un ensemble de fonctions permettant de dessiner des formes graphiques et de gérer les entrées de l'utilisateur. La gestion des entrées a déjà été décrite dans le chapitre précédent avec les modes requête, échantillonage et événement. Dans la suite, nous considérons uniquement la gestion par événement, qui est la plus répandue à l'heure actuelle.

Formes graphiques

L'affichage d'une forme graphique nécessite de définir ses caractéristiques. Celles-ci sont d'une part la forme géométrique et d'autres part les attributs graphiques de rendu. Toute librairie graphique fournit un ensemble de formes prédéfinies. En 2 dimensions, on trouve généralement les formes suivantes :

- segments de droites, polylignes, polygones ;
- rectangles, ellipses et arcs d'ellipses ;
- courbes de Bezier, splines ;

- chaînes de caractères.

Une forme est donc définie par son type et par ses caractéristiques géométriques, qui sont un ensemble de points, de dimensions et/ou de vecteurs.

En 3 dimensions, ou utilise le plus souvent des formes définies par un ensemble de facettes ou par l'association booléenne de formes élémentaires (cube, sphère, etc.) auxquelles on applique des déformations. Dans la suite nous nous limitons aux formes à 2 dimensions.

La seule géométrie d'un objet ne suffit pas pour le dessiner. Il faut lui donner des attributs graphiques. Selon les librairies graphiques, la palette des attributs disponibles varie énormément. Plus le modèle graphique est sophistiqué, plus la palette d'attributs est grande. La palette minimale est la suivante :

- épaisseur des traits ;
- couleur et motif des traits ;
- couleur de motif remplissage ;
- police de caractères ;
- mode de transfert.

Les couleurs peuvent être définies à partir d'une base de données de couleur, ou bien en spécifiant directement une valeur de couleur pour les systèmes sans table de couleur ou sinon un indice dans la table de couleurs. Un motif de coloriage (aussi appelé texture) est une image qui est répétée plusieurs fois sur la surface à dessiner, comme du papier peint.

Une police de caractères est un ensemble de formes géométriques qui définissent chacun des caractères de l'alphabet. Ces formes sont soit des images (on parle alors de police bitmap), soit des formes de type spline. Les polices bitmap on l'avantage d'un affichage rapide, mais l'inconvénient de ne représenter qu'une seule taille de caractères. A l'inverse les polices définies par des splines sont plus lentes à afficher car il faut calculer le contour de chaque caractère à partir de sa forme, puis le remplir, mais elles permettent d'afficher des caractères de n'importe quelle taille. Cependant, les résultats ne sont pas toujours satisfaisant pour des petites tailles à cause des problèmes d'aliassage, c'est-à-dire l'approximation due à la digitalisation des formes sur la grille que forme les pixels. Lorsque l'on utilise des polices définies par des splines, le système graphique conserve dans un cache les images des caractères les plus utilisés afin d'augmenter les performances de l'affichage.

Le mode de transfert définit comment la forme est dessinée par rapport à ce qui est déjà à l'écran. Dans le mode le plus courant, REPLACE, la forme est dessinée au-dessus de ce qui est déjà à l'écran et masque donc ce qui est en-dessous. Un mode souvent utilisé pour la réalisation d'interfaces est le mode XOR. En effet ce mode permet d'effacer facilement une forme en la redessinant une seconde fois à la même position. Cette technique est très utilisée pour le feed-back lorsque l'on déplace un objet à la souris (ligne ou rectangle élastique, déplacement d'un cadre de fenêtre ou d'un icone, etc.). En effet il faut assurer que le réaffichage soit suffisamment rapide pour suivre les déplacements du pointeur. Il est généralement beaucoup plus rapide d'effacer la forme représentant le feed-back en la redessinant à nouveau plutôt que de reconstituer la partie qu'elle cachait car cela suppose de connaître tous les objets qu'elle occulte et de les redessiner. La technique de dessin en XOR est détaillée plus loin dans cette section.

Les librairies graphiques offrent en général une palette d'attributs plus grande que celle décrite ci-dessus. Par exemple, il est souvent possible de contrôler la forme des extrémités des formes ouvertes et des jonctions des polygones et polylignes : on peut avoir des extrémités et jonctions arrondies, carrées ou avec chanfrein. Des

modes de transfert spéciaux permettent d'obtenir des effets de transparence, des masques permettent de dessiner dans une partie spécifique de l'écran, etc.

Dessin en XOR

En utilisant le mode de transfert XOR, l'affichage d'un pixel de valeur *f* de la forme sur un pixel de valeur *p* du fond produit un pixel de valeur *p'* telle que

```
p' = f xor p
```

Si l'on redessine la forme une deuxième fois à la même position avec le même mode de transfert XOR, on efface la forme, c'est-à-dire que l'on retrouve l'écran dans l'état dans lequel il était avant le premier affichage de la forme. En effet, pour chaque pixel de valeur *f* de la forme, le pixel affiché a pour valeur *p'' = p* :

```
p'' = f xor p' = f xor (f xor p) = (f xor f) xor p
comme f xor f = 0 et 0 xor p = p on obtient p'' = p
```

Si l'on a un écran noir et blanc pour lequel 1 = noir et 0 = blanc et que l'on affiche une forme en noir, chaque pixel appartenant à la forme est inversé à l'écran puisque 1 xor p = not p. Donc si l'on dessine la forme deux fois, chaque pixel est inversé deux fois et revient donc dans son état initial.

Par contre, sur un écran couleur, les résultats sont moins prévisibles. Supposons que l'on utilise une table de couleurs et que l'on ait 4 bits par pixel. Si le noir est représenté par la valeur de pixel 1111 et que l'on dessine en xor sur un pixel de valeur 1001, le résultat est un pixel de valeur 1111 xor 1001 = 0110. La couleur résultante est alors imprévisible : on obtient un effet "technicolor".

Il existe plusieurs façons de contrôler les couleurs effectivement affichées. La plus courante consiste à privilégier la couleur affectée au fond de l'écran en considérant que les formes affichées ont une intersection importante avec les parties où le fond est visible. Supposons que la valeur de pixel utilisée pour le fond d'écran est *b*. Si l'on veut que la forme affichée en XOR apparaissent avec la couleur *c* sur le fond d'écran, il suffit de dessiner la forme en XOR avec la couleur *(c xor b)*. En effet, lorsque l'on affichera sur le fond de couleur b, on obtiendra la couleur

```
p = (c xor b) xor b = c xor (b xor b) = c
```

Par contre les parties de la forme qui sont affichées sur des pixels de valeurs différentes de b produisent toujours des couleurs imprévisibles.

Si l'on peut se permettre de gaspiller la moitié de la table des couleurs, on peut complétement contrôler les couleurs affichées avec le mode XOR. Il suffit de remarquer que si l'on dessine en XOR une forme avec la valeur de pixel 1111, tout pixel de valeur *p* est transformé en un pixel de valeur *not p*. En sépare donc les entrées de la table de couleur en deux ensembles D et F tels que

pour toute valeur d de D, not d est dans F
pour toute valeur f de F, not f est dans D

D est l'ensemble des valeurs de pixels utilisées pour dessiner des formes en mode normal. F est l'ensemble des couleurs qui seront utilisées lorsque l'on dessinera une forme en XOR avec la valeur de pixel 1111. Comme à chaque valeur de pixel de D correspond exactement une valeur de pixel de F, on peut contrôler précisément l'effet du XOR. Par exemple, on peut affecter la même couleur (par exemple le noir) à toutes les valeurs de pixel de F. Dans ce cas les formes dessinées en XOR apparaîtront dans cette couleur.

Il faut noter que si l'on n'utilise pas de table de couleurs, aucune de ces méthodes ne s'applique et l'on ne peut échapper à l'effet technicolor. Cependant, il est

fréquent que ce type de matériel offre des plans de dessin supplémentaires qui permettent de dessiner des formes indépendantes du contenu de l'écran.

Systèmes avec ou sans état

Il existe deux catégories de systèmes graphiques : ceux qui conservent une mémoire des formes dessinées à l'écran et ceux qui ne conservent pas de mémoire du contenu de l'écran autre que l'écran lui-même. On parle dans le premier cas de systèmes à état ou à structure d'affichage, dans le second de systèmes sans état.

Dans un système sans état, les fonctions disponibles pour le programmeur sont des opérations de dessin, paramétrées par les formes à dessiner et les attributs à utiliser. Chaque dessin se fait "par-dessus" le contenu courant de l'écran. Le système ne garde pas de trace du contenu de l'écran autre que le buffer d'écran lui-même.

Dans un système à état, les fonctions disponibles pour le programmeur lui permettent d'accéder et de modifier une structure de données appelée *structure d'affichage*. Le système graphique se charge de maintenir la cohérence entre la structure d'affichage et le contenu de l'écran. Il est par exemple possible d'insérer dans la structure d'affichage un objet graphique défini par sa forme et ses attributs, puis de modifier plus tard l'un de ces attributs. Le système graphique ayant gardé en mémoire la structure d'affichage, il pourra mettre à jour le contenu de l'écran pour prendre en compte les modifications.

Les structures d'affichages les plus courantes sont des listes, des arbres et des DAG (Direct Acyclic Graphs, ou graphes sans cycle, ou arbres avec partage). Dans le cas d'une liste, chaque élément décrit un objet graphique avec sa forme et ses attributs. L'ordre de la liste détermine l'ordre de superposition des objets graphiques. Dans un arbre ou un DAG, chaque nœud contient des attributs et parfois des transformations géométriques qui s'appliquent à tout le sous-arbre, et chaque feuille est un objet graphique. Comme pour la liste, l'ordre des fils d'un nœud définit l'ordre de superposition des objets qu'il contient. Avec cette structure, il est possible de modifier un attribut d'un nœud pour affecter un grand nombre d'objets graphiques. Dans le cas du DAG, on peut partager des sous-arbres et donc des formes graphiques complexes, ce qui économise de la mémoire. De plus une modification au niveau d'un sous-arbre partagé se propage à l'ensemble des nœuds qui incluent ce sous-arbre.

Les systèmes graphiques à structure d'affichage sont capables de mettre à jour l'écran de façon optimisée lorsque la structure d'affichage est modifiée. Ils permettent également de faire du "picking", c'est-à-dire de déterminer l'objet qui est désigné par une position (x, y) de l'écran, ce qui est très utile dans le cas des interfaces graphiques.

La plupart des systèmes graphiques utilisés pour les interfaces graphiques 2D sont des systèmes sans état : Xlib (X Windows), QuickDraw (Macintosh), Win32 (Windows), Postscript (NeXT). Les systèmes à structure d'affichage sont le plus souvent destiné au 3D : GKS, PHIGS, GL, OpenGL, QuickDraw3D. La raison principale de cette différence est que le nombre de formes et d'attributs en 3D devient très important et que des accélérateurs matériels existent pour optimiser l'affichage d'objets 3D. A l'inverse, la structure d'affichage peut s'avérer inutilement lourde en 2D, et bien souvent l'application peut reconstituer le contenu de l'écran beaucoup plus efficacement que le système graphique car elle a une meilleure connaissance de la structure de l'affichage (par exemple les parties qui se recouvrent).

Il faut noter qu'il est toujours possible de construire un système graphique à structure d'affichage à partir d'un système sans état. Il s'agit en réalité de deux niveaux d'abstraction différents.

Systèmes de fenêtrage

La notion de fenêtre a été inventé à Xerox PARC lors des travaux qui ont conduit à la première station de travail graphique, le Star. Le principe des fenêtres est de fournir aux applications une abstraction de la notion de zone d'affichage. Sans fenêtre, une application a seulement accès à l'écran dans son ensemble, et il très difficile à plusieurs applications de se coordonner pour se partager l'écran.

Un système de fenêtrage fournit au programmeur un ensemble de fonctions qui permettent à une application de créer, modifier et détruire des fenêtres, de dessiner dans des fenêtres et de récupérer les entrées qui se produisent dans une fenêtre.

Modèles de fenêtrage

Les fenêtres sont généralement de forme rectangulaire. Certains systèmes permettent de gérer des fenêtres de forme quelconque. Les fenêtres peuvent se chevaucher : dans ce cas une fenêtre cache une partie (ou la totalité) de l'autre. Un ordre de superposition (ou rang) permet d'ordonner totalement les fenêtres et de déterminer, lorsque deux fenêtres se chevauchent, laquelle masque l'autre.

Dans le modèle le plus simple, l'écran contient un ensemble de fenêtres. Chaque application peut créer une ou plusieurs fenêtres. La création et la destruction d'une fenêtre sont découplées de son apparition à l'écran : après création, la fenêtre doit être rendue visible explicitement. Elle peut être cachée sans pour autant être détruite. La création d'une fenêtre est souvent une opération coûteuse : si une fenêtre (par exemple une boîte de dialogue) doit apparaître et disparaître souvent, il est préférable de la créer une fois pour toute au début de l'application plutôt que de la créer et de la détruire à chaque fois. Les opérations de base dans ce modèle de fenêtrage sont les suivantes :

- Créer une fenêtre : fournir sa géométrie (position et taille) et ses attibuts : couleur de fond, épaisseur du cadre, titre, rang, etc. ;
- Afficher / cacher une fenêtre ;
- Détruire une fenêtre ;
- Changer la géométrie ou les attributs d'une fenêtre : position, taille, couleur, rang, etc.

Une extension de ce modèle est le modèle hiérarchique. Dans ce modèle, chaque fenêtre peut contenir des (sous-)fenêtres. L'écran lui-même est considéré comme une fenêtre (dans laquelle on peut dessiner, comme les autres fenêtres). L'écran est la racine de l'arbre de fenêtres, il est créé à l'initialisation par le système de fenêtrage et il ne peut être détruit. Chaque fenêtre n'est visible que dans la partie visible de sa fenêtre parente. On a donc une inclusion géométrique des sous-fenêtres dans leur fenêtre parente (figure ci-dessous). Les coordonnées d'une fenêtre sont relatives à la fenêtre parente, de telle sorte que si l'on déplace la fenêtre parente, le sous-arbre suit.

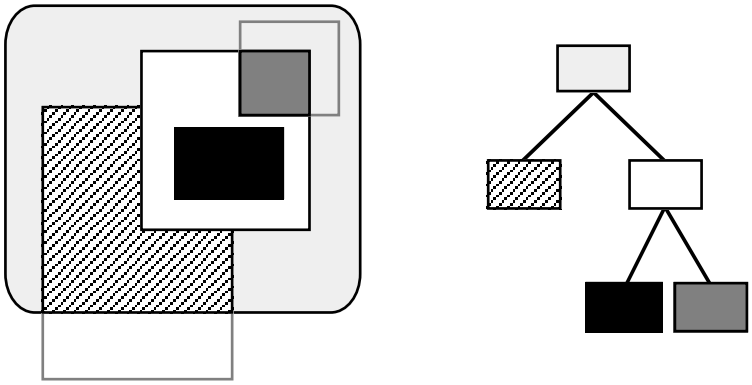

figure : fenêtres hiérarchique et arbre correspondant.

Les opérations dans le modèle hiérarchique sont les mêmes que dans le modèle de base, à l'exception de la création de fenêtre, pour laquelle il faut spécifier la fenêtre parente. Certains systèmes hiérarchique permettent de reparenter une fenêtre, c'est-à-dire de la déplacer (avec son sous-arbre) vers un autre parent.

Affichage et réaffichage des fenêtres

Le dessin d'une forme dans une fenêtre est limité par le système de fenêtrage à la partie visible de la fenêtre. Dans le cas du système hiérarchique, le dessin s'effectue par défaut "derrière" les sous-fenêtres, c'est-à-dire que la partie visible d'une fenêtre n'inclut pas les parties visibles de ses sous-fenêtres. Il est parfois possible de dessiner "par-dessus" les sous-fenêtres d'une fenêtre.

La partie visible d'une fenêtre est susceptible de changer au cours du temps. Par exemple, lorsque l'on change l'ordre de superposition entre deux fenêtres qui se recouvrent, une partie de la fenêtre qui était dessous devient visible. Le système de fenêtrage doit donc s'assurer que le contenu de cette fenêtre est réaffiché, c'est-à-dire qu'il doit mettre la fenêtre dans l'état où elle aurait été si elle n'avait pas été occultée. Si le système graphique sous-jacent est un système à structure d'affichage, le réaffichage de la fenêtre peut être pris en charge directement par le système graphique. Mais la plupart du temps, le système graphique est sans état. Deux solutions sont alors possibles : ou bien le système de fenêtrage mémorise le contenu des parties cachées pour gérer le réaffichage lui-même, ou bien il demande à l'application d'effectuer le réaffichage elle-même, c'est-à-dire qu'il se décharge de la tâche de réaffichage.

Pour mémoriser le contenu des parties cachées, on peut créer une copie de chaque fenêtre dans la mémoire hors-écran. Chaque opération de dessin doit avoir lieu dans la copie hors-écran et dans la partie visible de la fenêtre à l'écran. Lorsqu'une partie cachée devient visible, il suffit d'aller la cherche dans la copie hors-écran. Malheureusement cette méthode est très coûteuse en temps et en espace mémoire : en temps car il faut effectuer chaque opération de dessin deux fois, en mémoire car il faut créer un nombre potentiellement grand de copies hors-écran. Même si l'on économise la mémoire en ne stockant hors-écran que le contenu des parties cachées (et non pas les fenêtres entières), la taille mémoire nécessaire est trop importante.

Aussi la plupart des systèmes de fenêtrage optent pour la deuxième solution : demander à l'application de redessiner les parties cachées lorsqu'elles deviennent visibles. Cette solution impose à l'application de mémoriser l'information nécessaire pour reconstituer l'affichage de n'importe quelle partie de fenêtre, c'est-à-dire d'implémenter à sa manière une structure d'affichage comparable à celle d'un système graphique à état...

La demande de réaffichage peut se faire de plusieurs manières. Lorsqu'elle crée une fenêtre, l'application peut fournir au système de fenêtrage l'adresse d'une fonction à appeler pour réafficher une partie de la fenêtre. Ce type de fonction s'appelle une *fonction de rappel* ou "callback". Une autre alternative est d'utiliser le mécanisme des événements : le système de fenêtrage envoie un événement de type "Réaffichage" à l'application. C'est cette solution qui est utilisée dans la plupart des systèmes de fenêtrage actuels. En fait, comme on va le voir, le sytème de fenêtrage est une source d'événements au même titre que le clavier ou la souris.

Extension du modèle d'événements

Le système de fenêtrage récupère les événements d'entrée du système graphique. Il doit les redistribuer aux applications concernées : de même qu'une application peut dessiner dans ses fenêtres sans s'occuper des autres fenêtres présentes à l'écran ni des autres applications, elle ne doit recevoir que les événements qui concernent une de ses fenêtres.

Notion de focus

Pour permettre ce démultiplexage des événements vers les différentes applications, le système de fenêtrage gère un *focus* pour chaque périphérique d'entrée. Le focus permet de déterminer, pour chaque événement produit par ce périphérique, la fenêtre qui recevra cet événement. Connaissant la fenêtre, le système de fenêtrage peut diriger l'événement vers l'application qui en est propriétaires. Dans le cas le plus général, le focus peut être spécifié selon l'un des modes suivants :

- *explicite* : tous les événements sont envoyés à la même fenêtre, spécifiée explicitement par l'application.
- *implicite* : les événements sont envoyés à la fenêtre qui est à la position (x,y) contenue dans l'événement. Ce mode ne s'applique qu'aux périphériques de localisation, car seuls ceux-ci transmettent une position (x, y).
- *esclave* : le focus de ce périphérique suit celui d'un autre périphérique, spécifié par l'application.

Une configuration courante des modes de focus consiste à mettre la souris en mode explicite et le clavier en mode esclave sur la souris. Les événements clavier et souris sont alors envoyés à la fenêtre "sous" la souris. Une autre configuration consiste à mettre le clavier et la souris en mode explicite sur une fenêtre donnée, dite fenêtre active, un clic sur une fenêtre inactive la rend à son tour active.

Pour implémenter le mode implicite, le système de fenêtrage recherche, en parcourant l'arbre (ou la liste) des fenêtres, la fenêtre qui est la plus profonde et qui contient la position (x, y). Cette fenêtre est dite *cible* de l'événement. Dans le mode explicite, la fenêtre cible est la fenêtre qui reçoit tous les événements ; dans le mode esclave, c'est la fenêtre cible du périphérique maître. Avant que l'événement ne soit transmis à l'applications, la fenêtre cible est stockée dans l'événement et les coordonnées (x, y) sont transformées pour être relatives à la fenêtre cible. Ainsi, l'application reçoit un événement relatif à un fenêtre donnée, et peut interpréter ses coordonnées directement dans le repère de cette fenêtre.

Filtrage des événements

Dans certains systèmes de fenêtrage, il est possible de filtrer les événements afin de ne transmettre à une application qu'un sous ensemble des événements qui lui sont destinés. Par exemple si une application n'utilise pas la souris il est inutile de lui transmettre les événements qui concernent la souris.

Le filtrage est surtout utile dans les systèmes implémentés selon le schéma client-serveur, car il réduit le trafic entre le serveur d'affichage et l'application cliente. En particulier, les événements de déplacement de la souris peuvent sucharger l'application de façon inutile si elle ne les traite pas. Il est souvent possible de distinguer les événements de déplacement de la souris avec ou sans bouton enfoncé. De cette façon, une application peut demander à ne recevoir que les événements correspondant à des actions de type drag-and-drop.

Evénements du systèmes de fenêtrage

Le système de fenêtrage peut produire de nouveaux types d'événements, c'est-à-dire générer des événements en plus de ceux qu'il retransmet du système graphique. Ainsi, le système de fenêtrage synthétise souvent des événements correspondant à l'entrée et à la sortie du curseur dans une fenêtre ("Enter" et "Leave"). Cela permet à une application de suivre la position de la souris dans sa hiérarchie de fenêtres sans avoir à traiter les événements de déplacement de la souris.

Nous avons également vu que le système de fenêtrage peut utiliser des événements pour demander à l'application de réafficher des parties de fenêtres. Ces événements dits de réaffichage ont pour périphérique source le système de fenêtrage lui-même. Le système de fenêtrage peut envoyer d'autres événements aux applications : par exemple, il peut envoyer des événements informant l'application des changements de taille et de position de ses fenêtres. Cela est utile pour que l'application puisse prendre en compte les manipulations des fenêtres par l'utilisateur via la gestionnaire de fenêtres (voir ci-dessous).

Le gestionnaire de fenêtres

Le système de fenêtrage est une couche logicielle qui permet au programmeur d'application interactive de gérer les fenêtre et leur contenu. Il ne doit pas être confondu avec le *gestionnaire de fenêtres* ("window manager"), qui est l'interface de manipulation de fenêtres qui est fourni à l'utilisateur. Le gestionnaire de fenêtres est une application interactive qui a la particularité de ne pas gérer ses propres fenêtres mais de permettre la manipulation des fenêtres de base des autres applications. On appelle fenêtre de base les fenêtres d'un modèle hiérarchique qui ont pour parent l'écran.

Les fonctions que fournit le gestionnaire de fenêtres à l'utilisateur sont le déplacement, changement de taille, iconification, désiconification et destruction des fenêtres de base. En général, ces commandes sont accessible par l'intermédiaire de la barre de titre des fenêtres de base.

Selon les systèmes de fenêtrage, l'implémentation du gestionnaire de fenêtres varie. Sur X Windows, c'est une application complètement indépendante. Il existe de nombreux gestionnaires de fenêtres (mwm, twm, gwm, etc.) parmi lesquels l'utilisateur peut choisir au gré de ses besoins. Par contre sur le Macintosh, chaque application est responsable de la prise en compte de la manipulation des fenêtres par l'utilisateur. Pour cela, lorsqu'elle reçoit un événement, elle doit calculer la zone de la fenêtre dans laquelle il a eu lieu (le Macintosh a un système de fenêtrage non hiérarchique) et s'il a eu lieu dans la barre de titre, elle doit transmettre l'événement à une procédure de traitement du gestionnaire de fenêtres qui se charge du feed-back et de l'exécution de la commande correspondante.

Exemple : X Windows

Le système de fenêtrage a été créé au cours des années 80 au sein du "project Athena" du MIT à Boston. Le but de ce projet était de fournir des moyens de calcul à l'ensemble des étudiants du campus, accessibles depuis n'importe quelle station de travail. A l'époque, chaque constructeur avait son propre système de fenêtrage, plutôt rudimentaire, et il était impossible d'envisager de développer tous les logiciels du projet sur chacun de ces systèmes de fenêtrage. Cela a conduit au développement du système X Windows (qui doit son nom au système W dont il s'est inspiré et qui avait été développé auparavant à Stanford), qui est devenu aujourd'hui un standard dans le monde Unix. Les créateurs de X Windows ont très tôt décidé que leur système serait disponible pour tout le monde sous forme source, afin d'encourager les portages sur le plus grand nombre possible de plateformes.

Le modèle client-serveur

L'originalité de X Windows est d'être fondé sur une architecture client-serveur : le serveur, dit serveur X s'exécute sur la machine qui dispose des périphériques d'entrée-sortie (un ou plusieurs écrans, un clavier et une souris). Il gère ces ressources matérielles ainsi qu'un ensemble de ressources logicielles (par exemple les polices de caractères). Les clients du serveur X sont les applications graphiques interactives. Elles se connectent, éventuellement au travers du réseau, au serveur, lui envoient des requêtes (création de fenêtre, dessin, etc.) et reçoivent du serveur réponses, erreurs et événements.

Les avantages principaux du modèle client-serveur sont l'indépendance vis-à-vis du matériel et la transparence du réseau : une application X peut s'exécuter sur n'importe quel serveur, sans recompilation, y compris à travers le réseau. De plus, centraliser les traitements graphiques dans un serveur permet de simplifier la gestion des ressources partagées, et permet de faire bénéficier toute application d'une amélioration des performances du serveur. Avec les systèmes classiques, l'application doit s'exécuter sur la machine qui dispose des périphériques physiques, et doit être recompilée chaque fois que l'on veut l'installer sur une nouvelle plateforme ou lorsqu'une nouvelle version du systèmes de fenêtrage est installée. Avec l'architecture client-serveur, une application de calcul intensif peut s'exécuter par exemple sur un CRAY et visualiser ses résultats sur une station de travail Unix classique.

L'inconvénient potentiel de l'approche client-serveur est la dégradation des performances due aux échanges de message via le réseau, ou même entre processus d'une même machine. La clé de l'efficacité de X est l'utilisation d'un protocole asynchrone : lorsque le client émet une requête, elle n'est pas immédiatement envoyée au serveur. Elle est stockée dans une tampon de sortie jusqu'à ce que l'une des situations suivantes se produise :

- le tampon de sortie est plein ;
- l'application force l'envoi du buffer par une commande "flush" ;
- l'application envoie une requête synchrone, qui requiert une réponse immédiate du serveur (par exemple quelle est la position actuelle d'une fenêtre) ;
- l'application attent un événement (appel de XNextEvent).

Dans ces 4 cas, les requêtes stockées dans le tampon de sortie sont envoyées au serveur en un seul message, ce qui optimise l'utilisation des canaux de communication. (En effet, dans un protocole comme TCP/IP, l'envoi d'un message d'un caractère prend autant de temps que l'envoi d'un message de 1 ou 2 kilo-octets.) De plus, si le client et le serveur sont sur des machines différentes, cet asynchronisme leur permet de travailler réellement en parallèle. Sur un bon

réseau local, on observe couramment que les performances sont meilleures lorsque le client et le serveur sont sur deux machines que s'ils sont sur la même machine. Par ailleurs, on peut mesurer que l'asynchronisme peut apporter un gain de performances allant jusqu'à un facteur 100 par rapport à une version synchrone du même protocole (envoi immédiat de chaque requête et attente de réponse).

Le fait que l'attente d'un événement force l'envoi des requêtes en attente rend l'asynchronisme du protocole pratiquement transparent pour le programmeur. En effet, une application X est une application dirigée par les événements : elle exécute une boucle qui attend un événement et le traite. Le traitement d'un événement génère en général des requêtes au serveur, qui sont stockées dans le tampon de sortie, et envoyées au serveur lorsque l'on boucle sur l'attente de l'événement suivant.

Il existe très peu de requêtes synchrones. Le client maintient un cache de certaines données qui permettent d'éviter des aller-retour vers le serveur. De plus les requêtes de création (de fenêtre, de contexte graphique, de police de caractères, etc.), qui semblent devoir être des requêtes synchrones sont en fait asynchrones : l'identificateur (unique) de la ressource à créer est généré par le client, stocké dans la requête en attente d'envoi, et retourné immédiatement à l'application :

```
wid = XCreateWindow (wparent, x, y, w, h, ...)
```

est implémenté comme :

```
Window XCreateWindow (Window parent, int x, int y, int w,
int h ...) {
      Window w = NewID ();
      SendRequest (CREATE_WINDOW, w, parent, x, y, w, h,
...);
      return w;
}
```

C'est-à-dire que le client demande au serveur de créer une fenêtre avec un identificateur donné.

Une fois qu'un client est connecté au serveur X, il n'y a pas de protection des ressources : n'importe quel client peut manipuler n'importe quelle ressource du serveur pour peu qu'il connaisse son identificateur. Cela est très utile dans diverses situations, notamment pour les gestionnaires de fenêtres qui ont justement pour mission de permettre la manipulation des fenêtres créées par les autres applications.

Modèle graphique, de fenêtrage et de gestion des entrées

Le modèle graphique de X est sans état. Les types de formes graphiques sont assez limités (par exemple il n'y a pas de courbes de Bezier ou de splines en standard) et la palette des attributs graphique est suffisante pour la plupart des applications 2D. L'ensemble des attributs graphiques sont groupés dans une structure appelée contexte graphique ("graphic context" ou GC). Des requêtes spécifiques permettent de créer et de manipuler les GC. Cela permet de réduire les requêtes de dessin (qui prennent un GC en paramètre et non pas une liste d'attributs) et d'améliorer les performances du serveur qui peut repérer l'usage répété d'un même GC. Un client peut dessiner dans n'importe quelle fenêtre : il lui suffit de connaître son identificateur. En pratique c'est rarement le cas.

Le modèle de fenêtrage est un modèle hiérarchique. Les fenêtres sont rectangulaires. Un serveur peut gérer plusieurs écrans, qui apparaissent comme autant de racines d'un arbre de fenêtre. Il est donc impossible d'avoir des fenêtres à cheval sur plusieurs écrans comme sur le Macintosh par exemple. Les fenêtres peuvent être reparentées (à condition que leur nouveau parent soit sur le même

écran, c'est-à-dire dans le même arbre de fenêtres que l'ancien). Le reparentage, est utilisé par les gestionnaires de fenêtres (voir ci-dessous).

La gestion des entrées est fondée sur les événements. Elle a été spécialement étudiée pour éviter d'engorger le réseau. Chaque application doit donner la liste des types d'événements qu'elle souhaite recevoir pour chaque fenêtre en spécifiant un *masque d'événements*. Par défaut, le masque est vide et l'application qui créer une fenêtre ne reçoit aucun événement pour cette fenêtre. Il est possible pour une application de recevoir des événements de n'importe quelle fenêtre, même s'il ne l'a pas créée : il lui suffit de connaître l'identificateur de la fenêtre. A quelques exceptions près, plusieurs clients peuvent recevoir des événements d'une même fenêtre : chaque masque d'événement est propre à chaque couple (client, fenêtre).

Pour réduire le trafic sur le réseau, le traitement des événements de déplacement de la souris est spécial : plutôt que d'envoyer autant d'événements que de déplacements élémentaires, ce qui encombrerait le réseau et pourrait surcharger l'application, le serveur n'envoie un événement de déplacement à un client que lorsque le dernier événement de déplacement qu'il lui a envoyé a été traité. Si le réseau transmet les événements rapidement et si l'application traite les événements rapidement, on aura un suivi fluide de la souris. Si les performances se dégrade, le suivi sera moins fluide mais toujours "au plus près" de la souris : on n'aura pas d'engorgement du réseau ou de la file d'attente qui provoqueraient une latence (éventuellement croissante !) de l'application.

Gestionnaires de fenêtres

Sous X, un gestionnaire de fenêtre est une application cliente du serveur X comme les autres. Pour permettre l'écriture de gestionnaires de fenêtres, le jeu de requêtes et d'événements a été enrichi par rapport à ce que l'on peut attendre d'un système de fenêtrage. Ainsi, il est possible d'interroger le serveur sur son état : liste des écrans, arbres des fenêtres, etc. D'autre part il est possible de recevoir des événements émis par le système de fenêtrage qui permettent de suivre précisément les changements de cet état : création, déplacement, changement de taille, destruction de fenêtre, etc. Il est même possible d'intercepter certaines requêtes émises par les autres clients. Par exemple, si le gestionnaire de fenêtres intercepte la requête de création de fenêtre, lorsqu'un client demande à créer une fenêtre, le serveur envoie un événement de demande de création au gestionnaire de fenêtres, qui a la charge de créer la fenêtre, éventuellement en changeant ses paramètres.

Même si l'interception de fenêtre est très intéressante dans certains cas, la simple notification des changements d'état et la possibilité de reparenter des fenêtres suffisent à implémenter un gestionnaire de fenêtres. Typiquement, lorsqu'un client crée une fenêtre de base (descendante directe d'un écran), le gestionnaire est notifié de la création. Il crée à son tour une fenêtre de base, un peu plus grande que celle qu'a créé le client, et reparente la fenêtre du client vers la fenêtre qu'il vient de créer. Cette fenêtre sert maintenant de cadre à la fenêtre du client. Le gestionnaire de fenêtres peut la décorer d'un cadre, de boutons, menus, etc. permettant la manipulation de la fenêtre. Les actions sur le cadre sont traitées par le gestionnaire de fenêtre, tandis que les actions dans la fenêtre du client sont traitées par celui-ci, qui ne s'est probablement même pas rendu compte de l'intervention du gestionnaire de fenêtres !

Boîtes à outils

La programmation de l'application interactive même la plus simple avec une librairie graphique et un système de fenêtrage devient rapidement rhédibitoire.

Créer un simple bouton nécessite par exemple de créer une fenêtre (dans un système de fenêtrage hiérarchique) qui contiendra le bouton, de le dessiner, de récuperer les événements qui concernent cette fenêtre, de les traiter par une machines à états, etc.

Les boîtes à outils d'interface fournissent un ensemble de composants appelés objets interactifs ou "widgets" (abbréviation de "window object") et un ensemble de fonctionnalités destinées à faciliter la programmation d'applications graphiques interactives. A travers son jeu de widgets, une boîte à outil implémente un "look and feel" particulier, c'est-à-dire un ensemble de règles de présentation et de comportement qui caractérisent la boîte à outils.

Les widgets

Les widgets d'une boîtes à outils sont regroupés en classes (au sens des langages à objets) : un widget appartient à une classe qui détermine :

- son apparence graphique (ou présentation) ;
- son comportement en réaction aux actions de l'utilisateur ;
- son interface avec le reste de l'application.

Par exemple, la classe des boutons définit leur apparence graphique (un cadre avec un nom à l'intérieur), leur comportement ("enfoncement" ou inversion video lorsque l'on clique dessus) et la façon dont ils sont liés au reste de l'application (par exemple une fonction appelée lorsque le bouton est cliqué).

L'arbre des widgets

On distingue en général deux catégories de widgets : les widgets "simples" (comme les boutons, barres de défilement, en-têtes de menus), et les widgets "composés" qui sont destinés à contenir d'autres widgets, simples ou composés (comme les boîtes de dialogue ou les menus). Les widgets sont donc organisés en un ou plusieurs *arbres de widgets* : la racine de l'arbre est un widget composé qui correspond à une fenêtre de base de l'application. Les nœuds de l'arbre sont des widgets composés qui permettent de structurer visuellement et/ou fonctionnellement le contenu de la fenêtre. Les feuilles de l'arbre sont des widgetst simples avec lesquels l'utilisateur peut interagir directement. Un arbre de widget correspond donc à une hiérarchie d'inclusion géométrique : un widget fils est inclus dans son widget parent. Cette règle d'inclusion géométrique peut cependant ne pas être systématique. Par exemple, un menu déroulant est souvent considéré comme un fils de l'en-tête de menu mais il n'est évidemment pas inclus géométriquement dans cet en-tête.

Chaque widget a un nom qui doit être unique parmi ses frères dans l'arbre. On peut donc désigner un widget par son chemin d'accès complet dans l'arbre, comme un nom de fichier dans un système de fichiers hiérarchique.

Apparence

L'apparence d'un widget est paramétrée par un ensemble d'attributs. Par exemple, il doit être possible de spécifier la couleur et le texte d'un bouton. Les valeurs de ces attributs sont définis lorsque l'application crée le widget et peuvent être modifiés par la suite. Souvent, les valeurs des attributs peuvent également être définies de façon externe à l'application. Cela permet de modifer l'aspect de l'interface sans toucher à l'applicatios. Dans l'environnement X Windows, un fichier texte permet d'effectuer cette paramétrisation. Par exemple la ligne

```
.MonAppli.mbar.edit.menu.copier.label: Copy
```

indique que le texte du widget dont le chemin d'accès est "mbar.edit.menu.copier" dans l'application "MonAppli" est "Copy". Sur le Macintosh, ces attributs sont stockés dans des "ressources" qui peuvent être éditées avec un éditeur approprié.

La paramétrisation externe de l'apparence est souvent utilisée pour adapter les logiciels à différentes langues (on parle de "localisation"). Cette paramétrisation est une configuration de l'application, car elle n'est prise en compte qu'au lancement de celle-ci. En général, il n'est pas possible de changer l'apparence de façon externe pendant que l'application s'exécute. Par contre l'application peut à tout moment changer les valeurs des attributs d'un widget.

Comportement

Le comportement d'un widget est le plus souvent complétement prédéfini dans sa classe. Certains aspects peuvent éventuellement être configurés par des attributs comme pour l'apparence, mais c'est le comportement d'un widget qui est le plus caractéristique de son type. Ainsi, un bouton est un bouton parce qu'il se comporte comme un bouton.

De façon interne, le comportement est défini par la description du traitement des événements clavier et souris qui arrivent au widget. Dans certains cas, cette description se fait par un ensemble de liaisons d'événements ("bindings" ou "handlers"). Une liaison d'événement associe une fonction à exécuter lors de l'occurence d'un événement d'un type donné ou d'une séquence d'événements d'un type donné. Une machine à états (voir chapitre précédent) peut être traduite en un ensemble de liaisons d'événements : pour chaque événement susceptible d'être pris en compte par la machine à états, on crée une liaison dont la fonction associée exécute une transition en fonction de l'état courant de la machine qui est stocké dans le widget.

Interface d'application des widgets

Pour être utile un widget doit pouvoir communiquer avec le reste de l'application. Ainsi, lorsque l'utilisateur clique sur le bouton, le comportement du widget doit changer son apparence, et doit également informer l'application que ce bouton a été activé. Il existe trois principaux mécanismes de communication entre un widget et l'application :

- les fonctions de rappel ou "callbacks", disponibles dans la plupart des boîtes à outils ;
- les variables actives, disponibles par exemple dans Tcl/Tk et dans le constructeur d'interfaces XFaceMaker ;
- les événements, utilisés par exemple dans la boîte à outils AWT de Java ou dans la boîte à outils PowerPlant de l'environnement de développement CodeWarrior de Metrowerks.

Fonctions de rappel

Une *fonction de rappel* est définie en l'enregistrant auprès d'un widget. Une fois enregistrée, elle est appelée par le widget lorsque celui-ci est activé. Si le comportement est défini par une machine à états, c'est une action liée à l'une des transitions qui appelle la fonction de rappel. Le code d'un programme utilisant des fonctions de rappel à l'aspect suivant :

```
/* fonction de rappel */
void CbOK (Widget w, void* data) {
     printf ("fonction de rappel : %s\n", (char*) data);
}

/* programme principal */
```

```
main () {
      ....
      /* créer le widgets et lui associer sa callback */
      ok = CreateButton (....);
      RegisterCallback (ok, CbOK, "coucou");
      ...
      /* boucle de traitement des événements */
      MainLoop ();
}
```

Dans cet exemple, le bouton est créé au début du programme principal. Puis celui-ci appelle la boucle principale MainLoop, qui est une fonction fournie par la boîte à outils. Cette fonction reçoit les événements, les envoie aux différents widgets qui les traitent, appelant des fonctions de la boîte à outils pour produire le feed-back et les fonctions de rappel. Le corps de la fonction de rappel (ici CbOK) doit être spécifié ailleurs. La fonction de rappel est appelée avec le nom du widget qui l'a déclenchée et une donnée opaque qui a été enregistrée lors de la spécification de la fonction de rappel (RegisterCallback). Ici la donnée opaque est une chaîne de caractères "coucou" qui est imprimée par la fonction de rappel. La même fonction de rappel pourrait être attachée à plusieurs widgets mais avec une donnée opaque différente.

Ces données opaques sont indispensables pour transférer de l'information entre les widgets et l'application. Par exemple, supposons une boîte de dialogue qui contienne les traditionnels boutons "OK" et "Annuler". Ces deux boutons doivent (entre autres) fermer la boîte de dialogue. Pour cela ils doivent connaître la boîte de dialogue. Si l'on veut éviter l'usage de variables globales, il faut communiquer cette information à la fonction de rappel via la donné opaque :

```
/* fonction de rappel pour fermer une fenêtre */
void CbClose (Widget w, void* data) {
      Widget dlog = (Widget) data;
      CloseWidget (dlog);
}

/* programme principal */
main () {
      ...
      /* créer la boîte de dialogue */
      dlog = CreateDlog (....);
      /* créer le bouton OK et sa callback */
      ok = CreateButton (....);
      RegisterCallback (ok, CbClose, dlog);
      ...
      MainLoop ();
}
```

Les avantages des fonctions de rappel sont leur généralité et leur simplicité. Leur inconvénient principal est qu'elles conduisent à une délocalisation du flot d'exécution qui rend la mise au point et la maintenance difficiles. Ainsi, dans les exemples ci-dessus, on remarque qu'il n'y a pas dans le programme d'appel explicite aux fonctions de rappel. Ceux-ci sont "cachés" dans le comportement des widgets. On remarque également que le type de la donnée opaque oblige à une conversion de type dans la fonction de rappel qui est une source importante d'erreurs à l'exécution. Dans des applications réelles, le nombre de fonctions de rappel devient rapidement très grand et la mise en place des données opaques devient un casse-tête si l'on ne met pas en place des conventions de codages et des structures de données appropriées. Cette complexité paraît pourtant inutile face à la simplicité apparente des actions à réaliser comme l'ouverture et la fermeture de fenêtres...

Considérons le cas d'une boîte de dialogue dont l'action doit être exécutée sur activation du bouton OK. Une façon simple et intuitive de programmer cette boîte de dialogue pourrait se présenter comme ceci :

```
MonDlog () {
      /* créer la boite et ses widgets */
      ...
      /* traiter l'interaction avec la boite */
      tanque on n'a pas cliqué sur OK ou Annuler
            traiter un événement
            mettre à jour l'état courant de la boîte
      fin tantque
      /* OK */
      si on a clique sur OK
            exécuter la commande de la boîte de dialogue
      fermer la boîte de dialogue
}
```

Malheureusement, le traitement dirigé par les événements, par son contrôle centralisé dans la boucle d'événements, impose une "atomisation" du code de traitement des actions de l'utilisateur. Le code qu'il faut écrire a ainsi l'aspect suivant :

```
/* fonctions de rappel des champs de la boîte */
CbChamp1 (Widget w, void* data) {
      DlogState* dlog = (DlogState*) data;
      /* mettre à jour l'état en fonction du champ */
      dlog->champ1 = GetValue (w);
}
...
/* fonctions de rappel pour OK et Annuler */
CbOk (Widget w, void* data) {
      DlogState* dlog = (DlogState*) data;
      /* exécuter la commande de la boîte de dialogue */
      ...
      /* fermer la boite */
      CloseWindow (dlog->window);
      free (dlog)
}
CbCancel (Widget w, void* data) {
      DlogState* dlog = (DlogState*) data;
      CloseWindow (dlog->window);
      free (dlog);
}

/* creation de la boite */
MonDlog () {
      /* creer la structure stockant l'etat */
      DlogState* state = new (DlogState);
      ...
      /* creer la boite de dialogue et ses widgets */
      state->window = CreateWindow (...);
      w = CreateXXX (...);
      CreateCallback (w, CbChamp1, state);
      ...
      /* créer les boutons OK et Annuler */
      ok = CreateButton (...);
      CreateCallback (w, CbOk, state);
      cancel = CreateButton (...);
      CreateCallback (w, CbCancel, state);

      /* MainLoop se charge de l'appel des callbacks */
}
```

Variables actives

Les *variables actives* sont une alternative aux fonctions de rappel qui peut être utilisée dans de nombreuses situations. Une variable active établit un lien entre une variable de l'application et un widget représentant une valeur. Ce lien est bidirectionnel : si l'application change la valeur de la variable, l'aspect du widget change pour refléter la nouvelle valeur ; si l'utilisateur agit sur le widget pour changer la valeur qu'il représente, cette nouvelle valeur est affectée à la variable. Une variable active peut éventuellement être liée à plusieurs widgets, les forçant à tous refléter la même valeur. L'exemple d'utilisation ci-dessous concerne une case à cocher qui représente un état booléen :

```
main () {
      bool  recto_verso = false;
      ...
      /* création du widget */
      rv = CreateCheckBox (....);
      /* établissement du lien avec la variable active */
      SetActiveVariable (rv, &recto_verso);
      ...
}
```

Dans cet exemple, lorsque l'application change la valeur de la variable recto_verso, il faut s'assurer que le widget met à jour son état. Trois approches sont possibles :

- on exige que l'application appelle la fonction UpdateActiveVariable de la boîte à outils pour mettre à jour le widget. Cette approche est risquée car il est probable que l'on oublie des appels à UpdateActiveVariable.
- on met en place une tâche de fond, activée par exemple à chaque passage dans la boucle d'événements, qui détecte les variables actives dont les valeurs ont changé et met à jour leurs widgets associés. Cette solution est potentiellement coûteuse si l'on a un très grand nombre de variables actives.
- on utilise un mécanisme du langage. Par exemple, certains langages (C++ et ADA par exemple) permettent de redéfinir l'affectation. Il suffit alors de définir un nouveau type dont l'affectation est redéfinie pour appeler UpdateActiveVariable. C'est l'approche la plus efficace, mais elle n'est pas toujours applicable (par exemple en C ou Pascal).

Si une boîte à outils de dispose pas de variables actives, il est souvent possible de les simuler grâce avec des fonctions de rappel. Voici ce que cela donne dans l'exemple de la case à cocher, en supposant que les fonctions GetValue et SetValue de la boîte à outils permettent d'accéder et de modifier la valeur représentée par un widget (case à cocher, bouton radio, potentionmètre, etc.) :

```
/* structure de données définissant une variable active :
   une liste de widget et un pointeur vers la variable C
*/
typedef struct {
      WidgetList  wl;
      int*        var;
} ActiveVar;

/* fonction à appeler pour mettre à jour les widgets
   lorsque l'on change la valeur d'une variable active.
   GetActiveVar parcourt une liste de valeurs actives
   et retourne celle qui est liée à la variable var.
*/
void UpdateActiveVar (int* var) {
      ActiveVar* = GetActiveVar (var);
      Widget w;
```

```
        for (w = wl->first; w; w = w->next) {
              SetValue (w, *av->var);
        }
}

/* fonction de rappel associée à un widget qui est lié
   à une variable active. La donnée opaque est un pointeur
   sur la variable active.
*/
void CbUpdateVar (Widget w, void* data) {
        ActiveVar* av = (ActiveVar*) data;
        Widget cw;
        /* stocker la nouvelle valeur */
        *av->var = GetValue (w);
        /* mettre à jour les autres widgets */
        for (cw = wl->first; cw; cw = cw->next) {
              if (cw != w) SetValue (cw, *av->var);
        }
}

main () {
        bool recto_verso = false;
        ActiveVar rv_av = CreateActiveVar (&recto_verso);
        ...
        /* création du widget */
        rv = CreateCheckBox (....);
        /* association de la variable active.
           AddWidget ajoute un widget à la liste des widgets
           de la variable active. */
        AddWidget (rv_av, rv);
        SetCallback (rv, CbUpdateVar, &rv_av);
        ...
        MainLoop ();
}
```

[parler des fonctions de conversion, de la possibilité de stocker l'ancienne valeur pour savoir les valeurs actives qui ont changé, montrer comment redéfinir l'affectation en C++ pour rendre la mise à jour transparente...]

Evénements

La troisième façon pour le widget de communiquer avec l'application est d'utiliser des événements : le widget se comporte comme un périphérique logique qui émet des événements lorsque son état change, de la même façon qu'un périphérique physique émet des événements lorsque son état change.

Le traitement de ces événements de haut niveau est généralement réalisé en associant à chaque widget un objet "cible" auquel les événements émis par le widget sont transmis. Cette cible est similaire dans son principe à la fenêtre cible définie par le focus des événements dans un système de fenêtrage (voir plus haut). On peut d'ailleurs imaginer un mécanisme de distribution plus sophistiqué à l'image du focus implicite ou esclave des systèmes de fenêtrage.

Un langage à objets est particulièrement adapté à cette approche : on définit la classe Handler des objets pouvant traiter des événements :

```
class Handler {
public:
        virtual void HandleEvent (WidgetEvent* ev) = 0;
};
```

Pour pouvoir être recevoir les événements émis par un widget, un objet doit hériter de Handler et redéfinir la méthode HandleEvent. Par exemple, on peut définir une boîte de dialogue comme un objet qui traite les événements envoyés par les champs d'une boîte de dialogue ainsi que par ses boutons OK et Cancel :

```
class Dlog : public Handler {
protected:
      /* widget représentant la boîte de dialogue*/
      Widget      dlog;
public:
      /* constructeur */
      Dlog (Widget dlog);

      /* fonction executee sur OK */
      virtual void Validate () = 0;

      void HandleEvent (WidgetEvent* ev) {
            switch (ev->type) {
            case CANCEL :
                  CloseWindow (dlog);
                  break;
            case OK :
                  Validate ();
                  CloseWindow (dlog);
                  break;
            default :
                  Handler::HandleEvent (ev);
            }
      }
};
```

Pour créer une boîte de dialogue "réelle", il faut créer une nouvelle classe dans laquelle on redéfinit Validate et HandleEvent :

```
class PrintDlog : public Dlog {
protected:
      char* fileName;
      int   numCopies;
public :
      PrintDlog (Widget dlog) : Dlog (dlog) {
            fileName = 0;
            numCopies = 1;
      }
      void Validate () {
            if (filName && numCopies > 0)
                  DoPrint (fileName, numCopies);
      }
      void HandleEvent (WidgetEvent* ev) {
            switch (ev->type) {
            case SETNAME :
                  fileName = GetStringValue (ev->widget);
                  break;
            case SETCOPIES :
                  numCopies = GetValue (ev->widget);
                  break;
            default :
                  Dlog::HandleEvent (ev);
            }
      }
};
```

Enfin il faut créer une instance de cette classe et les widgets constituant la boîte de dialogue :

```
main () {
```

```
        PrintDlog* print;

        Widget dlog = CreateDialog (....);
        Widget name = CreateEntry (....);
        Widget copies = CreateScale (....);
        Widget ok = CreateButton (....);
        Widget cancel = CreateButton (....);

        print = new PrintDlog (dlog);

        /* definir la cible de chaque widget avec son type */
        SetTarget (name, SETNAME, print);
        SetTarget (copies, SETCOPIES, print);
        SetTarget (ok, OK, print);
        SetTarget (cancel, CANCEL, print);
        ...
        MainLoop ();
}
```

L'avantage des événements est de découpler clairement l'émission et le traitement sans pour autant distribuer le code des traitements dans diverses fonctions comme c'est le cas avec les fonctions de rappel. Les événements permettent également de modifier plus aisément les comportements. Par exemple si l'on souhaite que la touche retour chariot fasse la même chose qu'un clic sur le bouton OK, il suffit de s'arranger pour que le retour chariot émette un événement de type OK vers la bonne cible. Enfin, on peut utiliser pour décrire le traitement de ces événements les même machines à états que celles que l'on a utilisé pour traiter les événements de plus bas niveau. Dans l'exemple ci-dessus, une machine à états permettrait de spécifier aisément que la zone texte doit être non vide pour pouvoir valider par OK.

Placement des widgets

Une partie des fonctionnalités d'une boîte à outils est dédiée au contrôle du placement des widgets à l'écran. Ce problème est complexe car il faut être capable de décrire des placements indépendamment de la taille des widgets. Par exemple, si l'on construit un menu, la largeur du menu doit être au moins égale à la largeur du plus large de ses items. Mais cette taille dépend de divers facteurs qui ne sont connus qu'à l'exécution : police de caractère disponible pour écrire le texte de chaque item, ainsi que le texte lui-même qui peut être spécifié de façon externe à l'application par l'intermédiaire des ressources. Un autre exemple est celui d'une fenêtre contenant une barre de menus, une zone texte et une barre de défilement verticale. Si l'utilisateur change la taille de la fenêtre par l'intermédiaire du gestionnaire de fenêtre, les widgets internes doivent s'ajuster : la barre de menus doit s'ajuster horizontalement, la barre de défilement doit s'ajuster verticalement en restant à droite de la fenêtre et la zone texte doit s'ajuster pour occuper le reste de l'espace disponible.

Dans certains cas (par exemple sur le Macintosh), le placement est à la charge de l'application : la boîte à outils n'offre pas de fonctionnalités particulière sinon la possibilité d'interroger et de modifier la taille et la position de chaque widget.

Dans les autres cas, la boîte à outils offre un ensemble de *gestionnaires de géométrie*, correspondant à diverses stratégies de placement. Dans la X Toolkit et MOTIF, ces gestionnaires de géométrie sont implémentés dans les widgets composés : lorsque l'on ajoute un widget dans l'arbre, son nœud parent est un widget composé qui se charge de placer le nouveau widget en fonction de ses frères. MOTIF fournit par exemple un widget composé RowColumn qui fait des placements de widgets en ligne ou en colonne, et qui est utilisé notamment pour

les menus : chaque nouveau widget est placé en-dessous du dernier widget placé, et la largeur du RowColumn est éventuellement ajustée si le nouveau widget est plus large que tous les autres. Dans Tcl/Tk, les gestionnaires de géométrie sont indépendants des widgets : on peut associer n'importe quel gestionnaire de géométrie à n'importe quel widget composé.

Les exemples présentés au début de cette section montrent que le placement doit prendre en compte des contraintes provenant des feuilles de l'arbre des widgets et des contraintes provenant de la racine de l'arbre. Les contraintes provenant des feuilles sont dues au fait que chaque widget simple a une taille "naturelle" ou minimale pour pouvoir s'afficher correctement, qui dépend de ses attributs de présentation. Les contraintes provenant de la racine sont dues au fait que l'utilisateur peut généralement contrôler la taille des fenêtres principales de l'application.

Une technique de placement utilisée par exemple dans le widget "Frame" de MOTIF consiste à décrire un ensemble de contraintes orientées imposant une distance fixe entre les bords des widgets. Pour qu'un bouton OK, par exemple, reste en bas et à droite d'une boîte de dialogue, il suffit d'"accrocher" son côté droit au côté gauche de la boîte de dialogue, et son côté inférieur au côté inférieur de la boîte de dialogue. Si les deux côtés opposés d'un widget ne sont pas soumis à des contraintes, le widget conserve sa taille naturelle dans cette dimension : ici, le bouton OK aura sa taille naturelle. Si l'on attache les bords gauche et droit d'un barre de menus aux bords gauche et droit de la fenêtre englobante, on impose à la barre de menu une taille horizontale (figure).

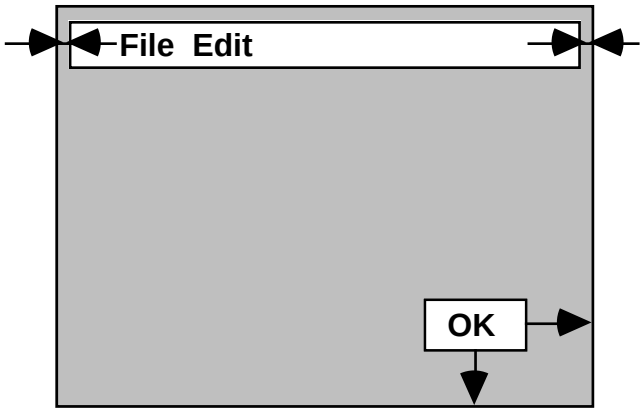

figure : placement des widgets par contrainte de distance

L'avantage de ce placement par contraintes est que l'on peut recalculer le placement aussi bien lorsque la taille d'un des composants change que lorsque la taille de la fenêtre change. L'inconvénient est que le nombre de contraintes de distances à spécifier devient rapidement important et le risque de décrire un ensemble de contraintes incohérentes, c'est-à-dire ne pouvant être satisfaites simultanément.

D'autres méthodes générales de placement existent : Tcl/Tk dispose du "packer" (voir plus loin), et Ilog Views permet des placements relatives à une grille : on spécifie la grille comme un ensemble de lignes horizontales et verticales placées relativement à la fenêtre, puis on attache les widgets à cette grille avec des contraintes de distance comme pour le placement de type "Frame". Cette technique a l'avantage de permettre des placements plus sophistiqués que le Frame tout en simplifiant la spécification. De plus, les graphistes travaillent presque toujours avec des grilles car elles permettent des placement plus esthétiques.

Les boîtes à outils disposent également de gestionnaires de géométrie plus spécialisés : le RowColumn de MOTIF a déjà été mentionné ; Tcl/Tk a un gestionnaire de placement sur une grille bidimensionnelle ("grid") qui permet de réaliser des tables complexes.

Autres fonctions d'une boîte à outils

Les principales de fonctions d'une boîte à outils concernent la gestion des widgets : création, destruction, placement, changement des attributs, spécification des fonctions de rappel, etc. Une boîte à outil impose un style de programmation dirigé par les événements et fournit donc également un ensemble de fonctions pour le contrôle du flux d'exécution. Nous avons déjà mentionné la fonction MainLoop qui est la boucle principale de traitement des événements. Les boîtes à outils permettent généralement de paramétrer le déroulement de la boucle principale pour intégrer des tâches de fond, pour prendre en compte d'autres sources de données que les événements, et pour permettre la programmation d'interactions modales. D'autre part, certaines boîtes à outils sont extensibles et permettent de créer de nouvelles classes de widgets.

Tâches de fond

Dans une appliction dirigée par les événements, il est crucial pour que le système donne une bonne impression de réactivité aux actions de l'utilisateur que la boucle d'événements soit parcourue suffisamment rapidement. Sinon, les actions de l'utilisateurs engorgent la file d'attente des événements et le décalage entre l'émission de l'événement et le moment de son traitement augmente. Si l'on doit réaliser un calcul long (typiquement supérieur à 1/2 ou 1 seconde), il faut faire en sorte que l'utilisateur soit informé de la progression de l'opération et que la boucle d'événements est parcourue, ne serait-ce que pour prendre en compte les événements de réaffichage par exemple. Sinon, pendant toute la durée du calcul, l'application ne serait pas capable de raffraichir ses fenêtres si elles venaient à être déplacées ou démasquées. Une situation similaire se produit lorsque l'on utilise de l'animation : pendant que l'animation se déroule, l'application doit pouvoir répondre aux événements.

Une boîte à outil doit donc fournir un moyen de réaliser une ou plusieurs tâches de fond, en parallèle avec la boucle principale de traitement des événements. Si le système d'exploitation fournit des "threads" ou processus légers, la mise en œuvre est relativement simple, au moins conceptuellement : on crée un processus léger qui exécute la tâche de fond. Dans la réalité, les choses sont compliquées par la nécessité de prévenir les conflits d'accès aux variables partagées par plusieurs processus légers : il faut vérouiller et déverouiller les objets partagés, ce qui peut conduire à des interblocages.

Si le système d'exploitation ne fournit pas de processus légers, il faut simuler le parallélisme en implémentant la tâche de fond par une fonction appelée par la boucle d'événements à chaque tour de boucle ou, pour des tâches moins prioritaires, lorsque la file d'attente est vide. La fonction implémentant la tâche de fond doit gérer un état global de façon à exécuter seulement une partie de la tâche à chaque invocation. L'exemple suivant montre comment on peut utiliser une tâche de fond pour lire un fichier en tâche de fond. Il utilise la fonction CreateIdleProc de la boîte à outils. Cette fonction prend deux paramètres : la fonction implémentant la tâche de fond et une donnée opaque transmise à la fonction lorsqu'elle sera appelée. CreateIdleProc s'arrange pour que MainLoop appelle la fonction passée en paramètre au prochain passage dans la boucle d'événements. Nous supposons ici que CreateIdleProc résulte en exactement un appel de la fonction passée en paramètre. Pour créer une tâche de fond, il faut donc que cette

fonction appelle elle-même CreateIdleProc pour s'assurer que la tâche puisse se dérouler.

```
/* structure décrivant l'état de la tâche de fond */
typedef struct {
      int   fd;         /* fichier à lire */
      char* buffer;     /* buffer pour lire le fichier */
      int   size; /* taille du buffer */
      char* curBuf;     /* début de zone libre du buffer */
      Widget progress;  /* barre de progression */
} ReadFileTask;

/* fonction implémentant la tâche de fond */
void ReadFileIdleProc (void* data) {
      ReadFileTask* task = (ReadFileTask*) data;
      /* lire un bloc */
      int n = read (task->fd, task->curBuf, 1024);
      if (n == 0) {
            /* fin de fichier */
            close (task->fd);
            /* traiter le contenu du buffer */
            ...
            free (task);
      } else {
            task->curBuf += n;
            /* informer la barre de progression */
            SetValue (task->progress,
                (task->curBuf-task->buffer)/task->size);
            /* relancer la tâche */
            CreateIdleProc (ReadFileIdleProc, data);
      }
}

/* fonction lançant la tâche de fond.
   CreateReadFileTask est une fonction de l'application
   qui crée et initialise une structure ReadFileTask */
void ReadFile (char* name) {
      ReadFileTask* task;
      int fd = open (name, "r");
      if (fd < 0) return;    /* erreur */
      /* creer et initialiser l'état de la tâche */
      task = CreateReadFileTask (fd);
      /* lancer la tâche de fond */
      CreateIdleProc (ReadFileIdleProc, task);
}
```

Dans certaines boîtes à outils (par exemple Tck/Tk), on peut demander programmer l'exécution de la tâche de fond après un certain délai (commande "after" de Tk). Cela est particulièrement utile pour produire des animations puisque dans ce cas la tâche doit s'exécuter par exemple tous les 1/10ème de seconde.

Autres sources de données

La boucle principale de la boîte à outils traite une seule source de données : les événements transmis par le système de fenêtrage. Beaucoup d'applications, notamment les applications réseau, doivent gérer plusieurs sources de données : par exemple un serveur doit gérer une source de données pour chaque client connecté. On pourrait utiliser le mécanisme des tâches de fond pour que l'application "écoute" simultanément ces différentes sources, en créant une tâche de fond dédiée à chaque source qui attende que des informations arrivent de cette

source. Cependant, cette technique risque d'aboutir à une attente active consomatrice de ressources.

Aussi de nombreuses boîtes à outils permettent de prendre en compte des sources extérieures de données, soit en étendant leur modèle d'événements soit en permettant de définir des "handlers" pour chaque source extérieure.Dans la X Toolkit, les seules sources de données qui peuvent être prises en comptes sont les descripteurs de fichiers Unix (qui peuvent correspondrent à des canaux de lecture/écriture de fichiers, mais aussi à des canaux de communication avec d'autres processus, éventuellement au travers du réseau). Dans la boîte à outils Tcl/Tk, on peut définir des sources de données arbitraires. On peut par exemple imaginer une source de données qui délivrerait un événement tous les n millisecondes, ou lorsqu'une variable change de valeur, etc.

Interaction modale

Dans une interaction modale, on réduit le champ des possibilités de l'utilisateur en le forçant à interagir seulement avec une partie de l'interface. Par exemple, quand une boîte de dialogue informe l'utilisateur qu'il n'y a pas assez de place pour enregistrer son document, on souhaite imposer à l'utilisateur de valider cette boîte en rendant le reste de l'interface inactif. Cela implique un certain nombre de services spécifiques de la part de la boîte à outils, notamment :

- la possibilité de limiter les événements pris en compte à un sous-ensemble des widgets ;
- la possibilité d'activer une boucle locale de traitement des événements jusqu'à ce qu'une certaine condition soit satisfaite.

Pour limiter les widgets susceptibles de recevoir des événements, les boîtes à outils offrent la notion de "grab" (littéralement, s'emparer de). Lorsque l'on pose un "grab" sur un widget, les événements d'entrée qui parviennent à la boîte à outils et qui concernent des widgets qui ne sont pas des descendants du widget "grabbé" sont ignorés. Les événements qui ne correspondent pas à des actions de l'utilisateur, comme les demandes de réaffichage par exemple, continuent d'être pris en compte pour tous les widgets. Le "grab" peut être local à l'application ou global. S'il est local, l'utilisateur peut activer une autre application tandis que s'il est global il ne peut changer d'application. Les grabs globaux doivent être réservés à des situations particulières (typiquement les pannes ou les problèmes matériels).

La possibilité de traiter localement les événements sans passer par la boucle principale permet d'éviter la délocalisation du flot de contrôle à travers les fonctions de rappel. Selon les cas, la boîte à outils fournit des fonctions élémentaires de traitement des événements qui permettent au programmeur d'écrire sa propre boucle, ou bien des boucles prédéfinies qui terminent selon des conditions telles que la disparition d'un widget (WaitUntilClosed) ou le changement de valeur d'une variable (WaitUntilChanged).

Supposons que l'on souhaite implémenter une commande qui formatte une diskette. Il est prudent de demander confirmation de cette commande à l'utilisateur :

```
/* fonctions de rappel pour le boutons Oui et Non */
CbYes (Widget w, void* data) {
      int* res = (int*) data;
      *res = 1;
}

CbNo (Widget w, void* data) {
      int* res = (int*) data;
      *res = 0;
```

```
}

/* boite de dialogue modale pour demander confirmation */
bool YesNo (char* msg) {
      int res = -1;
      /* creer une boite de dialogue avec un message et
         deux boutons Oui et Non */
      dlog = CreateWindow (...);
      ...
      yes = CreateButton (...);
      CreateCallback (yes, CbYes, &res);
      no = CreateWidget (...);
      CreateCallback (no, CbNo, &res);

      /* poser un "grab" sur la boite de dialogue */
      GrabLocal (dlog);

      /* attendre que la reponse ait change */
      WaitUntilChanged (&res);
      CloseWindow (dlog);
}

/* exemple d'utilisation */
Reformatter (...) {
      ...
      /* demander confirmation */
      ok = YesNo ("Cette commande efface votre disquette.
                   Etes-vous sur de vouloir continuer ?");
      if (ok) {
            /* reformattage */
            ...
      }
}
```

On remarque que le style de programmation devient plus classique : la fonction Reformatter a un code séquentiel qui prend en compte le résultat de la fonction YesNo. Sans traitement local des événement, il aurait fallu découper la fonction Reformatter en deux en mettant la partie qui réalise le reformattage dans la fonction de rappel du bouton Oui. Il est donc tentant d'utiliser fréquemment les boucles locales de traitement des événements. Il faut cependant être très prudent car si elles ne sont pas associées à un "grab", elles peuvent conduire à des surprises. En effet, chaque boucle est en attente d'une condition unique (changement de valeur d'une variable donnée ou fermeture d'une fenêtre donnée). Si plusieurs boucles locales de traitement sont appelées de façon imbriquées, seule la boucle la plus interne est active et donc seule la condition qui contrôle cette boucle est prise en compte. En pratique, on ne peut donc utiliser de boucle locale que lorsque l'on utilise un "grab".

Création de widgets

Les boîtes à outils fournissent une collection de classes de widgets qui couvre l'essentiel des besoins des interfaces graphiques "classiques" : boutons, menus, boîtes de dialogue, barres de défilement, zones d'entrée de texte, etc. Cependant, il est fréquent qu'une application particulière nécessite ou puisse profiter d'un nouveau type de widget. Par exemple, dans un traitement de texte, une règle permet fréquemment de spécifier les marges et les positions des tabulations.

La plupart des boîtes à outils sont extensibles, c'est-à-dire que l'on peut définir de nouvelles classes de widgets. Cependant, la programmation d'une nouvelle classe est le plus souvent beaucoup plus difficile que l'utilisation des widgets existants, et nécessite l'utilisation des interfaces de programmation internes à la boîte à

outils. Par exemple, sous X Windows, la X Toolkit fournit un ensemble de mécanismes de base (les "Intrinsics") qui sont utilisés pour programmer un jeu de widget "widget set" comme MOTIF. Malgré cela, la programmation de nouveau widgets reste délicate. Cette difficulté d'extension explique que beaucoup d'applications se contentent des widgets fournis par la boîte à outils en standard, au détriment de leur utilisabilité.

Les boîtes à outils implémentées au-dessus d'un système de fenêtrage hiérarchique implémentent souvent la présentation des widgets par des fenêtres. Cela offre l'avantage que le système de fenêtrage fait déjà une grande partie du travail : en identifiant la fenêtre dans laquelle a lieu chaque événement (d'entrée, de réaffichage, etc.), il facilite le routage de l'événement vers le bon widget par la boîte à outils.

Voici, à titre d'exemple, une classe C++ (simplifiée) qui décrit un widget générique :

```
class Widget {
protected:
      Widget* parent;   // widget parent
      Widget* next;     // frère à ce niveau de l'arbre
      char*   name;     // nom du widget
      int     x, y,     // position
      int     w, h;     // taille
      Window  window;   // fenetre implémentant le widget

      virtual void Draw () = 0;
      virtual void doRealize ();
public:
      Widget (Widget* parent);
      virtual void DispatchEvent (Event* ev);
      void Resize (int w, int h);
      void Move (int x, int y);
      void Realize ();
};
```

La méthode DispatchEvent traite les événement de réaffichage en appelant la méthode Draw, qui doit être redéfinie dans les sous-classes. Les méthodes Resize et Move permettent de manipuler la géométrie du widget. La méthode Realize construit le widget, c'est-à-dire crée sa fenêtre et la rend visible. Elle appele doRealize qui peut être redéfinie dans les sous-classes pour calculer la taille par défaut du widget.

La classe ButtonWidget hérite de Widget et illustre la façon de créer de nouvelles classes de widgets :

```
class ButtonWidget : public Widget {
protected:
      char*    label;    // texte du bouton
      Color    bg, fg;   // couleurs de fond et de texte
      Font     font;     // police de caractères
      Callback activate; // fonction de rappel

      void Draw ();
      void doRealize ();
public:
      ButtonWidget (Widget* parent);
      void SetForeground (Color fg);
      void SetBackground (Color bg);
      void SetFont (Font f);
      void SetLabel (char* l);
      void SetCallback (Callback cb);
```

```
        void DispatchEvent (Event* ev);
};
```

Les méthodes SetXXX permettent de définir ou de modifier les attributs de présentation et les callbacks du widget. La méthode doRealize héritée de Widget est redéfinie : elle calcule la taille du bouton en fonction de la police de caractères et du label du bouton. De même, Draw est redéfinie et implémente le dessin du bouton dans sa fenêtre en appelant les fonctions de dessin de la librairie graphique. Enfin, DispatchEvent est redéfinie : elle traite les événements d'entrée en implémentant la machine à états du bouton, et elle passe les événements qu'elle ne sait pas traiter (en particulier les événements de redessin) à la classe Widget. C'est DispatchEvent qui appelle la fonction de rappel "activate" lorsque le bouton est activé.

Générateurs d'interfaces

La programmation d'applications graphiques interactives avec une boîte à outils d'interface est aujourd'hui la méthode la plus répandue. Pour des applications importante, elle implique cependant un coût de développement important, notamment par la difficulté de mise au point dû au modèle de programmation par événements et la quasi-impossibilité d'automatiser les tests. Les générateurs d'interfaces sont des outils de plus haut niveau destinés à réduite ces coûts. Comme leur nom l'indique, il s'agit d'outils destinés à produire une partie de l'application interactive à partir d'une description de haut niveau.

L'approche des générateurs d'interface peut être comparée aux outils de production de compilateurs : grâce aux expressions régulières et aux grammaires algébriques, on peut décrire respectivement le lexique et la syntaxe d'un langage de programmation. Des outils de génération sont capables de produire automatiquement, à partir de ces description, un analyseur lexical et un analyseur syntaxique du langage. Grâce à ces outils, il est beaucoup plus facile de modifier le lexique et/ou la syntaxe que si les analyseurs étaient écrits à la main. En revanche, il faut que le lexique et la syntaxe du langage puissent être décrits à l'aide des formalismes utilisés par les outils de génération, en l'occurence les expressions régulières et une certaine classe de grammaires algébriques (LL, SLR, LR(0), LALR ou LR(1) selon les outils). De plus, une partie non négligeable du compilateur doit être écrite "à la main" : il s'agit de l'analyseur syntaxique et du générateur de code (il existe cependant certains outils de génération qui peuvent également être utilisés pour ces composants).

Certains générateurs d'interfaces sont tout à fait comparables aux générateurs de compilateurs : ils prennent une description textuelle d'une partie de l'application interactive et produisent du code qui implémente cette partie. C'est la cas par exemple de UIL qui produit du code pour la boîte à outils MOTIF à partir d'une description de la hiérarchie des widgets et des valeurs de leurs attributs et fonctions de rappel. Un outil qui prend la description d'une machine à états pour produire la fonction de traitement des événements d'une classe de widgets est un autres exemple de générateur.

D'autres générateurs d'interfaces sont eux-mêmes des outils interactifs : ils permettent de construire interactivement la description d'une partie d'une application interactive. Ces générateurs sont composés de trois parties :

- un *éditeur interactif* qui permet de construire, par manipulation directe, la présentation de l'interface et de spécifier au moins une partie des fonctions de rappel. Cet éditeur dispose en général d'un mode "test" qui permet de faire fonctionner l'interface construite.
- un *compilateur* qui permet de traduire la description de l'interface produire par l'éditeur en un module qui peut être compilé.

- un *module d'exécution* ou "run-time" qui doit être inclus dans l'application finale. Ce module contient le corps d'un certain nombre de fonctions génériques qui sont utilisées par le code généré par le compilateur. Ce module contient également les fonctions qui permettent de charge une description de l'interface telle qu'elle est générée par l'éditeur.

Il est ainsi possible de produire une application interactive de deux façons : soit l'application charge la description de l'interface produite par l'éditeur (figure A), soit elle intègre le code produit par le compilateur (figure B).

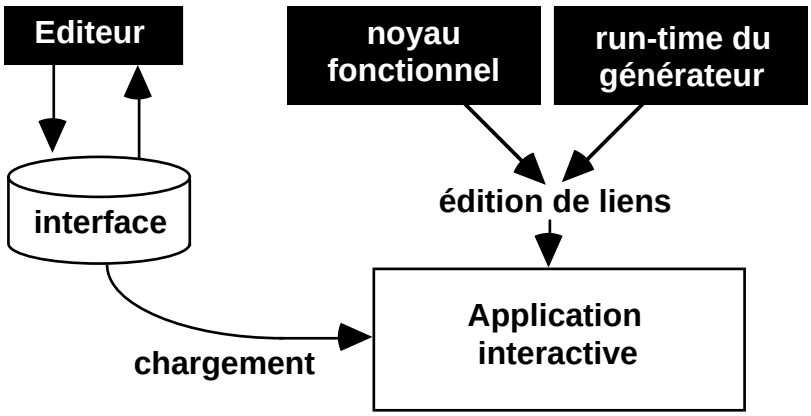

figure A : schéma d'un générateur d'interface interactif.

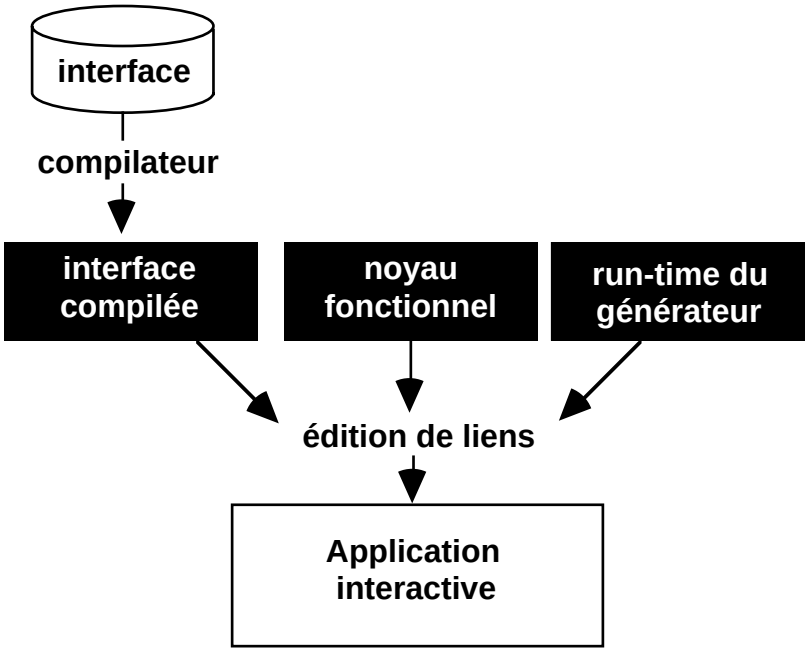

figure B : génération de l'application finale

La première solution est utile dans la phase de développement car il n'est pas nécessaire de recompiler l'application complète à chaque modification de l'interface : on teste l'interface sous l'éditeur, puis de façon plus complète avec l'application interactive, sans passer par la compilation. La deuxième solution permet de produire l'application en phase finale de développement, en phase finale de développement. En effet cette application est plus efficace que celle produite avec la première solution puisque le code de l'interface est compilé. Elle est

également autonome puisqu'elle ne dépend pas d'un fichier séparé contenant la description de l'interface.

L'intérêt de ces générateurs est double. D'une part, il est plus facile de créer la présentation de l'interface par manipulation directe que par une description textuelle. En particulier, on peut voir immédiatement l'effet des différents attributs de présentation et des paramètres de placement. D'autre part, une partie de l'interface peut être testée immédiatement. En effet même si le noyau fonctionnel n'est pas accessible depuis l'éditeur, on dispose en général d'un langage interprété qui permet d'écrirer le corps des fonctions de rappel. En mode test, l'éditeur active les fonctions de rappel ce qui permet de tester les enchaînements des boîtes de dialogues, l'activation et la désactivation des items de menus, etc.

L'inconvénient des générateurs d'interface est de se focaliser sur la partie présentation. Le lien avec le noyau fonctionnel peut représenter un travail considérable pour lequel le générateur n'est d'aucune aide. Le générateur ne fournit en général peu ou pas d'assistance pour des fonctions génériques comme le copier-coller, la gestion des documents (sauvegarde, rechargement, etc.) et les parties dynamiques de l'interface (par exemple un menu qui contient la liste des fenêtres actives, ou une présentation qui change en fonction de la taille de la fenêtre). Enfin la plupart des générateurs d'interface sont construits au-dessus d'une boîte à outils et sont donc limités par la palette de widgets fournis par cette boîte à outils. Certains générateurs sont cependant extensibles et peuvent être reconfigurés pour accepter de nouvelles classes de widgets.