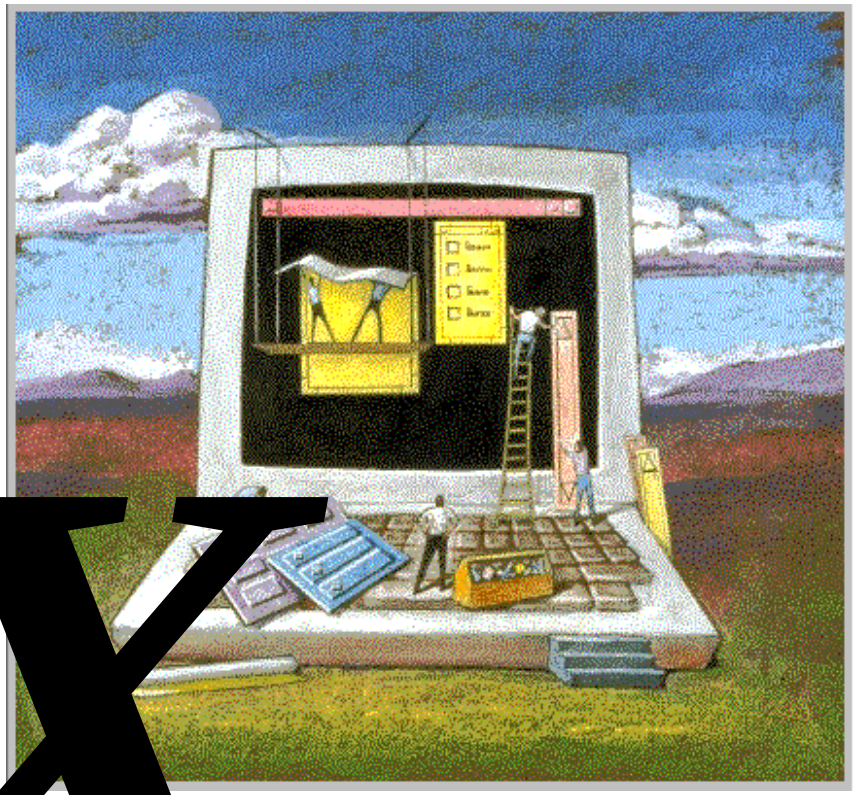
Système X-Window

Concepts et Librairies
Xlib, Xt et Xm

Franck Diard, Michel Buffa, François Abram, 1998
diard@i3s.unice.fr, http://www.essi.fr/~diard
abram@i3s.unice.fr, http://www.essi.fr/~abram
V 1.4. Mars 1998.



X

1. XWindow

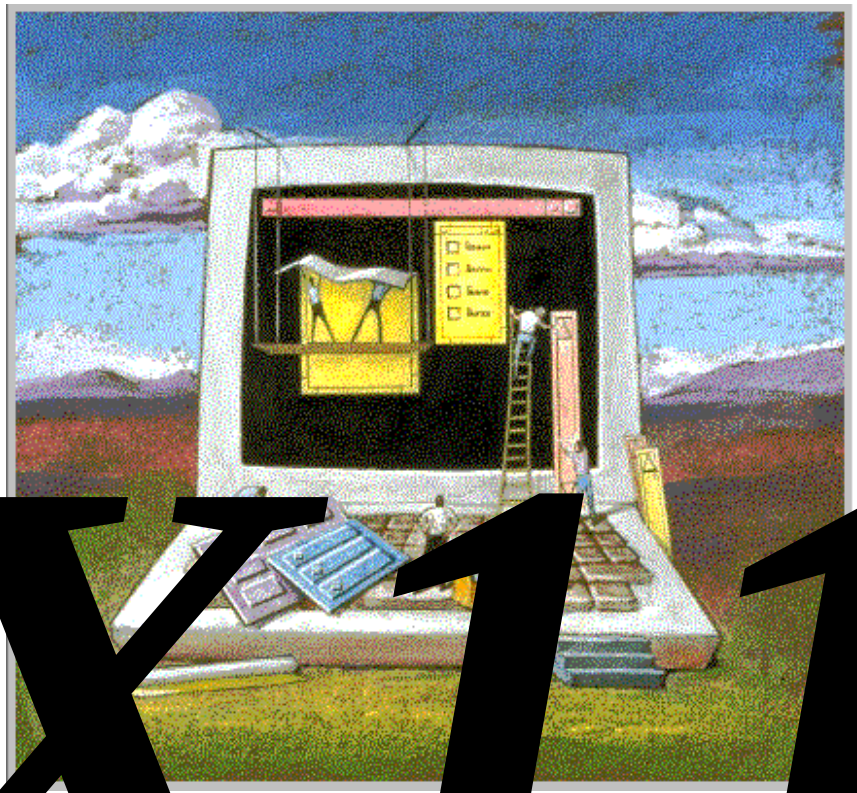
2. Xt Intrinsics

3. Motif

X Window

Une introduction

Franck Diard, Michel Buffa, François Abram, 1998
diard@i3s.unice.fr, http://www.essi.fr/~diard
abram@i3s.unice.fr, http://www.essi.fr/~abram
V 1.4. Mars 1998.



X11

1. X11: Concept et Architecture.

Le package logiciel X-Window (appelé couramment X) est un système permettant aux programmeurs de développer des applications graphiques portables et distribuées. Cette section introduit le concept de X ainsi que ses principaux avantages et inconvenients.

1.1. Qu'est ce que X?

X a été défini pour répondre à ces caractéristiques:

- Indépendant vis-à-vis du matériel.
- Le système est transparent à travers le réseau.
- Plusieurs applications doivent pouvoir tourner en même temps.
- "Policy-less": on peut implémenter tout style d'interaction.
- Hiérarchie de fenêtres retaillables.
- Fenêtres recouvrables, gestion des parties cachées.
- Une application peut ouvrir N fenêtres, pas forcement sur la même machine.
- Haute performance pour le texte, le graphique et les images.
- Extensibilité.

Le système X-Window permet de distribuer les informations graphiques. Les résultats graphiques calculés par une machine distante peuvent être visualisés sur une machine locale. Un programme peut ainsi envoyer ses résultats graphiques sur une autre station et récupérer les événements des périphériques d'interface tels que le clavier et la souris de la station distantes.

La machine locale est par exemple une station de travail (SGI, HP, Sun...), un Macintosh, un PC ou un terminal X.

La machine distante est le plus souvent une machine plus puissante sur laquelle le programme est exécuté (VAX, CRAY, machine parallèle):

.

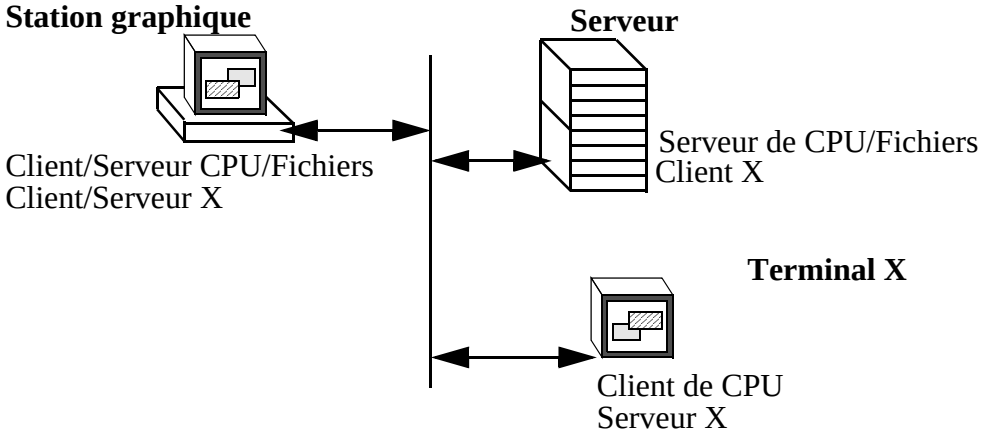


Figure 1: **Affichage déporté des informations graphiques.**

Il existe donc un moyen pour coder toutes les opérations graphiques ainsi que l'interactivité avec des périphériques, pour pouvoir les échanger entre les machines. Ces informations passent par un protocole et ensuite par un médium tel que le réseau.

1.2. Commandes graphiques.

Le concept le plus fondamental de X est celui des commandes graphiques qui sont diffusées selon le modèle client-serveur.

Tout ceci prend un sens dans le contexte du réseau. On veut pouvoir bénéficier de l'interface graphique d'un programme s'exécutant à distance.

A l'époque des terminaux "textes" (VT100), le rafraîchissement d'un écran coûtait peu: 24 lignes par 80 colonnes avec un caractère codé sur un octet, soit à peu près 2000 octets par écran. Si l'on veut passer en mode graphique, la définition doit être supérieure. Pour un terminal graphique disposant d'une résolution 640 par 480 pixels, un octet codant ici la couleur du pixel, on obtient 307200 octets. Plutôt que d'envoyer toutes les information graphiques de l'écran, X propose d'envoyer uniquement la commande graphique (trace de ligne, affichage de texte,...) et de le faire effectuer par la machine distante (console).

L'accès à la carte graphique sur la console est complètement encapsulé

par X11. On ne peut pas aller modifier le frame-buffer du display sans passer par X11 (il y a des exceptions, nous y reviendrons).

On distingue donc deux notions essentielles: le CPU, qui exécute le code du programme et ce que l'on appelle un "display".

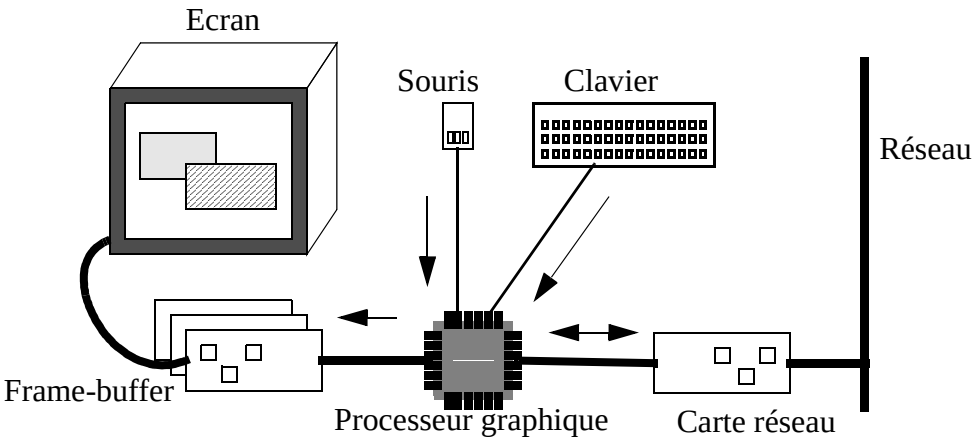


Figure 2: **Notion de display.**

Un display (Voir Fig.2) est composé d'un processeur capable d'effectuer les commandes graphiques (ce processeur peut être dédié - c'est le cas d'un terminal - ou générique - celui de la station de travail), d'un clavier, d'une souris et un ou plusieurs couples carte graphique/ moniteur.

La modification du frame-buffer du display est effectuée en local sur ordre de la machine distante.

1.3. Pourquoi pas X?

Bien que X soit implémenté de manière efficace, le concept même de l'encodage des commandes graphiques dans un protocole est assez lourd. Le deuxième inconvénient consiste en sa généricité. Les commandes graphiques ne doivent pas intégrer des optimisations spécifiques au matériel pour pouvoir être portées sur toutes les machines (certaines optimisations existent, nous y reviendrons).

Ces deux inconvénients entraînent pour les machines un surcoût en temps machine et en mémoire. X n'était donc pas destiné en priorité aux micros. Mais il était en fait seulement en avance. Les micros d'aujourd'hui le supportent facilement. Une configuration minimale se situe vers 8 megas de RAM et 20-100 megas d'espace disque.

1.4. Historique de X.

En 1984, le Massachusetts Institute of Technology (MIT) démarre le projet Athena. Le but était de prendre tout l'assortiment des kits graphiques de plusieurs constructeurs (évidemment incompatibles) et de développer un réseau de stations pouvant servir au télé-enseignement. Le MIT voulait créer un environnement graphique totalement indépendant du matériel.

En 1986, les constructeurs ont commencé à contribuer au développement de X. En 1988, le MIT officialise la version 11 release 2. La plus récente des versions est la version 6 (les versions 4 et 5 sont les plus répandues).

Les constructeurs BULL,DEC, HP, IBM et SUN sont réunis en consortium pour soutenir le MIT. Le consortium gère l'évolution de X et garantie sa pérennité.

Des informations sur le consortium X sont disponibles sur le site: ***http://www.x.org/ftp/pub/DOCS/XConsortium***.

Actuellement, X a confirmé son succès: de nombreuses stations de travail Unix le proposent en standard et il est disponible sur les micros tels que Mac, PC, Amiga...

1.5. Culture générale.

Le succès de X est dû à UN homme: Robert Scheifler du MIT. Il a su définir clairement les objectifs de X et les maintenir. Il a créé une philosophie de distribution (gratuité, source fourni).

La politique d'ouverture d'X11, grandement favorisée par le succès d'Internet, à créé un énorme précèdent: le choix unanime de toute la profession sur une technologie "domaine public". Le paysage informatique en a été entièrement bouleverse, créant la mode des systèmes ouverts.

1.5.1. Distribution.

X11 est un package logiciel totalement gratuit, mais copyrighté. Il est disponible à ***ftp://lcs.export.mit.edu***. Le source de toutes les couches (voir ci-après) sont fournis. Une nouvelle version du système est produite environ tous les deux ans (actuellement la révision 6). Mais des versions corrigées apparaissent sous forme de patches à installer.

Il y a une très forte dynamique autour de X dans le monde universitaire et de nombreux programmes d'application sont distribués gratuitement par les auteurs. Ces programmes sont connus sous forme de contribs. La distribution X11 du MIT intègre en standard un grand nombre de contribs, ce qui rend son exploration très intéressante.

Un autre site propose des archives complètes des logiciels X11: ***ftp://ftp.x.org***.

1.5.2. Bibliographie.

Des bibliographies et des recueil d'informations techniques électroniques sont disponibles sur les serveurs:

- ***ftp.x.org:/contrib/docs/Xbibliography.ps***.
- ***gatekeeper.dec.com:/pub/X11/R5-contrib/Xbibliography***.
- ***landru.unx.com:/pub/X11***.

Quelques livres de référence pour X:

- Jones, Oliver, Introduction to the X Window System, Prentice-Hall, 1988, 1989.
- Scheifler, Robert, and James Gettys, with Jim Flowers and David Rosenthal, "X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, X Version 11, Release 5, Third Edition," Digital Press, 1992.
- Nye, Adrian, "Xlib Programming Manual, Volume 1" and "Xlib Reference Manual, Volume 2," O'Reilly and Associates.
- Mansfield, Niall. "The X Window System: A User's Guide," Addison-Wesley, 1989.
- Quercia, Valerie and Tim O'Reilly. "X Window System User's Guide," O'Reilly and Associates.

1.5.3. Cours X11 sur WWW.

Des cours au format hyper-text sont disponibles sur Internet.

- ***http://www.cs.curtin.edu.au/units/cg252-502/src/notes/html***.
- ***http://www.cms.dmu.ac.uk:80/~aug/FastTrack***.

1.5.4. Sources d'informations.

Le forum de *news comp.window.x* permet des échanges sur tous les sujets relatifs à X, les développeurs du consortium l'utilisent pour diffuser les

informations importantes.

D'autres forums de news sont très utiles: ***comp.windows.x***, c***omp.windows.x.apps***, ***comp.windows.x.announce***.

La FAQ ***comp.windows.x*** est également une excellente source d'informations. On peut la télécharger à ***ftp://ftp.inria.fr/faq***.

D'autres sont disponibles à ***ftp://ftp.x.org/contrib/faqs*** et ***ftp://rtfm.mit.edu.***

1.5.5. Concurrents.

Le concurrent le plus sérieux de X11 fut le défunt NEWS développé par SUN.

La force de X11 par rapport à Windows est son aspect client-serveur (Windows n'a pas en standard de protocole graphique transportable par réseau).

Windows, même dans ses versions les plus performantes (NT, 95) ne propose pas de système graphique distribué.

NeXTSTEP a presenté pendant plusieurs années une sérieuse alternative à X. Proposant le concept du display-poscript en standard et des fonctions graphiques très puissantes adaptées au matériel NeXT (video-live en PIP, move-opaque 32 bits avec alpha-channel), NeXTSTEP fascinait la communauté. De plus, le kit de developpement (Interface-Builder) possédait une conception objet très poussée qui permettait des temps de développement records. Malheureusement la firme NeXT a arreté la fabrication des machines NeXT. Elle continue le developpement de NeXTSTEP sur des machines génériques comme les PC, Dec et Sun, mais pour des resultats nettement moins spectaculaires.

1.6. Modèle client-serveur.

X repose sur un modèle client-serveur:

Un serveur peut être accédé par plusieurs clients et un client peut accéder à plusieurs serveurs.

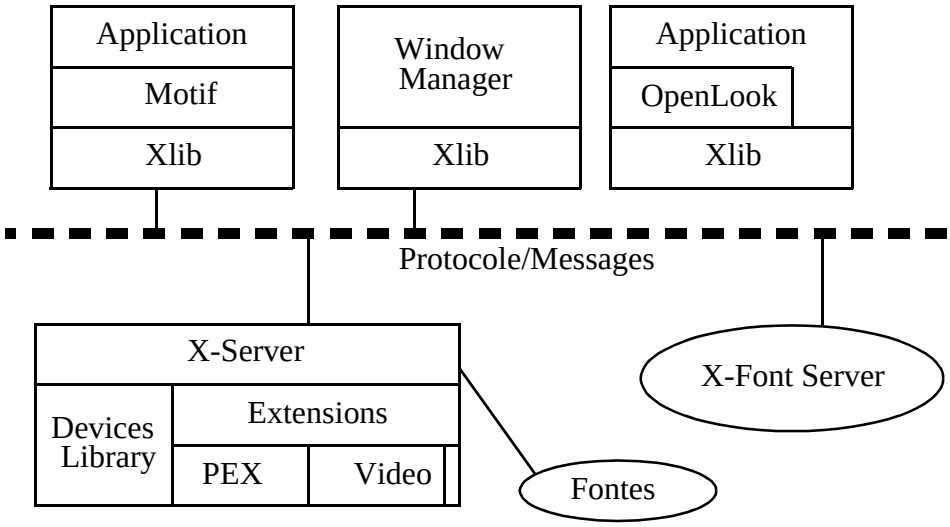


Figure 3: **Modèle Client-Serveur.**

Les clients passent par la couche de protocole Xlib pour être compatibles avec le serveur.

1.6.1. Le serveur X.

Le serveur X11 est le logiciel qui contrôle l'accès au hardware (clavier, frame-buffer, souris,...). Il est donc forcement logé dans la console (terminal, station): on ne peut pas le lancer en remote.

Exemple:

```
local_host:> rlogin another_host
Passwd:
another_host:> xinit
XServer Error, can't access to frame-buffer.
```

Toutes les applications X11 (clientes) adressent des requêtes au serveur X11 qui peut être en local ou sur une autre machine (mode remote).

On ne peut lancer qu'un seul serveur par console sinon:

```
Fatal server error:
Server already running !!!
xinit:  Server error.
```

Le serveur interprète les requêtes des clients et les exécute.

Il signifie les actions de l'utilisateur (mouvements souris, frappe clavier) en envoyant des messages aux clients.

1.6.2. Identification des displays.

Un display X11 est identifié par une adresse Internet (Ex: 157.169.25.4) suivie de ":0". Une machine Internet possède un numéro unique spécifié par l'association InterNIC (lors du raccordement à Internet). Le nombre "0" désigne la carte graphique gérée par la station.

Bien sûr on peut utiliser les noms et les domaines à la place du numéro IP (il faut pour cela bénéficier du DNS ou des pages jaunes pour un accès complet au réseau Internet).

DNS: Comment ca marche?

Une machine Internet possède un numéro IP unique spécifié par l'association InterNIC (lors du raccordement à Internet). A chaque fois qu'une machine veut transcrire une adresse du style machine.organisation.pays en numéro IP, elle s'adresse à un serveur DNS. On spécifie à une machine le numéro IP de plusieurs serveurs DNS dans un fichier de configuration (variable selon les machines).

Exemple:

```
diard@jessica:/etc> cat resolv.conf
nameserver 157.169.25.4
nameserver 157.169.25.10
nameserver 157.169.25.1
```

Par exemple essi2.essi.fr:0 peut être utilisé. Pour l'accès à une machine d'un sous-réseau local, le nom de la machine suffit (essi2:0).

Certaines machines peuvent posséder plusieurs cartes graphiques (par exemple une carte 24 bits pour l'affichage et une autre carte 8 bits pour le développement). Il faut donc donner le numéro de la carte dans laquelle on veut afficher.

Le display destinataire de l'application X11 est identifié par diffèrent niveau de priorité.

Sur la ligne de commande dans le shell, on peut spécifier (dans les programmes X11 correctement écrits):

```
local_host:> xclock -display another_host:0
```

Si aucun argument n'est spécifié, la variable d'environnement shell est exploitée.

Pour la positionner:

local_host:> setenv DISPLAY essi2.essi.fr:0

local_host:> export DISPLAY

L’interface graphique des applications X11 à partir de ce moment sera redirigé sur le display essi2.essi.fr:0.

2. Composition de X.

X est bâti sur un système de couches.

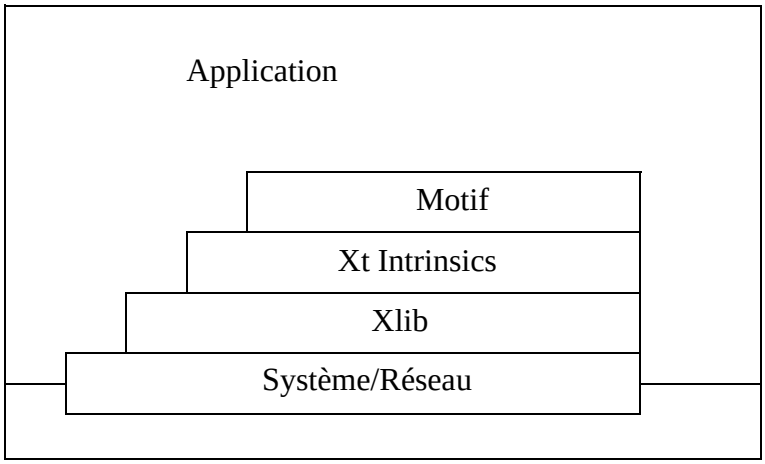


Figure 4: **Couches de X.**

L'interface la plus basse est l'interface des sockets, ce sont des voies de communication qui permettent de véhiculer des paquets d'octets entre de machines du réseau. L'aspect communication de X est entièrement sous-traité à Unix.

2.1. Le protocole X.

Toutes les transactions (requêtes du client - réponses du serveur) X11 sont encodées dans un flux d'octets qui passe dans la connexion entre le client et le serveur. Nous reviendrons sur le protocole utilisé (nous verrons plus tard que dans le cas d'un lancement d'application X11 en local, certaines librairies permettent de court-circuiter la couche réseau de X11 pour accélérer le traitement des commandes).

La couche supérieure gère le protocole. Les messages X sont de quatre types:

- Les requêtes d'un client vers un serveur.

- les réponses du serveur aux requête des clients.
- la signification par le serveur des événements du type:
- une touche du clavier a été enfoncée.
- on a clique sur un bouton de la souris.
- envoi par le serveur d'éventuelles erreurs.

Le protocole X constitue la base de l'échafaudage X, sa pérennité est garantie par le consortium.

Le protocole X ne suppose aucun protocole reseau et en depend ni du réseau (***RS232***, ***ETHERNET***, ***TOKEN RING***…) ni du systeme (UNIX,VMS,MSDOS). Le protocole X est performant: "le client doit pouvoir suivre la souris", un aller-retour (round-trip) entre 5 à 10 millisecondes (200 kbits/seconde).

2.2. Xlib.

La Xlib est une interface de programmation du protocole X, elle est décrite en langage C.

Elle permet entre autres, au travers de primitives simples de tracer des lignes ou de récupérer les caractères entrés au clavier. Ce sont des fonctions de base, des commandes graphiques. Elle est installée sous le nom ***/usr/X11/lib/libX11.a***.

C'est cette librairie qui encode dans le protocole les commandes graphiques.

2.3. X-ToolKit.

La création d'un interface graphique est très fastidieuse si l'on doit gérer soi-même le fait que la souris cliquée à tel endroit correspond à telle action... Une couche permet de s'affranchir de la gestion des icônes, boutons, menus....

Une librairie destinée à simplifier la programmation de gadgets a été crée: X Toolkit Intrinsics. Connue sous le nom de Xt, la librairie introduit le concept de widget. Un widget est une structure de donnée accompagnant une fenêtre et permettant de qualifier aisément certains paramètres (taille, fond, contours...), mais aussi de dicter plus facilement la réaction de l'interface. Une hiérarchie de widgets est introduite par Xt, proposant ainsi un comportement objet qui se révèle fort utile. La librairie Xt est installée dans ***/usr/X11/lib/libXt.a***.

2.4. Toolbox.

Le dernier niveau est appelé la boite à outils (toolbox). Les plus connues sont les Athena's Widgets (gratuit), OpenLook (gratuit) et Motif (payant). Ces boites à outils définissent des widgets de type bouton de commande, boite de dialogue, barre de défilement... Les boites à outils introduisent des notions de présentation des applications (le fameux look and feel). Motif semble actuellement avoir pris le dessus, bien qu'il soit payant.

2.5. Le window-manager.

Le "window-manager" est un élément clé de X11. C'est une application X11 qui permet de gérer les fenêtres qui sont ouvertes sur le display. Appelé communément "environnement", il propose des menus, des icônes pour positionner, redimensionner, fermer les fenêtres connectées sur le display où il tourne.

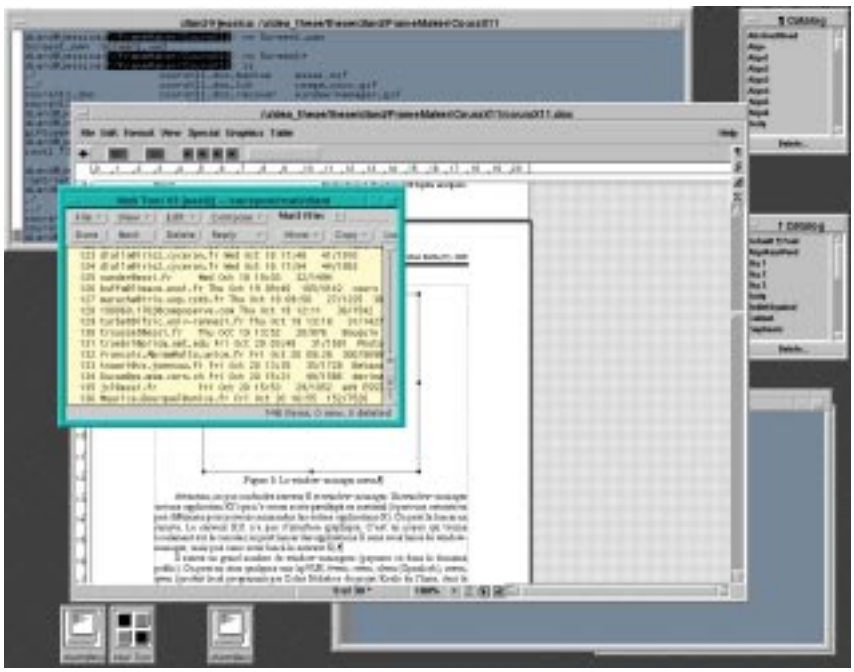


Figure 5: **Le window-manager mwm.**

Attention, ne pas confondre serveur X et window-manager. Un window-manager est une application X11 qui n'a aucun accès privilégié au matériel (à part une autorité un peu différente pour pouvoir commander les autres applications X). On peut le lancer en remote. Le serveur X11 n'a pas

d'interface graphique. C'est un noyau qui tourne localement sur la console (on peut lancer des applications X sans avoir lancé de window-manager, mais pas sans avoir lancé le serveur X).

Il existe un grand nombre de window-managers (commerciaux ou dans le domaine public). On peut en citer quelques uns: hpVUE, fvwm, ctwm, olwm (Openlook), mwm, gwm (produit local programmé par Colas Nohaboo du projet Koala de l'Inria, dont la principale particularité est son extensibilité: ***http://www.inria.fr/koala/colas***).

Ils offrent tous à peu près les mêmes caractéristiques, l'ergonomie et l'extensibilité les différencient.

Certains (hpVUE, fvwm, olvwm…) implémentent le concept d'écran virtuel.

Ecrans virtuels: Comment ca marche?

Le window-manager propose plusieurs écrans auxquels on peut accéder par une fenêtre spéciale. Il stocke pour chaque écran virtuel les identifieurs des fenêtres appartenant à celui-ci (une simple table). Lorsqu'on accède à un écran, le window-manager "mappe" les fenêtres concernées et "unmappe" les autres. Ce mécanisme est purement logique (pas de transferts ni d'allocations de mémoire vidéo supplémentaires).

Devant la diversité des window-managers, un nouvel environment de développement vient de voir le jour. Il est déjà porté sur toutes les plates-formes des constructeurs du consortium X. Il s'appelle CDE (Common Desktop Environment).

2.6. Problèmes liés au réseau.

Le fait d'utiliser des connexions réseau apporte un certain nombre de contraintes.

En ce qui concerne la fiabilité, la connexion peut être poupée à tout moment. On suppose qu'il n'y a pas de perte de paquets ni de duplication (TCP/IP s'en charge), la seule erreur est la perte de connexion.

Des décisions sur les formats de données doivent être prises, car le réseau permet de mélanger des machines de différents constructeurs

Les identifieurs X11 (de fenêtres, de fontes…) sont codés sur 32 bits et les coordonnées sur 16 bits.

Il y a deux façons de coder un entier en binaire: "little endian" (Intel) et "big endian" (les autres constructeurs). X doit donc prendre en compte le fait que le client code les entiers différemment. Pour cela deux ports sont ouverts, un pour chaque famille.

Le protocole ne doit pas être trop lourd pour permettre les connexions à travers un modem (peu raisonnable) ou une ligne RNIS (mieux, 64 Kbauds).

3. Configuration de XWindow

3.1. Démarrer avec X11

3.1.1. Où se trouvent les commandes X-Window ?

Les commandes X-Window se trouvent en général dans le répertoire ***/usr/bin/X11***. Il convient d'inclure celui-ci dans le mécanisme des règles de recherche. Pour les Unixiens utilisateurs de csh ou tcsh, il est vivement conseillé d'inclure la commande ci-dessous dans le ***.cshrc***.

```
set path = ($path /usr/bin/X11)
```

3.1.2. Les autorisations de connexions

En utilisant un système de fenêtrage, l'utilisateur est amené à ouvrir des fenêtres (sessions) sur différentes machines. Pour que cela soit possible un fichier appelù ***.rhosts***, se situant dans le répertoire d'accueil (home dir) de l'utilisateur, donnera les autorisations nécessaires. L'exemple ci-dessous montre le contenu du fichier .rhosts d'un utilisateur toto s'autorisant des sessions automatiques depuis et vers les machines mach1, mach2 et mach3 du domaine org.fr (son répertoire d'accueil étant commun aux trois machines) :

```
mach1.org.fr toto mach2.org.fr toto mach3.org.fr toto
```

3.1.3. Le fichier de configuration du serveur X

Un serveur X a besoin de connaître les préférences de l'utilisateur concernant un certain nombre de paramètres. C'est le rôle du fichier ***.Xdefaults*** situé dans le répertoire d'accueil de l'utilisateur. L'exemple ci-dessous est une configuration minimum pour démarrer :

```
! Ressources generales par defaut.
!
*BitmapFilePath : /usr/include/X11/bitmaps
```

```
*font : 9x15

!
! Ressources de Xterm
!
XTerm.termName : xterm
XTerm.geometry : 80x34
XTerm.SunFunctionKeys : on
XTerm.VT100.boldFont : 6x13bold
XTerm.VT100.Font : 6x13
XTerm.VT100.font1 : 5x8
XTerm.VT100.font2 : 6x9
XTerm.VT100.font3 : 9x15
XTerm.VT100.font4 : 10x20
XTerm.VT100.ScrollBar : on
```

3.1.4. Fichier de configuration de twm

Les gestionnaires de fenêtres en général permettent à l'utilisateur de personnaliser un certain nombre de paramètres tels que les répertoires d'icônes, les polices utilisées dans les bandeaux de fenêtres, les menus déroulants. Dans le cas de ***twm*** un fichier appelé .twmrc joue ce rôle, il est situé dans le répertoire d'accueil de l'utilisateur. L'exemple suivant est une configuration minimum pour démarrer.

```
WarpCursor
BorderWidth 5
TitleFont "9x15"
MenuFont "9x15"
IconFont "9x15"
ResizeFont "9x15"
IconManagerFont "9x15"
NoTitleFocus
Zoom
ShowIconmanager
#Button = KEYS : CONTEXT : FUNCTION
Button1 = : root : f.menu "button1"
Button2 = : root : f.menu "button2"
Button3 = : root : f.menu "button3"
Button1 = m : window : f.menu "button1"
Button2 = m : window : f.menu "button2"
Button3 = m : window : f.menu "button3"
Button1 = m : title : f.menu "button1"
Button2 = m : title : f.menu "button2"
Button3 = m : title : f.menu "button3"
Button1 = : icon : f.iconify
Button2 = : icon : f.move
Button1 = : title : f.raise
Button2 = : title : f.move


IconDirectory "/usr/include/X11/bitmaps"
menu "button1"
{
    "Rafraichir ecran" f.refresh
    "Nouveau .twmrc" f.twmrc
    "Destruction fenetre" f.destroy
    "Nouveau .Xdefaults" !"xrdb -load .Xdefaults &"
}
menu "button2"
{
    "Actions speciales" f.title
    "UnFocus" f.unfocus
    "UnIconify" f.iconify
    "Move Window" f.move
    "Resize Window" f.resize
```

```
    "Raise Window" f.raise
    "Lower Window" f.lower
    "Focus on Window" f.focus
    "Zoom" f.fullzoom
    "Kill twm" f.quit
}
menu "button3"
{
}
```

3.1.5. Fichier de configuration de mwm

```
!!
!! Default Window Menu Description
!!

Menu DefaultWindowMenu
{
      Restore        _R      Alt<Key>F5      f.restore
      Move           _M      Alt<Key>F7      f.move
      Size           _S      Alt<Key>F8      f.resize
      Minimize       _n      Alt<Key>F9      f.minimize
      Maximize       _x      Alt<Key>F10     f.maximize
      Lower          _L      Alt<Key>F3      f.lower
      no-label                               f.separator
      Close          _C      Alt<Key>F4      f.kill
}


!!
!! Key Binding Description
!!

Keys DefaultKeyBindings
{
      Shift<Key>Escape        window|icon             f.post_wmenu
      Alt<Key>space           window|icon             f.post_wmenu
      Alt<Key>Tab             root|icon|window        f.next_key
      Alt Shift<Key>Tab       root|icon|window        f.prev_key
      Alt<Key>Escape          root|icon|window        f.circle_down
      Alt Shift<Key>Escape    root|icon|window        f.circle_up
      Alt Shift Ctrl<Key>exclam root|icon|window      f.set_behavior
      Alt<Key>F6              window                  f.next_key transient
      Alt Shift<Key>F6        window                  f.prev_key transient
      Shift<Key>F10           icon                    f.post_wmenu
!     Alt Shift<Key>Delete    root|icon|window        f.restart
}

!!
!! Button Binding Description(s)
!!

Buttons DefaultButtonBindings
{
      <Btn1Down>      icon|frame      f.raise
      <Btn3Down>      icon|frame      f.post_wmenu
      <Btn3Down>      root            f.menu  DefaultRootMenu
}

Buttons ExplicitButtonBindings
{
      <Btn1Down>      frame|icon      f.raise
      <Btn3Down>      frame|icon      f.post_wmenu
      <Btn3Down>      root            f.menu  DefaultRootMenu
!     <Btn1Up>        icon            f.restore
      Alt<Btn1Down>   window|icon     f.lower
!     Alt<Btn2Down>   window|icon     f.resize
!     Alt<Btn3Down>   window|icon     f.move

}
```

```
Buttons PointerButtonBindings
{
      <Btn1Down>      frame|icon      f.raise
      <Btn3Down>      frame|icon      f.post_wmenu
      <Btn3Down>      root            f.menu  DefaultRootMenu
      <Btn1Down>      window          f.raise
!     <Btn1Up>        icon            f.restore
      Alt<Btn1Down>   window|icon     f.lower
!     Alt<Btn2Down>   window|icon     f.resize
!     Alt<Btn3Down>   window|icon     f.move
}

!!
!!  END OF mwm RESOURCE DESCRIPTION FILE
!!
```

3.1.6. Démarrer une session X sur une station

Dans la livraison du MIT on trouve la commande startx qui lance le chargement d'un serveur X et l'initialisation d'un certain nombre de clients X. La commande ***startx*** recherche la présence d'un fichier appelé ***.xinitrc*** dans le répertoire d'accueil de l'utilisateur, il contient les applications à lancer en ouvrant la session. Si le fichier .xinitrc n'existe pas, des applications trouvées dans le fichier ***/usr/lib/X11/xinit/xinitrc*** seront alors lancées par défaut. Toutes les applications figurant dans un ***.xinitrc*** doivent être lancées en arrière plan sauf la dernière car, quand celle-ci se terminera, la session X toute entière prendra fin. Dans l'exemple suivant la commande xquit permet de quitter la session :

```
xhost + > /dev/null &
twm &
xconsole -iconic &
xclock -geometry 120x120+1017+6 &
xterm -ls -sb &
xquit -geometry 60x72+1078+776
```

3.1.7. Démarrer une session X sur un terminal X

Il y a plusieurs solutions pour initialiser une session X sur un terminal X. Une des méthodes généralement utilisée est l'établissement préalable d'une session telnet entre le terminal et la machine hôte de référence. Une fois cette connexion obtenue, l'utilisateur pourra lancer une procédure de commandes contenant les applications à démarrer comme dans l'exemple suivant :

```
#!/bin/csh
xhost + > /dev/null &
xrdb ~/.Xdefaults & twm & xclock -geometry 120x120-0+0 &
xterm -ls -sb &
```

Il est également possible de configurer le terminal X afin qu'il utilise XDM. Dans ce cas une bannière de connexion invite l'utilisateur à se présenter (nom et mot de passe). En cas de succès des applications sont lancées automatiquement.

L'initialisation du terminal X, ainsi que la fin de la session X, sont propres à chaque type de terminal et ne sont donc pas mentionnées ici.

3.2. Les Ressources

3.2.1. Introduction

Dans l'environnement X-Window le mot ressource est utilisé pour spécifier des valeurs qui sont utilisées par les applications, et qui peuvent être facilement modifiées par l'utilisateur. Dans le principe il convient de distinguer les ressources standards (communes à toutes les applications) et les ressources particulières (inventées par le créateur de l'application).

3.2.2. Les ressources standards

Toute application s'appuyant sur la bibliothèque Xt Intrinsics hérite des ressources suivantes :

- font pour spécifier les polices de caractères utilisées ;
- geometry pour indiquer le positionnement et la taille des fenêtres ;
- background pour qualifier le fond d'écran d'une fenêtre ;
- borderwidth pour donner la taille en pixels du contour des fenêtres ;
- display pour indiquer le nom de la station qui héberge le serveur X ciblé ;
- …

Ces ressources sont appelées les ressources standards.

3.2.3. Les ressources des applications

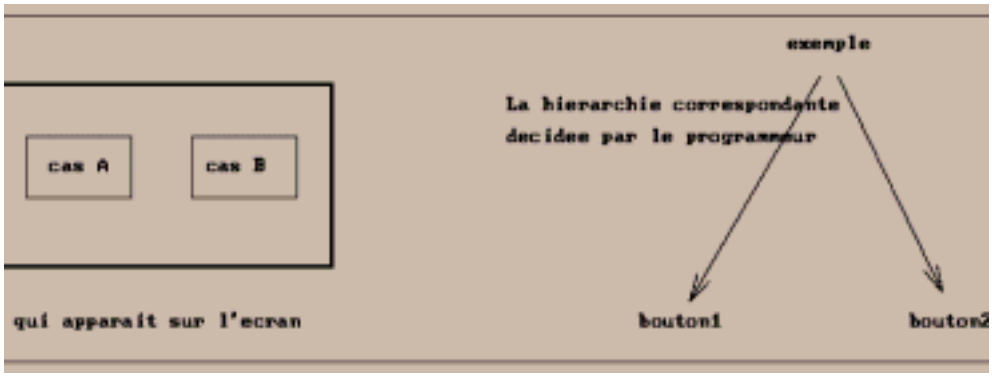
En plus des ressources standard, les programmeurs définissent des ressources propres à leurs applications. Ces ressources servent à personnaliser les utilisations, l'utilisateur ayant la maîtrise de nombreux paramètres. Ainsi l'application xterm fournie par le consortium X prévoit pour l'utilisateur la possibilité de spécifier ses choix pour les ressources suivantes :

- scrollBar pour indiquer si l'utilisateur désire ou non la présence d'une barre de défilement sur le côté de sa fenêtre ;
- saveLines pour mentionner le nombre de lignes sur lequel portera l'effet d'une barre de défilement ;
- …

Le manuel d'une application X-Window contient toujours la liste des ressources propres à cette application ainsi que leurs valeurs par défaut.

3.2.4. Le nommage des ressources

Les ressources d'une application X-Window sont associées aux fenêtres qui la composent sous la forme d'une arborescence respectant les imbrications des fenêtres composantes. Les fenêtres composant une application ont un nom (utilisable à l'extérieur du programme) donné par le programmeur, il doit figurer dans le manuel de l'application. L'application exemple crée une fenêtre contenant deux sous-fenêtres qui sont des boutons de commandes.



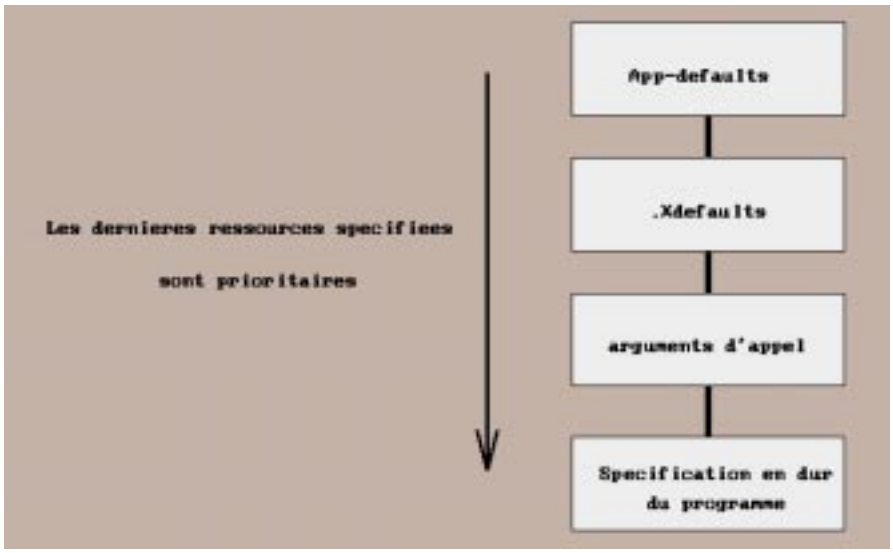
Pour personnaliser l'application exemple l'utilisateur pourra procéder de la fa,con suivante :

exemple.bouton1.font : 6x13
exemple.bouton1.borderWidth : 5
exemple.bouton1.label : Com A
exemple.bouton2.font : 9x15
exemple.bouton2.borderWidth : 5
exemple.bouton2.label : Com B

L'utilisateur a ici utilisé les ressources standard font et borderWidth , il demande une police de caractères spécifique pour chaque bouton de commande. Par contre pour le bord de la fenêtre il stipule la même valeur pour bouton1 et bouton2 et aurait pu utiliser la notation suivante :

exemple*borderWidth : 5
!! ou
Exemple*borderWidth : 5

La notation * permet de s'affranchir de tout ou partie de la hiérarchie des fenêtres. L'exemple ci-dessus créerait toutes les fenêtres de l'application avec un bord de 5 pixels quel que soit leur niveau d'emboîtement. La seconde ligne commen,cant par Exemple cible toutes les applications de la classe Exemple dont exemple est un membre. C'est une notation fréquemment utilisée.



Dans le jargon X-Window on dit que exemple est une application et que exemple.bouton2 est un widget, label étant une ressource propre à

l'application exemple. Les (sous) fenêtres ne sont que la partie visible des widgets, un widget est une structure de données accompagnant une fenêtre et permettant de la qualifier à l'aide de ressources.

3.2.5. Où spécifier des ressources ?

Il y a divers endroits pour spécifier les ressources des applications X-Window. L'auteur d'une application utilisera un fichier de ressources qu'il placera dans le répertoire /usr/lib/X11/app-defaults, mais il pourra également spécifier dans son programme des ressources inaltérables.

L'utilisateur pourra personnaliser l'application en positionnant des ressources dans le fichier .Xdefaults situé dans son répertoire d'accueil. Il pourra également indiquer des ressources en arguments de la ligne de commande qui lance l'application ciblée. Lorsqu'une même ressource est citée dans plusieurs endroits, l'ordre de préséance est celui de la figure ci-dessus.

3.2.6. Modifier des ressources pour tous les utilisateurs

L'auteur d'un client X peut désirer que son application soit installée avec des valeurs de ressources par défaut correctement initialisées pour tous les utilisateurs potentiels. Il peut alors installer dans le répertoire /usr/bin/X11/appdefaults un fichier dont le nom sera la classe de son application, et ce fichier sera consulté à chaque démarrage de l'application. Les clients X ont tous une classe dont le nom est celui de l'application mais avec la première lettre en majuscule (en général). Ainsi la classe de l'application xterm est Xterm, celle de l'application xlatex est Xlatex… Ce mécanisme offre la possibilité de faire cohabiter plusieurs versions d'une même application (sous des noms différents) tout en partageant le même fichier dans /usr/bin/X11/appdefaults.

Un administrateur d'applications X averti pourra souhaiter franciser les clients X les plus fréquemment utilisés. Pour être efficace il portera ses modifications dans le fichier app-defaults existant ou en créera un exemplaire. L'exemple suivant montre l'application xfd avant et après francisation.

Le fichier /usr/bin/X11/app-defaults/Xfd tel que livré par le consortium X :

```
*quit.Label : Quit
*prev.Label : Prev Page
*next.Label : Next Page
*select.Label : Select a character
```

Le fichier /usr/bin/X11/app-defaults/Xfd après modification :

```
*quit.Label : Sortir
*prev.Label : Page precedente
*next.Label : Page suivante
*select.Label : Choisissez un caractere
```

3.2.7. Modifier des ressources pour un utilisateur

Le fichier .Xdefaults situé dans le répertoire d'accueil de l'utilisateur lui permet de personnaliser les applications X-Window. Lorsque le fichier .Xdefaults a été modifié, il faut en avertir le serveur X de la fa,con suivante:

```
xrdb -load ~/.Xdefaults
```

3.2.8. Les ressources spécifiées en arguments de la ligne de commande

Un certain nombre de ressources peuvent être placées en arguments de la ligne de commande. C'est le cas de toutes les ressources standard (font, borderWidth, …) et aussi de certaines ressources des applications dans la mesure où le programmeur en a décidé ainsi. Les ressources spécifiées de cette fa,con sont toujours prioritaires par rapport au contenu de app-defaults et .Xdefaults. L'exemple suivant montre l'appel de l'application xterm en spécifiant des ressources sur la ligne de commande :

```
xterm -fn 9x15 -ls -sb
```

-fn 9x15 est la spécification de la ressource standard font , -ls et -sb sont des spécifications de ressources propres à l'application xterm.

C'est au développeur d'une application de prévoir les ressources personnalisables, il doit également déterminer si les ressources peuvent être simplement spécifiées en arguments de la ligne de commande. Ainsi dans l'application xterm, la ressource scrollBar peut être positionnée en argument de la ligne de commande en utilisant -sb. C'est le créateur de l'application qui en a décidé ainsi, aussi l'indique-t-il dans son manuel.

3.2.9. La commande xrdb

xrdb permet de prendre en compte la personnalisation des ressources par un utilisateur. Cette commande utilise la syntaxe du préprocesseur du langage

C pour permettre de présenter des fichiers de ressources adaptés à différents cas. xrdb connaît (entre autres) les symboles suivants :

- COLOR, qui est positionné si l'écran utilisé est répertorié du genre StaticColor, PseudoColor, TrueColor ou DirectColor (cf. la commande xdpyinfo) ;
- SERVERHOST, le nom de la machine hébergeant le serveur X ;
- VENDOR , une chaîne permettant d'identifier le fournisseur du serveur X utilisé (elle est hélas souvent très longue).

Dans un fichier .Xdefaults, l'utilisateur pourra s'autoriser des écritures du genre :

```
#ifdef COLOR
Mwm*background : SlateGrey
Mwm*foreground : yellow
#endif
```

On utilise en général xrdb en lui fournissant le nom d'un fichier de ressources en argument comme dans l'exemple suivant :

```
xrdb /.Xdefaults
```

3.2.10. Les nouveautés de X11R5

Au niveau de la gestion des ressources, un certain nombre d'améliorations ont été apportées par la révision X11R5.

Dans le nommage des ressources, on peut maintenant introduire le caractère ? pour remplacer un élément d'un nom complet de ressource, comme dans l'exemple suivant :

```
Xdf.?.quitter.background : red
```

La nouvelle ressource customization permet à l'utilisateur d'être automatiquement aiguillé vers un fichier app-defaults adapté à son terminal. En général les auteurs d'applications fournissent maintenant un fichier appdefaults pour les terminaux monochromes et un autre pour les terminaux couleurs. Le fichier contenant les spécifications de couleur est en général

suffixé par -color comme dans l'exemple suivant (l'application xdf) :

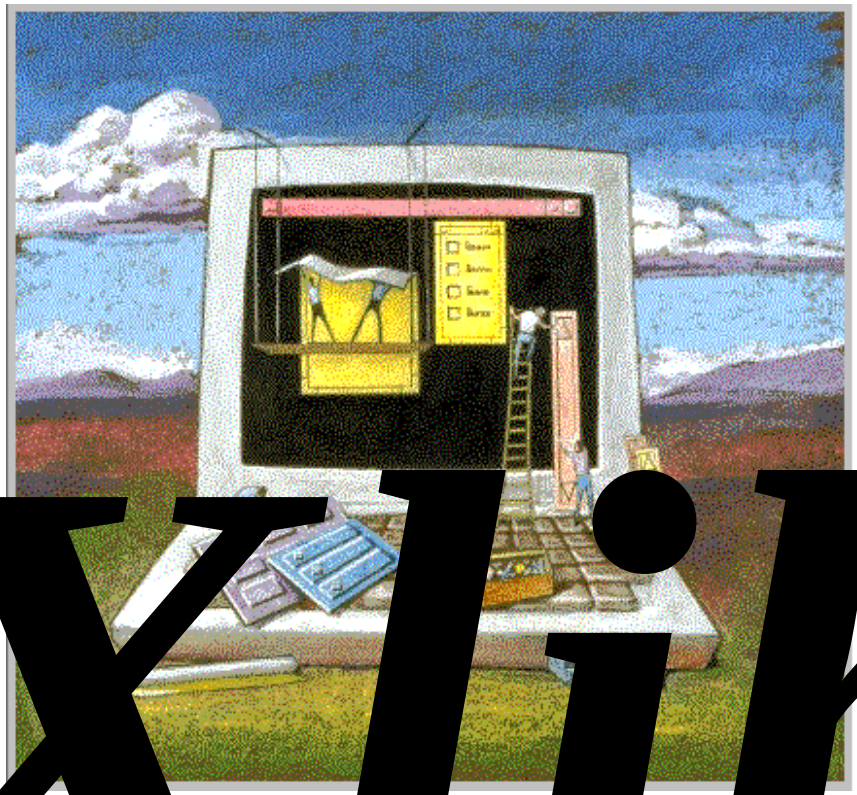
- ***Xdf*** (fichier des ressources par défaut pour les terminaux monochromes)
- ***Xdf-color*** (fichier des ressources par défaut pour les terminaux couleurs)

En positionnant la variable customization à la valeur -color dans un fichier .Xdefaults (*customization : -color), l'utilisateur disposant d'un écran couleur aura une coloration automatique de l'application.

X-Window

Programmation C de Xlib

Franck Diard, Michel Buffa, François Abram, 1998
diard@i3s.unice.fr, http://www.essi.fr/~diard
abram@i3s.unice.fr, http://www.essi.fr/~abram
V 1.4. Mars 1998.



Xlib

1. Display et fenêtres.

La Xlib est une librairie d'interface entre le programmeur et le protocole X11. Le protocole X est le seul langage compréhensible par le serveur X.

Cette librairie est une ensemble de fonctions bas-niveau simples (relativement) que le programmeur peut utiliser et dont les librairies haut-niveau (genre Xt et Motif) se servent.

Il est important de comprendre comment Xlib est conçue, pour pouvoir la programmer efficacement.

1.1. Environnement de programmation.

1.1.1. Conventions concernant les fonctions Xlib.

Toute fonction de la Xlib commence par la lettre X. On ne capitalise que les mots clé. Exemple: ***XOpenDisplay()***, ***XCopyColormapAndFree()***… Certain appels comme ***DisplayPlanes()*** sont des macros et non pas des fonctions.

Toute fonction Xlib commence par l'argument display.

1.1.2. Compilation.

Les programmes utilisant les fonctions de la Xlib doivent inclure:

```
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
```

Pour compiler un programme qui utilise les fonctions de la Xlib et faire l'édition de liens, on utilise les options suivantes:

```
~:> gcc main.c -I/usr/openwin/include
        -L/usr/openwin/lib/X11 -lX11
```

-I/usr/openwin/include: où le compilateur trouvera les includes X11.
-lX11: on linke avec la librairie X11.
-L/usr/openwin/lib/X11: où le linker trouvera les librairies X11.

1.2. Le display.

Pour accéder à un display X11, il faut établir une connexion entre la machine locale et la machine distante (un socket Unix, de type TCP/IP).

```
Display * dpy;
char * display_name=":0";/* ou NULL */

dpy=XOpenDisplay(display_name);
.
.
/* ne pas oublier de le fermer ! */

XCloseDisplay(dpy);
```

Display ** est un pointeur sur une structure dans la mémoire du client qui regroupe des informations sur les proporiétés du display ouvert. Ces informations seront utilisées par la Xlib pour des vérifications sur la possibilité de certaines commandes avant leur envoi au display. Dans l'include ***Xlib.h est défini:

```
typedef struct _XDisplay
{
    XExtData *ext_data;     /* hook for extension to hang data */
    struct _XFreeFuncs *free_funcs; /* internal free functions */
    int fd;                 /* Network socket. */
    int conn_checker;       /* ugly thing used by _XEventsQueued */
    .
    .
    struct _XSQEvent *qfree; /* unallocated event queue elements */
    int (*savedsynchandler)(); /* user synchandler when Xlib usurps */
} Display;
```

La structure ***Display*** est générée au moment de la connection par la fonction ***XOpenDisplay()***.

display_name peut être soit une chaîne spécifiant le nom du display, par exemple "***foc.essi.fr:0.0***", ou bien ***NULL*** (0). Dans ce dernier cas, le nom du display sera celui contenu dans la variable d'environnement shell ***$DISPLAY***.

Attention, pour ouvrir une connexion sur un display local à partir d'une machine distante, il faut que le serveur local l'autorise:

```
local_host:> xhost +another_host
local_host:> rlogin another_host
Passwd:
another_host:> DISPLAY=local_host:0
another_host:> xclock...
```

La commande ***xhost*** et les autres commandes de sécurité de X seront détaillées dans un autre chapitre.

L'argument ***dpy*** sera utilisé dans toutes les requêtes X (en premier argument). Par exemple:

```
XDrawPoint(dpy, drawable, gc, x, y);
```

Il existe une série de macros concernant le display: plusieurs macros sont disponibles pour obtenir des informations sur le display, comme le nombre de couleurs, sa taille, l'ID de la root-window...

Le numéro de l'écran par défaut (souvent sollicité plus loin) peut être obtenu par:

```
DefaultScreen(display);
```

Exemples de macros:

```
int DisplayHeight(dpy, screen_number)
int DisplayWidth(dpy, screen_number)
int DisplayPlanes(dpy, screen_number)
```

1.3. Les fenêtres X.

Une fenêtre est un espace graphique (pas forcément rectangulaire). A chaque fenêtre est affectée une structure descriptive stockée dans le serveur. Les paramètres des fenêtres (taille, hauteur…) sont dans la mémoire du serveur[1]. Du coté du programme client, on ne manipule que des identifieurs de fenêtres. C'est un nombre unique codé sur 32 bits (29 utilisés). Le type ***Window*** est donc un identifieur et en aucun cas un pointeur:

```
typedef unsigned long XID;
typedef XID Window;
```

Une fenêtre occupe environs 100 octets dans la mémoire du serveur: elle ne coûte rien! Les applications courantes comportent plus d'une centaine de fenêtres (***Xmh***...).

Une fenêtre (la structure mémoire et l'identifieur) est détruite si la connexion du display sur laquelle elle est ouverte est rompue.

1.3.1. Caractéristiques des fenêtres.

1.3.1.1. Fenêtres mères et coordonnées.

L'écran X11 se compose d'une root-window qui est en fait le fond de

1. Il est très important de comprendre ceci.

l'écran, et qui contient à son tour d'autres fenêtres qui sont toutes ses filles.

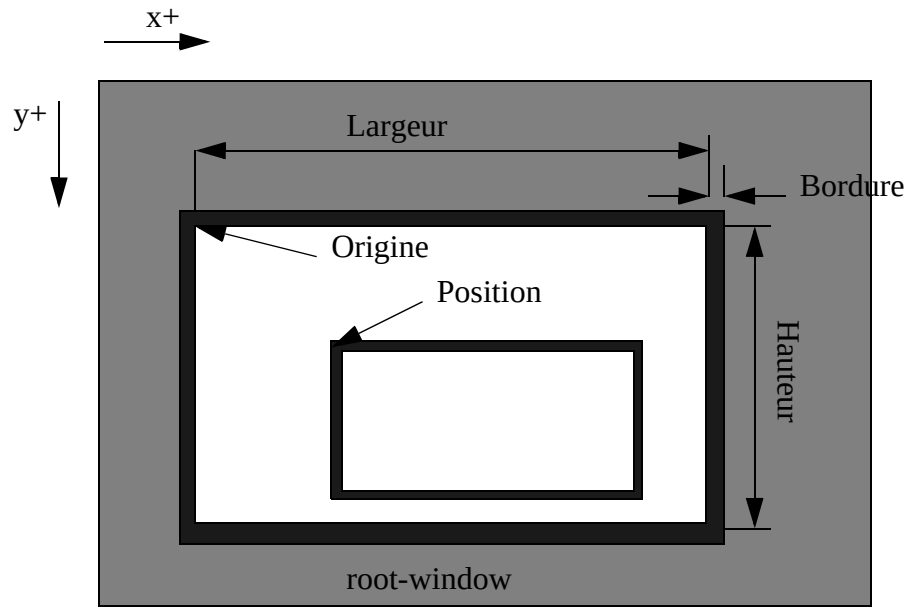


Figure 1: **Système de coordonnées sous X.**

Une fenêtre possède une fenêtre mère, affectée lors de sa création. Une fenêtre peut sortir des limites de sa fenêtre mère et ses coordonnées lui sont relatives (Figure 1). Comme les filles sont clippées par leur fenêtre mère, les fenêtres peuvent sortir de l'écran. Chaque fenêtre se comporte comme un "petit" écran. Une fenêtre peut être de n'importe quelle taille. La largeur, la hauteur et la position d'une fenêtre forme sa géométrie.

Les fenêtres peuvent se superposer. Ainsi une fenêtre peut en cacher une autre.

La taille d'une fenêtre est mesurée en pixels et n'inclut la largeur des bordures. Si la taille de la bordure est 0, elle est invisible.

Une fenêtre n'est pas sensible aux entrées (souris, clavier) si le pointeur de la souris ne la désigne pas[2].

1.3.1.2. Les caractéristiques Depth et Visual Type:

La profondeur (Depth) est le nombre de bits par pixel qui va être disponible pour représenter les couleurs que l'on pourra utiliser pour dessiner dans la fenêtre.

En annexe est présenté un éclaircissement sur le codage des couleurs dans les cartes vidéo.

2. Comportement géré par le window-manager.

Le Visual Type représente la manière dont le serveur va gérer les couleurs. Il existe en effet différents moyens d'accéder à l'écran. Peu de hardwares supportent de multiples Visuals. Nous y reviendrons.

1.3.1.3. Classe d'une fenêtre.

Une fenêtre possède une classe qui diffère selon qu'elle est capable ou non d'afficher des résultats.

Les classes possibles sont:

- ***InputOutput***: la fenêtre peut recevoir des entrées et afficher des résultats.
- ***InputOnly***: la fenêtre ne peut que recevoir des entrées. Exemple: des boutons poussoirs.

1.3.1.4. Attributs des fenêtres.

Les attributs d'une fenêtre contrôlent de nombreux aspects concernant l'apparence d'une fenêtre ainsi que la manière dont elle va réagir aux différents événements.

Exemple d'attributs:

- Couleur du fond, de la bordure.
- A quels événements la fenêtre va-t-elle réagir?
- La fenêtre autorise-t-elle le window-manager à la déplacer, la retailler...

1.3.1.5. Hiérarchie.

Les fenêtres sont arrangées en arbre (Figure 2). Elle ont toutes un ancêtre sauf la root-window (r). La root-window est crée lors du lancement du serveur X.

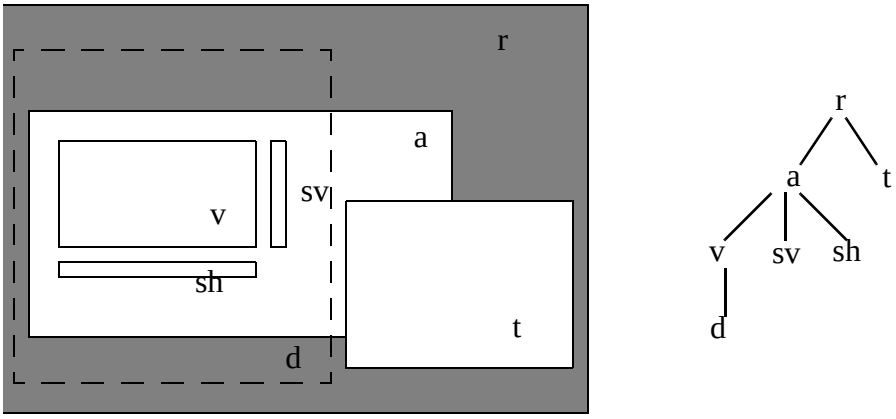


Figure 2: **Hiérarchie des fenêtres X11.**

Les filles directes de la root-window sont spéciales, ce sont des ***top-level***. Une fenêtre top-level est généralement la première créée par une application.

Le stacking order intervient à deux niveaux: il règle la priorité entre les fenêtres ***top-level*** (un application passe entièrement devant une autre). Ensuite, il règle la priorité entre les filles des top-level.

1.3.2. Programmation des fenêtres.

Il existe plusieurs moyens pour créer une fenêtre.

XCreateWindow() est utilisée pour créer une fenêtre quelconque X, les paramètres sont passés dans la structure ***XSetWindowAttributes*** (voir plus ploin).

Exemple :

```
Window win_des;
XSetWindowAttributes xswa;
int winX,winY,winW,winH;
int border_width;
int screen;

/* largeur, hauteur de la fenêtre */
winW=600;
winH=600;

/* largeur de la bordure */
border_width = 3;

/* Position de la fenêtre par rapport à la taille de la RootWindow */
winX=500;
winY=400;

screen=DefaultScreen(dpy);
```

```
/* Couleur de la bordure et du background */
xswa.border_pixel     = BlackPixel(dpy,DefaultScreen(dpy));

win_des = XCreateWindow(display, DefaultRootWindow(dpy),
                winX, winY, winW, winH, border_width,
                DefaultDepth(dpy,screen), InputOutput,
                DefaultVisual(dpy,screen),
                CWBorderPixel, &xswa);

/* Affichage de la fenêtre à l'écran */
XMapWindow(dpy,win_des);
```

La fonction ***XMapWindow()*** est utilisée pour afficher une fenêtre et tous ses enfants. Une fenêtre peut être retirée de l'écran sans être détruite avec ***XUnmapWindow()***.

Il existe une manière plus simple pour créer une fenêtre . L'exemple précédent illustre bien la "lourdeur" et la complexité de la programmation sous Xlib. Heureusement, pour la création de fenêtrest la Xlib propose une fonction plus facile à utiliser.

La fonction ***XCreateSimpleWindow()*** permet de créer plus rapidement une fenêtre.

L'appel de ***XCreateSimpleWindow()*** au lieu de ***XCreateWindow()*** fait que la fenêtre créée hérite de tous les attributs de la fenêtre mère.

Exemple :

```
win_des= XCreateSimpleWindow(display,
RootWindow(display, screen),
x, y,
width, height,
border_width,
BlackPixel(display,screen),
WhitePixel(display,screen));
```

Les attributs de la fenêtre peuvent être modifiés après la création à l'aide de fonctions spécialisées telles ***XSelectInput()***, ***XResizeWindow()***…

Détaillons les différents paramètres de la fonction ***XCreateWindow()***.

```
#include <X11/X11.h>

Display *display;
Window window, parent_window;
int x, y;
unsigned int width, height;
unsigned int border_width;
int depth;
unsigned int class;
Visual *visual;
unsigned long valuemask;
XSetWindowAttributes*attributes;

window = XCreateWindow(display, parent_window, x, y, width, height, border_width, depth, class,
visual, valuemask, attributes);
```

On rappelle que le type ***Window*** est un identificateur de la fenêtre dans

la mémoire du serveur. Le premier paramètre est le display, retourné par la fonction ***XOpenDisplay()***. Le second paramètre est l'identificateur de la fenêtre mère. Il est facile d'obtenir l'identificateur de la ***RootWindow*** grâce à la macro ***DefaultRootWindow(dpy)***.

Les troisième et quatrième paramètres (***x*** et ***y***) donnent la position désirée du coin en haut à gauche de la fenêtre (en pixels). Nous employons le terme "désirée" car le Window Manager peut arbitrairement modifier la position de la fenêtre lors de ca création. Les deux paramètres suivants (***width*** et ***height***) correspondent à la taille de la fenêtre en pixels. Le paramètre suivant (***border_width***) indique l'épaisseur souhaitée du cadre autour de la fenêtre. Ici aussi, si la fenêtre possède comme mère la ***RootWindow***, le Window Manager peut affecter à la bordure une valeur différente de celle spécifiée dans le code.

Le paramètre ***depth*** correspond au nombre de bit-planes disponibles sur le display. Cette valeur peut être obtenue à l'aide de la macro ***DefaultDepth(display, screen_number)***, ou de la constante ***CopyFromParent***.

Le paramètre ***class*** est une constante pouvant prendre une des valeurs suivantes: ***InputOutput***, ***InputOnly***, ou ***CopyFromParent***. La majeure partie des fenêtres que l'on va manipuler seront de la classe ***InputOutput***.

Le paramètre ***visual*** sert à décrire un modèle abstrait du hardware graphique dont est équipé la machine sur laquelle l'application devra être exécutée. Nous étudierons plus en détail ce paramètre lors dans le chapitre consacré à la couleur. En attendant, ne prenons pas de risque et utilisons soit la macro ***DefaultVisual(display, screen_number)***.

Les deux paramètres suivants, ***valuemask*** et ***attributes*** sont très liés. La structure ***XSetWindowAttributes*** contient 15 champs conditionnant l'aspect et le comportement de la fenêtre. La variable ***valuemask*** est un masque de bits qui sert à indiquer quels champs parmis les 15 on va spécifier dans la variable contenant les attributs. Le couple structure/masque n'est qu'un moyen de factoriser le passage de paramètre.

Structure ***XSetWindowAttributes*** (tirée du manuel):

```
typedef struct
{
    Pixmap background_pixmap;       /* background, None, or
    ParentRelative */
    unsigned long background_pixel; /* background pixel */
    Pixmap border_pixmap;  /* border of the window */
    unsigned long border_pixel;     /* border pixel value */
    int bit_gravity;               /* one of bit gravity values */
    int bit_gravity;              /* one of bit gravity values */
    int win_gravity;                /* one of the window gravity values */
    int backing_store;              /* NotUseful, WhenMapped, Always */
    unsigned long backing_planes;   /* planes to be preserved */
    unsigned long backing_pixel; /* value to use in restoring planes */
    Bool save_under; /* should bits under be saved? (popups) */
    long event_mask;                /* set of events that should be saved */
    long do_not_propagate_mask;     /* set of events not propagate */
```

```
    Bool override_redirect; /* boolean value for override_redirect */
    Colormap colormap; /* color map to be associated with window */
    Cursor cursor;          /* cursor to be displayed (or None) */
} XSetWindowAttributes;
```

Voici les 15 champs qui servent à spécifier quels membres de la structure on a rempli.

CWBackPixmap, CWBackPixel, CWBorderPixmap, CWBorderPixel, CWBitGravity, CWWinGravity, CWBackingStore, CWBackingPlanes, CWBackingPixel, CWOverrideRedirect, CWSaveUnder, CWEventMask, CWDontPropagate, CWColormap, CWCursor.

Les attributs de la fenêtre peuvent être modifiés après la création à l'aide de fonctions spécialisées telles :

XChangeWindowAttributes(Display *display,Window w,
unsigned long valuemask, XSetWindowAttributes *attributes);

Un autre type de structure destinée à des paramètres: ***XConfigureWindow(Display *display, Window w, unsigned int value_mask, XWindowChanges *values).***

La structure de type XWindowChanges contient les champs:

```
typedef struct
{
    int x, y;
    int width, height;
    int border_width;
    Window sibling;
    int stack_mode;
} XWindowChanges;
```

Il faut préciser quels champs de la structures ***values*** on a renseigné à l'aide de ces masques:

CWX, CWY, CWWidth, CWHeight, CWBorderWidth, CWSibling, CWStackMode

D'autres fonctions concernant la géométrie sont encore plus spécialisées::

```
XResizeWindow(Display *display, Window w, unsigned int width,
                unsigned int height);
XMoveResizeWindow(Display *display, Window w, int x, int y,
                unsigned int width, unsigned int height);
XSetWindowBorderWidth(Display *display, Window w,
                unsigned int width);
```

2. Les événements.

2.1. Description.

Une application X est dirigée par les événements. Par événement on entend un message provenant du serveur, arrivant de manière asynchrone.

Ces événements peuvent être divisés en plusieurs catégories.

Viennent d'abord les événements correspondants à une action utilisateur comme la frappe d'un touche au clavier.

On trouve ensuite les événements générés par une action de window-manager: exposition (lorsqu'on découvre une partie d'une fenêtre en déplaçant celle qui se trouvait dessus) ou changement de taille.

Enfin, on a les événements envoyés par d'autres applications, ce qui permet de disposer de primitives de base de communication.

La gestion des événements est assurée de façon repartie: le serveur X maintient une file d'attente (FIFO: premier entré, premier sorti) de tous les événements et les envoie au différents clients.

Tout événement est rattaché à une fenêtre.

Il existe plusieurs types d'événements:

- Evénements matériels (envoyés par le Window-Manager):
- Click et mouvement souris.
- Evénements claviers.
- Changements de fenêtres (envoyés par le Window-Manager):
- Entrée dans la fenêtre (focus).
- Sortie de la fenêtre (perte de focus).
- Evénements rattachés à la géométrie de la fenêtre (envoyés par le Window-Manager):
- Destruction de la fenêtre.
- Affichage de la fenêtre.

- Exposition.
- Changement de taille.

2.2. Programmation.

Les Clients demandent à recevoir des événements par des masques à l'aide de fonctions telles que ***XSelectInput(Display * display,Window w, long event_mask)***.

Exemple:

```
XSelectInput(display, win, ExposureMask | KeyPressMask |
        ButtonPressMask | StructureNotifyMask | PointerMotionMask);
```

La fenêtre sera sensible à un changement d'exposition, aux entrées clavier,
aux clicks souris, au changement de taille (structure), et aux déplacements du curseur.

On peut également spécifier le masque d'événement ci-dessus à la création de la fenêtre.

Le traitement des événements se fait à l'aide de routines de lecture spécialisées, celles-ci concernent la file des événements rattachés à une connection sur un display.

La fonction suivante permet d'extraire un événement de la liste et de le copier dans une structure ***XEvent*** locale: ***XNextEvent(Display *, XEvent *)***.

XPeekEvent(Display *, XEvent *) permet quant à elle de copier l'événement mais il n'est pas détruit dans la liste du display.

Pour traiter les événements concernant notre fenêtre, soit on écoute la liste des événements globale rattachée à la connection du display et qui traite des toutes les fenêtres que l'on a ouvert (et dans ce cas on teste le champ ***XEvent->xany.window*** pour traiter seulement ceux qui concerne une fenêtre particulier), soit on s'adresse à une fonction plus spécialisée: ***XWindowEvent(Display *display, Window w, long event_mask, XEvent *event_return)***. Cette dernière fonction ne traite que les événements dont le type correspond à un type spécifié dans le masque ***event_mask*** et dont la fenetre propriétaire est celle passée en paramètre.

Attention, X-Window ne génère pour une fenêtre que les événements dont le type correspond à un type spécifié dans le masque lors de la création de la fenêtre ou avec ***XSelectInput()***. On écoute ensuite avec la fonction ***XWindowEvent*** certains types de messages (dans ce dernier cas, il y a deux filtres de messages).

Le traitement des événements reçus par une fenêtre se fait dans un

boucle de gestion d'événements: après la création et le mapping des fenêtres, les clients doivent boucler et agir en fonction des événements reçus:

Exemple d'une boucle de gestion d'événements.

```
XEvent report;

while(1)
{
    XNextEvent(display, &report); /* si un seule fenêtre */
    switch (report.type)
    {
        case Expose:
            /* traitement des événements de type Expose */
        break;
        case ButtonPress:
            /* traitement des événements de type ButtonPress */
        break;
        case KeyPress:
            /* traitement des événements de type KeyPress */
        break;
        case ConfigureNotify:
            /* traitement des événements de type ConfigureNotify */
        case /* ....*/ :
            /* .....*/
        break;
    }
}
```

La structure d'un événement (type ***XEvent***) est une union entre différentes structures d'événements bien adaptés à chaque classe d'événement.

La structure ci-dessous, tirée des includes, vous permet de connaître tous les types d'événements. On accédera aux informations retournées par ces événements
en accédant au champ correspondant de la structure union.

```
typedef union _XEvent {
      int type; /* must not be changed; first element */
      XAnyEvent xany;
      XKeyEvent xkey;
      XButtonEvent xbutton;
      XMotionEvent xmotion;
      XCrossingEvent xcrossing;
      XFocusChangeEvent xfocus;
      XExposeEvent xexpose;
      XGraphicsExposeEvent xgraphicsexpose;
      XNoExposeEvent xnoexpose;
      XVisibilityEvent xvisibility;
      XCreateWindowEvent xcreatewindow;
      XDestroyWindowEvent xdestroywindow;
      XUnmapEvent xunmap;
      XMapEvent xmap;
      XMapRequestEvent xmaprequest;
      XReparentEvent xreparent;
      XConfigureEvent xconfigure;
      XGravityEvent xgravity;
      XResizeRequestEvent xresizerequest;
      XConfigureRequestEvent xconfigurerequest;
      XCirculateEvent xcirculate;
      XCirculateRequestEvent xcirculaterequest;
      XPropertyEvent xproperty;
      XSelectionClearEvent xselectionclear;
      XSelectionRequestEvent xselectionrequest;
      XSelectionEvent xselection;
      XColormapEvent xcolormap;
```

```
    XClientMessageEvent xclient;
    XMappingEvent xmapping;
    XErrorEvent xerror;
    XKeymapEvent xkeymap;
    long pad[24];
} XEvent;
```

Le champ ***type*** de la structure ***XEvent***. permet de tester le type de chaque événement reçu.

Selon le type de l'événement, on peut indexer sur la structure mémoire concernée. Cela évite d'avoir une structure "fourre-tout" dans laquelle il y aurait tous les paramètres de tous les événements.

Par exemple, pour traiter les caractères entrés au clavier:

```
case KeyPress:
{
    char buffer;
    int bufsize=1;
    int charcount;

    charcount = XLookupString(&(report.xkey), &buffer, bufsize, NULL, NULL);

    switch(buffer) {
    case 'q':
        printf("Touche q pressée\n");
        break;
    default:
        printf("Touche non valide\n");
        break;
    }
}
```

Pour lire les coordonnées de la souris pendant un click souris:

```
case ButtonPress:
{
    switch (report.xbutton.button)
    {
        case Button1 :
        {
            x = xevent.xbutton.x;
            y = xevent.xbutton.y;
            printf("Bouton 1 pressé en %d %d\n",x,y);
            break;
        }
        case Button3 :
        {
            /* ......*/
            break;
        }
    }
    break;
}
```

Dans l'exemple ci-dessus, le type de l'événement ***ButtonPress*** signifie l'enfoncement d'une touche de la souris.

Dans l'exemple suivant, on récupère la nouvelle taille de la fenêtre (sans faire appel à la fonction ***XGetGeometry()***).

```
case ConfigureNotify:
{
    printf("Fenetre retaillee: width = %d height = %d\n",
```

```
                report.xconfigure.width, report.xconfigure.height);
    break;
}
```

Il existe de très nombreux événements, cependant nous ne nous servons généralement que des quatre ou cinq plus utiles.

Ne pas confondre:

- Le nom de la structure de l'événement. Exemple: ***XButtonEvent***, ***XKeyEvent***, ***XMotionEvent***, ***XExposeEvent***, ***XConfigureEvent***
- Le nom des constantes associés qui vont figurer dans le switch de la boucle de gestion des événements: ***KeyPress***, ***KeyRelease***, ***ButtonPress***, ***ButtonRelease***, ***MotionNotify***, ***Expose***, ***ConfigureNotify***
- Le nom des masques qui figurent dans l'appel à ***XSelectInput()***.

Par exemple, lors d'un mouvement de la souris, un événement du type ***MotionNotify*** est reçu (le fait que l'on veuille recevoir les événements du mouvement de la souris été spécifié par ***PointerMotionMask***).

3. Les fonctions graphiques.

La Xlib fournit une panoplie assez complète de fonctions de dessin 2D. Ces fonctions sont en général simples à utiliser:

- Tracés de points, lignes, rectangles, arcs, polygones et textes, creux ou pleins.
- Requêtes simples ou multiples (tracer 1 ligne ou plusieurs lignes?).

3.1. Graphic context.

Pour dessiner il est nécessaire de spécifier des choses telles que la largeur du trait, si l'on veut tracer des lignes en trait plein ou en pointilles, la couleur, les patterns pour le remplissage des formes géométriques...

Avec la Xlib, on passe ces informations aux fonctions de dessin grâce à un identifieur de type GC. Une telle structure doit être créée et initialisée avant l'appel des fonctions de dessin. Plusieurs structures de ce type peuvent être créées, rendant le dessin dans différents styles simple à réaliser. Le GC est un des paramètres obligatoire de toutes les fonctions de dessin de la Xlib.

3.2. Programmation du contexte graphique.

Utilisation: créer un GC par type d'opération graphique, par exemple un pour chaque fonte de caractères utilisée. On se servira des différents GCs selon que l'on désire écrire les textes à l'écran en italique, en gras, en gros, en petit...

Pour créer un GC il faut utiliser la fonction ***XCreateGC(Display *display, Drawable d, unsigned long valuemask, XGCValues *values)*** qui accepte 4 paramètres:

- Un pointeur (Display *) sur le Display.
- Un pointeur (XID) sur un Drawable (fenêtre ou pixmap).
- Un pointeur sur un objet de type ***XGCValues***.
- Un masque spécifiant quels champs de la structure

XGCValues on a renseigné.

On peut créer le GC soit en passant des champs dans la structure avant l'appel de la fonction ***XCreateGC()***, soit ne pas s'en occuper et configurer le contexte graphique par la suite à l'aide de fonctions spécifiques comme ***XSetForeground()***.

Un masque sur les champs du paramètre précédent. Si ce paramètre est NULL alors le GC est initialisé avec les valeurs par défaut. Sinon, on peut l'initialiser comme bitmask et indiquer quels champs de la structure ***XGCValues*** sont déjà initialisés.

La structure ***XGCValues*** contient des éléments tels que ***foreground***, ***background***, ***line_width***, ***line_style*** que l'on peut initialiser. Le masque est initialisé en utilisant des valeurs comme ***GCForeground***, ***GCBackground***, ***GCLineStyle***...

La Xlib fournit des macros très utiles lorsque l'on initialise un contexte graphique. Parmi les plus souvent utilisées ***BlackPixel()*** et ***WhitePixel()*** renvoient la valeur par défaut des couleurs noir et blanc pour un display donné. Ces valeurs sont prises dans la table de couleur (colormap) par défaut.

Deux exemples de création de GC:

Initialisation du GC par défaut, ensuite on peut le modifier:

```
GC gc;
.
.
/* Ouvrir le display, créer les fenêtres, etc... */
.
.
gc = XCreateGC(display, RootWindow(display, screen), 0, NULL);

XSetForeground(display, gc, BlackPixel(display, screen));
XSetBackground(display, gc, WhitePixel(display, screen));

/* maintenant on peut utiliser ce GC dans avec les fonctions de
dessin */
.
.
```

Initialisation du GC par défaut, ensuite on peut le modifier:

```
GC gc;
XGCValues values;
unsigned long valuemask;

/* Ouvrir le display, créer les fenetres, etc... */

values.foreground = BlackPixel(display, screen);
values.background = WhitePixel(display, screen);
gc = XCreateGC(display, RootWindow(display, screen), (GVForeground | GCBackground), &values);
/* maintenant on peut utiliser ce GC dans avec les fonctions de
dessin */
```

Finalement, la mise à jour des paramètres d'un contexte graphique se fait de la même manière que pour les fenêtres, on doit utiliser des fonctions spécifiques pour mettre à jour des champs de structure dans la mémoire du

serveur dont on ne possède que des identifieurs (***XID***).

3.3. Les fonctions graphiques.

Ces fonctions travaillent sur des Drawables: Pixmaps ou Windows. Le résultat dépend du GC.

Les fonctions de dessin:

XDrawPoint(display, drawable, gc, x, y)

Probablement la plus simple des fonctions graphiques. Elle dessine un point avec la couleur courante décrite dans le GC à la position (x, y).

XDrawPoints(display, drawable, gc, XPoint *pts, n, mode)

Fonction similaire à la précédente si ce n'est qu'un tableau de n éléments de type ***XPoint*** est dessiné. Le paramètre mode (un ***int***) vaut soit ***CoordModeOrigin*** soit ***CoordModePrevious*** selon que les coordonnées des points sont relatives à l'origine ou aux coordonnées du dernier point.

XDrawLine(display, drawable, gc, x1, y1, x2, y2)

Dessine une ligne entre (x1, y1) et (x2, y2).

XDrawLines(display, drawable, gc, XPoints *pts, n, mode)

Dessine un ensemble de lignes connectées, en prenant des paires de points dans la liste. Voir la fonction ***XDrawPoints()*** pour le paramètre mode.

XDrawRectangle(display, drawable, gc, x, y, width, height)

Dessine un rectangle dont le coin en haut à gauche est en (x, y) et de taille (***width***, ***height***).

XFillRectangle(display, drawable, gc, x, y, width, height)

Dessine un rectangle plein. L'attribut ***fill_style*** du GC spécifie la manière dont on effectue le remplissage (couleur pleine, pattern...)

XFillPolygon(display, drawable, gc, XPoint *pts, n, shape, mode

Semblable à ***XDrawLines*** sauf que le polygone dessiné est rempli. Le paramètre ***shape*** sert d'aide au serveur pour optimiser le remplissage du polygone.

Les valeurs possibles sont:

- ***Complex***: Les cotés du polygone peuvent s'intercepter.
- ***Nonconvex***: Les cotés du polygone ne s'interceptent pas mais il n'est pas convexe.
- ***Convex***: le polygone est convexe, le dessin sera nettement plus rapide.

```
XDrawArc(display, drawable, gc, x, y, width, height, angle1, angle2)
```

Les paramètres ***x***, ***y***, ***width*** et ***height*** définissent une boite (Bounding Box) contenant l'arc de cercle. Le centre de l'arc est le centre de la boite. Les paramètres angle1 et angle2 spécifient la zone du cercle qui sera dessinée. Ces angles sont en 1/64 de degrés, mesurés dans le sens trigonométrique. L'angle de valeur zéro est à la position 3 heures. Le paramètre angle2 est mesuré relativement à angle1.

Ainsi, pour dessiner un cercle entier, il faudra positionner ***width*** et ***height*** égaux au diamètre, et choisir angle1 = 0 et angle2 = 360*64 =23040.

```
XFillArc(display, drawable, gc, x, y, width, height, angle1, angle2)
```

Idem à la fonction précédente sauf que l'arc de cercle est plein. Les fonctions de tracé de texte seront étudiées ultérieurement. Les fonctions de tracé de bitmaps également.

Effacement d'une fenêtre:

```
XClearWindow(display, drawable)
```

Efface le contenu d'une fenêtre ou d'un Pixmap.

3.4. Les drawables.

On appelle Drawable un objet Xlib dans lequel on peut dessiner. En d'autres termes, un Drawable est un identificateur vers une structure qui va contenir des graphiques: une fenêtre (***Window***) ou une zone mémoire (***Pixmap***). On peut utiliser la fonction ***XCopyArea()*** de l'un à l'autre.

3.5. Le texte sous X.

3.5.1. Description.

Un texte peut être imprimé à l'écran en spécifiant une chaîne de caractères et une police. Une police est un ensemble de caractères de même

esthétique. Une fonte est un ensemble de polices de la même famille. On appelle abusivement une police, une fonte.

La métrique d'une fonte regroupe 5 paramètres.

- La hauteur.
- La profondeur.
- La largeur.
- L'approche.
- L'incrément.

Voir sur la figure suivante la signification de ces paramètres.

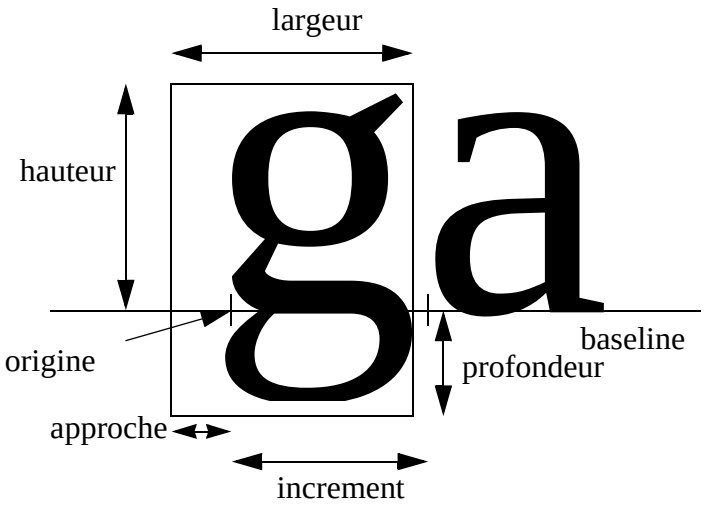


Figure 3: **Métrique d'une fonte X.**

Le principe de fonctionnement est toujours le même. La fonte doit être dans la mémoire du serveur pour que son accès y soit rapide. La structure mémoire descriptive de la fonte est chargée quand on en fait la requête. La seule chose vue par le programme est un identifieur unique (comme pour les fenêtres ou les colormaps).

Avant d'utiliser une police de caractères il faut l'avoir demandée au serveur. Le serveur possède des fontes par défaut (en ROM pour les terminaux X). En général elle est lue sur disque et envoyée au serveur; cependant, si une autre application est déjà en train de l'utiliser elle sera partagée dans la mémoire du serveur X.

On peut consulter les différentes polices (fontes) de caractères disponibles: les noms des différentes polices peuvent être consultés à l'aide

du programme ***xlsfonts***.

Une police sera visualisée à l'aide du programme ***xfd -fn font_name***. Un autre logiciel plus interactif permet de visualiser et choisir les différentes polices en même temps: ***xfontsel***.

3.5.2. Programmation de l'affichage de texte.

Plusieurs manières de charger une police de caractères:

- Manière simple: le chargement d'une police se fait à l'aide de la fonction ***Font *XLoadFont(display, font_name)***. Cette fonction charge la police et retourne un ID de la fonte (de type Font). Cet ID peut être ensuite utilisé par la fonction ***XSetFont(display, gc, font_ID)*** pour associer la police à un Contexte Graphique (GC).

- Manière un peu plus compliquée: il est très difficile de calculer la hauteur et la longueur d'une chaîne de caractères quelconque sans connaître les caractéristiques détaillées de la police utilisée (surtout si celle-ci n'est pas proportionnelle).

Si une police a déjà été chargée par un appel à ***XLoadFont(),*** on peut utiliser la fonction ***XFontStruct *XQueryFont(display, font_ID)*** qui fournit des informations très détaillées sur la police passée en paramètre. Ces informations se trouvent dans la structure ***XFontStruct***.

Remarque: il est possible de faire ***XLoadFont()*** et ***XQueryFont()*** en une seule fois à l'aide de la fonction ***XFontStruct *XLoadQueryFont(display, font_name)***.

Exemple de code pour la lecture d'une fonte de caractères:

```
XFontStruct *font_info;

char *fontname = "9x15";
if((*font_info = XLoadQueryFont(display,fontname)) == NULL)
{
    fprintf(stderr,"Police de caractères %s non trouvée \n",fontname);
    exit (-1);
}
```

Ne pas oublier de libérer la mémoire allouée: lorsque vous n'avez plus besoin d'une police de caractères, libérez-la en appelant la fonction ***XFreeFont(display, font_struct)***.

Largeur et hauteur d'une chaîne de caractères:

Chaque caractère d'une police donnée possède trois attributs qui sont utiles pour le calcul de la hauteur et de la largeur d'un texte:

- Une ***baseline***: ligne horizontale qui coupe le caractère en deux, au dessous de laquelle se trouve le jambage.
- Un ***ascent***: indique le nombre de pixels au-dessus de la baseline.
- Un ***descent***: indique le nombre de pixels au-dessous de la baseline.

La hauteur d'un texte s'obtient à l'aide des champs ***ascent*** et ***descent*** du champ ***max_bounds (de type XCharStruct)*** de la structure ***XFontStruct***.

La largeur d'un texte s'obtient à l'aide de la fonction ***XTextWidth()***.

Cet exemple montre comment procéder:

```
#define STRING "toto"
{
XFontStruct *fontstruct;
fontstruct = XLoadQueryFont(display,fontname);
height_font=
    fontstruct.max_bounds.ascent+fontstruct.max_bounds.decent;

   width_font = XTextWidth(fontstruct, STRING, strlen(STRING));
}
```

Pour afficher le texte (enfin), il existe trois fonctions principales.

```
XDrawString(display, drawable, gc, x, y, string, length)
```

Affiche une chaîne de caractères dans un Drawable. Attention, x et y représentent les coordonnées de la ***baseline*** de la chaîne.

Rappel: la ***baseline*** d'une chaîne est une ligne horizontale en dessous de laquelle les caractères possèdent un "jambage".

```
XDrawImageString(display, drawable, gc, x, y, string, length)
```

Similaire à la fonction précédente sauf que la boite qui contient la chaîne est remplie avec la couleur de background spécifié dans le contexte graphique passé en paramètre.

En inversant les champs ***background*** et ***foreground*** du GC, on peut facilement afficher du texte en inverse vidéo à l'aide de cette fonction.

```
XDrawText(display, drawable, gc, x, y, items, nitems)
```

Cette fonction permet d'afficher une ou plusieurs chaîne(s) de caractères à la fois. Chaque item est un objet de type ***XTextItem*** qui contient la chaîne de caractères, la police de caractères, un espacement horizontal à partir de la fin de l'item précédent...

4. Gestion de la couleur sous X11.

4.1. Concepts de base, vocabulaire incontournable

Jusqu'à présent nous avons spécifié les couleurs au travers des champs foreground et background d'un GC. Pour cela, nous avons utilisé les macros BlackPixel() et WhitePixel() car nous nous étions limités au noir et blanc. Il est cependant possible de mettre dans ces champs une valeur correspondant à une véritable couleur.

Parce que X a été conçu pour supporter une large variété d'architectures hardware, les mécanismes de gestion de la couleur proposés par la Xlib sont relativement complexes.

La plupart des écrans couleurs aujourd'hui utilisent le modèle RGB pour le codage des couleurs.

Un écran couleur utilise plusieurs bits par pixels (multiple bit planes screen) pour coder la couleur de ce pixel (pixel value).

Une colormap est utilisée pour transformer la valeur numérique d'un pixel (pixel value) vers la couleur effectivement visible à l'écran. Une colormap est en réalité une table située dans la mémoire du serveur; la valeur numérique d'un pixel est un index dans cette table (***pixel value***). A une pixel value de 16 correspond une couleur spécifiée par le seizième élément de la colormap. Un élément de la colormap est appelé un ***colorcell***.

La sensibilité des couleurs est donc très fine: chaque composante varie de 0 à 65535. Une couleur sous X est représentée par:

```
typedef struct {
        unsigned long pixel;/* pixel value */
        unsigned short red, green, blue;/* rgb values */
        char flags;         /* DoRed, DoGreen, DoBlue */
        char pad;
    } XColor;
```

En général, chaque colorcell contient 3 valeurs codées sur 16 bits, correspondant aux intensités de chacune des composante RGB.

Si on utilise 8 bits (sur les 16) pour coder chaque composante, une colorcell peut coder 256 puissance 3 couleurs (16 millions).

Le nombre de couleurs pouvant être affichées simultanément à l'écran dépend du nombre de bitplanes utilisés pour coder la pixel value. Un système à 8 bitplanes pourra indexer 2 puissance 8 colorcells (256 couleurs). Un

système avec 24 bitplanes pourra afficher 16 millions de couleurs.

4.1.1. Colormap.

X propose le concept de colormap virtuelle ou colormap privée, qui permet de gérer plusieurs colormaps, chacune proposant une palette différente; mais une seule peut être active à un moment donné. Ces colormaps sont swappées (échangées avec la colormap hardware) par le Window Manager lorsque le focus passe d'une fenêtre à l'autre. Conséquence de cette limitation : les fenêtres présentes à l'écran voient leurs couleurs s'altérer lorsque le focus est dans une fenêtre possèdant une colormap différente.

La colormap est une structure de données stockée dans le serveur X. On n'a pas accès directement aux valeurs. On procède à des requêtes. Un identifieur de colormap est un identifieur unique par display:

```
typedef unsigned long XID;
typedef XID Colormap;
```

4.1.1.1. Allocation de couleurs.

Si l'on veut dessiner en bleu, il faut allouer une case dans la colormap qui contient les composantes (0, 0, 255). L'allocation renvoie l' indice de cette couleur dans la colormap (par exemple: 37). Dessiner dans le frame-buffer avec la valeur de pixel égale à 37 produira des graphiques bleus.

Mais si 15 clients allouent la couleur bleue, il est dommage de dupliquer 15 fois l'entrée dans la colormap. X propose le partage des couleurs.

Un client désirant utiliser une couleur donnée ne spécifie pas explicitement la pixel value et la couleur de la colorcell correspondant à cette pixel value. A la place, il demande l'accès à une colorcell dans la colormap (gérée par le serveur X), et une pixel value lui est retournée.

On appelle ce mécanisme l'allocation de couleur: lorsqu'un client désire utiliser la couleur bleue, c'est comme s'il posait au serveur la question :

- "Quelle colorcell dois-je utiliser pour dessiner en bleu ?",

et le serveur lui répond :

- "Tu peux utiliser la colorcell correspondant à cette pixel value!"

C'est ça l'allocation des couleurs!

4.1.1.2. Partage des couleurs.

Dans les modes ***DirectColor***, ***GrayScale*** et ***PseudoColor***, on peut éditer la colormap. Une entrée de la colormap est nommée ***colorcell***.

Il en existe de deux types:

- En lecture seule (RO).
- En lecture/écriture (RW).

4.1.1.3. Cellules RO.

Une cellule RO positionnée par un client (s'il reste de la place dans la colormap) peut être partagée avec les autres clients qui sont lancés après. Dans ce cas, la couleur sera libérée après que le dernier client qui l'utilise soit fermé.

4.1.1.4. Cellules RW.

Une cellule RW positionnée par un client (s'il reste de la place dans la colormap) ne pourra pas être exploitée par un autre client. Comme on l'alloue en lecture/écriture, on pourrait changer son contenu et les autres clients qui l'exploitent changerait d'apparence. Cela est contraire aux principes de X.

4.1.2. Private vs. Public.

Un client X doit aussi pouvoir allouer ses 256 couleurs (par exemple une LUT de gris). Dans la colormap système standard, certaines ***colorcells*** sont RO (environ une quinzaine quand on démarre X). On peut donc allouer au plus 240 couleurs.

Dans ce cas, on peut déclarer qu'une fenêtre d'un client possède une colormap privée. Cette colormap sera installée sur l'écran dès que le focus concernera cette fenêtre.

Pour programmer la mise en place d'une colormap privée, on alloue une colormap et on l'affecte à une fenêtre. Elle sera installée dès que le focus entrera dans la fenêtre. Il suffit ensuite de remplir la colormap avec les entrées souhaitées.

```
Display *Dpy;
Window Win_desc;
XSetWindowAttributes xswa;
XColor xcol;
int Scr, n;
Visual *visual;
Colormap Cmap;

Scr = DefaultScreen(Dpy);
Win_desc = XCreateSimpleWindow(Dpy, RootWindow(Dpy, Scr), 100, 100, 640, 480, 3,
WhitePixel(Dpy, Scr), BlackPixel(Dpy, Scr));

visual = DefaultVisual(Dpy, Scr);
Cmap=XCreateColormap(Dpy,DefaultRootWindow(Dpy), visual, AllocAll);

xswa.colormap = Cmap;
XChangeWindowAttributes(Dpy, Win_desc, CWColormap, &xswa);
for (n = 0; n < 256; n++)
```

```
{
    xcol.pixel = (unsigned long) n;
    xcol.red = n << 8;
    xcol.green = n << 8;
    xcol.blue = n << 8;
    xcol.flags = DoRed | DoGreen | DoBlue;
    XStoreColor(Dpy, Cmap, &xcol);
}
```

Dans l'exemple précédent, on a rempli la colormap de niveaux de gris du noir au blanc.

Les fonctions ***XQueryColor()*** et ***XQueryColors()*** permettent de connaître les valeurs RGB de chaque colorcell. Pour mettre des colorcells à jour on utilise les fonction ***XStoreColor()*** et ***XStoreColors()***.

4.1.3. Allocation de couleurs partagées.

Lorsqu'on peut s'en contenter, pour les raisons précedemment citées, il est fortement conseillé d'allouer des couleurs en ***Read-Only***.

La pixel value retournée par les fonctions d'allocation de couleur servira à positionner les champs ***foreground*** ou ***background*** d'un ***GC*** ou bien les attributs ***background_pixel*** ou ***border_pixel*** d'une fenêtre.

4.1.3.1. Fonctions d'allocation de couleurs en Read-Only

XAllocColor() renvoie la pixel value (indice de colorcell dans la colormap) qui contient la valeur RGB demandée, ou qui contient la valeur RGB la plus proche disponible sur cet écran. ***XAllocNamedColor()*** renvoie la pixel value (indice de colorcell dans la colormap) qui contient la valeur RGB correspondant au nom passé en paramètre (qui doit être dans la base de données des couleurs standards), ou qui contient la valeur RGB la plus proche disponible sur cet écran. L'utilisation de cette fonction est recommandé car les couleurs ont plus de chances d'être partagées.

XParseColor() parse le nom d'une couleur ou la spécification hexadécimale d'une couleur et renvoie les valeurs RGB correspondantes. On l'utilise en général en tandem avec ***XAllocColor()***. L'utilisation de ces deux fonctions au lieu de ***XAllocNamedColor()*** permet de repérer de manière plus fine les erreurs de syntaxe dans la spécification de la couleur.

Il n'est pas possible de savoir si une colorcell est ***Read/Write*** ou ***Read-Only***!

La seule manière de savoir combien de colorcells sont disponibles pour l'allocation consiste à allouer N couleurs à l'aide de la fonction ***XAllocColorCells***(), avec une grande valeur de N, puis à recommencer en diminuant la valeur de N jusqu'à ce que l'allocation ait réussi.

4.1.3.2. La base de données de couleurs standards.

X propose une base de données de couleurs standards, qui associe des noms de couleurs à des valeurs RGB. Cette base de données standard encourage le partage des couleurs par les applications clients. Elle se trouve dans ***/usr/lib/X11/rgb.txt*** (***/usr/openwin/lib/rgb.txt*** sous OpenWindows) et possède près de 300 entrées.

Le partage des couleurs ne peut intervenir que si deux applications allouent une même couleur read-only, c'est-à-dire une couleur possèdant les mêmes valeurs des composantes R, G et B, et ne pouvant être modifiée (c'est pourquoi on l'appelle read-only). A l'aide de cette base de données, il y a plus de chances pour que les clients partagent les couleurs que s'ils les allouent eux-même en les prenant parmi les 2 puissance 48 combinaisons RGB possibles.

Il est à noter que le serveur X n'utilise pas directement le fichier ***/usr/lib/X11/lib/rgb.txt***, mais une version compilée de celui-ci.

4.1.3.3. Nommage hexadécimal des couleurs

Il est également possible, même si cela est déconseillé, de spécifier une couleur en hexadécimal.

Quatre formats possibles :

```
#RGB          (4 bits pour chaque composante)
#RRGGBB       (8 bits  "   "   "   "  )
#RRRGGGBBB    (12 bits "   "   "   "  )
# RRRRGGGGBBBB (16 bits "   "   "   "  )
```

Chaque lettre correspond à un chiffre hexadécimal. #3a7 et #3000a0007000 sont équivalents.

4.1.4. Création et installation de colormaps.

Nous avons parlé de colormaps hardware et de colormaps privées ou colormaps virtuelles. Nous allons étudier ici comment manipuler ces objets.

Une colormap hardware est un objet physique dont le contenu (valeurs RGB) est lu par le hardware vidéo de l'écran pour afficher les couleurs. La plupart des stations de travail possèdent une seule colormap hardware. Certaines machines haut de gamme en possèdent plusieurs, ce qui permet à plusieurs fenêtres d'avoir chacune leur propre colormap hardware.

En général, on peut modifier les valeurs RGB des colorcells de la colormap hardware, ou bien on peut swapper l'ensemble des valeurs de la colormap hardware avec celles d'une colormap située dans la mémoire de l'ordinateur (colormap virtuelle ou privée). Les visuals DirectColor, GrayScale et PseudoColor ne sont disponibles que sur des écrans ayant ces

possibilités.

Jusqu'à présent, nous n'avons alloué que des couleurs dans la colormap par défaut (créée lors du lancement du serveur). Sur des écrans limités à 256 couleurs, il arrive souvent que des clients gourmands en couleurs, ou qui nécessitent des couleurs précises pour fonctionner correctement (comme XV, Netscape, Xemacs), allouent la totalité des colorcells disponibles en Read/Write.

Dans ce cas, la solution pour une application en manque de couleurs consiste à utiliser une colormap privée.

Le Window Manager se chargera de faire l'échange entre la colormap hardware et cette colormap privée lorsque le focus sera dans une fenêtre de l'application.

Lorsqu'une application créée une colormap privée, elle doit positionner l'attribut colormap de sa top-level window avec l'identificateur de cette colormap, afin que le Window Manager sache quelle colormap installer. Le WM ne peut dialoguer qu'avec les top-level windows.

4.1.5. Fonctions de manipulation des colormaps

- ***XCreateColormap***(): Crée une colormap correspondant au visual passé en paramètre, avec toutes les colorcells allouées en Read/Write, ou sans allocation de colorcells. Si on n'alloue pas les colorcells lors de l'appel, on pourra par la suite les allouer indépendamment en Read-Only ou en Read/Write à l'aide des fonctions de manipulation de colorcells déjà étudiées. Si on alloue les colorcells lors de l'appel, il suffit ensuite de les initialiser à l'aide de ***XStoreColors***() par exemple.
- ***XfreeColormap***(): Libère les ressources occupées par une colormap privée. Envoie un événement ***ColormapNotify*** à toutes les fenêtres qui utilisaient cette colormap.
- ***XlistInstalledColormaps***(): Liste les colormaps installées.
- ***XCopyColorMapAndFree***(): Permet de copier une colormap dans une autre, et de libérer l'ancienne. Ceci est utile lorsque l'allocation des colorcells échoue aprés que certaines colorcells aient été allouées avec succès. Ca évite de re-créer une colormap et de recommencer l'allocation des colorcells depuis le début.
- ***XSetWindowColormap***(): Positionne l'attribut colormap d'une window.

- ***XInstallColormap***(): Non étudiée. Utile si on veut écrire un Window Manager.
- ***XUninstallColormap***(): Non étudiée. Utile si on veut écrire un Window Manager.

4.1.6. Exemple de code allouant des couleurs en Read-Only.

```
/*
 * Copyright 1989 O'Reilly and Associates, Inc.
 * See ../Copyright for complete rights and liability information.
 */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>

extern Display *display;
extern int screen_num;
extern Screen *screen_ptr;
extern unsigned long foreground_pixel, background_pixel, border_pixel;
extern char *progname;

#define MAX_COLORS 3

static char *visual_class[] = {
"StaticGray",
"GrayScale",
"StaticColor",
"PseudoColor",
"TrueColor",
"DirectColor"
};

get_colors()
{
   int default_depth;
   Visual *default_visual;
   static char *name[] = {"Red", "Yellow", "Green"};
   XColor exact_def;
   Colormap default_cmap;
   int ncolors = 0;
   int colors[MAX_COLORS];
   int i = 5;
   XVisualInfo visual_info;

   /* Try to allocate colors for PseudoColor, TrueColor,
    * DirectColor, and StaticColor.  Use black and white
    * for StaticGray and GrayScale */

   default_depth = DefaultDepth(display, screen_num);
   default_visual = DefaultVisual(display, screen_num);
   default_cmap   = DefaultColormap(display, screen_num);

   if (default_depth == 1) {
      /* must be StaticGray, use black and white */
      border_pixel = BlackPixel(display, screen_num);
      background_pixel = WhitePixel(display, screen_num);
      foreground_pixel = BlackPixel(display, screen_num);
      return(0);
   }

   while (!XMatchVisualInfo(display, screen_num, default_depth,
                        /* visual class */i--, &visual_info));

   printf("%s: found a %s class visual at default_depth.\n",
```

```
        progname, visual_class[++i]);

    if (i < 2) {
        /* No color visual available at default_depth.
         * Some applications might call XMatchVisualInfo
         * here to try for a GrayScale visual
         * if they can use gray to advantage, before
         * giving up and using black and white.
         */
        border_pixel = BlackPixel(display, screen_num);
        background_pixel = WhitePixel(display, screen_num);
        foreground_pixel = BlackPixel(display, screen_num);
        return(0);
    }

    /* otherwise, got a color visual at default_depth */


    /* The visual we found is not necessarily the
     * default visual, and therefore it is not necessarily
     * the one we used to create our window.  However,
     * we now know for sure that color is supported, so the
     llowing code will work (or fail in a controlled way).
     * Let's check just out of curiosity: */
    if (visual_info.visual != default_visual)
        printf("%s: PseudoColor visual at default depth is not default
                visual!\nContinuing anyway...\n", progname);

    for (i = 0; i < MAX_COLORS; i++) {
        printf("allocating %s\n", name[i]);

        if (!XParseColor (display, default_cmap, name[i], &exact_def)) {
            fprintf(stderr, "%s: color name %s not in database",
                    progname, name[i]);
            exit(0);
        }

        printf("The RGB values from the database are %d, %d, %d\n",
                exact_def.red, exact_def.green, exact_def.blue);

        if (!XAllocColor(display, default_cmap, &exact_def)) {
            fprintf(stderr, "%s: can't allocate color: all colorcells
                    allocated and no matching cell found.\n", progname);
            exit(0);
        }

        printf("The RGB values actually allocated are %d, %d, %d\n",
                exact_def.red, exact_def.green, exact_def.blue);
        colors[i] = exact_def.pixel;
        ncolors++;
    }

    printf("%s: allocated %d read-only color cells\n", progname, ncolors);

    border_pixel = colors[0];
    background_pixel = colors[1];
    foreground_pixel = colors[2];
    return(1);
}
```

4.1.7. Quand allouer des couleurs privées ?

Plusieurs cas se présentent :

- L'application dessine avec certaines couleurs qui doivent pouvoir être modifiées sans devoir re-dessiner (cyclage de couleur, modification du contraste, de la luminosité, etc...). Si on change les valeurs RGB d'une colorcell, tous les points à l'écran possèdant une pixel value correspondant à cette colorcell veront leur couleur changer.
- L'application doit utiliser des techniques d'overlays. En gros, dessiner des graphiques "par dessus" d'autres graphiques, et pouvoir les effacer facilement, à la manière d'un calque qu'on enlève. (on verra plus tard comment ça marche).
- L'application est très gourmande en couleurs (affichage d'image, synthèse d'image).

REMARQUE: l'allocation de couleurs privées n'est pas possible avec des visuals TrueColor ou StaticColor.

4.1.8. Fonctions d'allocation/manipulation de couleurs privées.

- ***XAllocColorCells***(): Cette fonction permet d'allouer des colorcells privées, dont on peut par la suite changer dynamquement les valeurs RGB. Pour allouer simplement quelques colorcells, il suffit de positionner le paramètre ncolors à la valeur désirée, et de mettre le paramètre nplanes à 0. Toutes les pixel values seront retournées dans le tableau pixels. On positionnera les valeurs RGB des colorcells allouées à l'aide des fonctions ***XStoreColor***(), ***XStoreColors***() ou ***XStoreNamedColor***().
- ***XAllocColorPlanes***(): Cette fonction ne sera pas étudiée dans l'immédiat.
- ***XStoreColor***(): permet de modifier la colorcell READ/Write correpondant à la pixel value passée en paramètre, en lui attribuant les valeurs RGB les plus proches de celles passées en paramètre dans la structure XColor. Ne pas oublier de positionner les flags ***DoRed***, ***Dogreen*** et ***DoBlue*** correspondants aux composantes RGB qui doivent être modifiées.

- ***XStoreColors***(): Idem, mais permet de modifier plusieurs colorcells d'un coup.
- ***XStoreNamedColors***(): Très semblable à ***XStoreColor***() sauf que la valeur RGB que va prendre la colorcell correspond à un nom de couleur (qui doit être dans la base de données des couleurs standards).

4.1.9. Exemple de code comprenant l'allocation de couleurs privées.

Ci-dessous un exemple d'allocation des couleurs dans une colormap privée. .

```
/*
 * Copyright 1989 O'Reilly and Associates, Inc.
 * See ../Copyright for complete rights and liability information.
 */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>

extern Display *display;
extern int screen_num;
extern unsigned long foreground_pixel, background_pixel, border_pixel;

#define MAX_COLORS 3

get_colors()
{
    int default_depth;
    Visual *default_visual;
    static char *name[] = {"Red", "Yellow", "Green"};
    XColor exact_defs[MAX_COLORS];
    Colormap default_cmap;
    int ncolors = MAX_COLORS;
    int plane_masks[1];
    int colors[MAX_COLORS];
    int i;
    XVisualInfo visual_info;
    int class;

    class = PseudoColor;
    default_depth = DefaultDepth(display, screen_num);
    default_visual = DefaultVisual(display, screen_num);
    default_cmap   = DefaultColormap(display, screen_num);

    if (default_depth == 1) {
        /* must be StaticGray, use black and white */
        border_pixel = BlackPixel(display, screen_num);
        background_pixel = WhitePixel(display, screen_num);
        foreground_pixel = BlackPixel(display, screen_num);
        return(0);
    }

    if (!XMatchVisualInfo(display, screen_num, default_depth,
                        PseudoColor, &visual_info)) {
        if (!XMatchVisualInfo(display, screen_num, default_depth,
                            DirectColor, &visual_info)) {

            /* No PseudoColor visual available at default_depth.
             * Some applications might try for a GrayScale visual
             * here if they can use gray to advantage, before
             * giving up and using black and white.
             */
```

```
        border_pixel = BlackPixel(display, screen_num);
        background_pixel = WhitePixel(display, screen_num);
        foreground_pixel = BlackPixel(display, screen_num);
        return(0);
      }
  }

  /* got PseudoColor visual at default_depth */


  /* The visual we found is not necessarily the
   * default visual, and therefore it is not necessarily
   * the one we used to create our window.  However,
   * we now know for sure that color is supported, so the
   * following code will work (or fail in a controlled way).
   */


  /* allocate as many cells as we can */
  ncolors = MAX_COLORS;

  while (1) {
     if (XAllocColorCells (display, default_cmap, False,
                        plane_masks, 0, colors, ncolors))
        break;

     ncolors--;

     if (ncolors = 0)
        fprintf(stderr, "basic: couldn't allocate read/write colors\n");
        exit(0);
  }

  printf("basic: allocated %d read/write color cells\n", ncolors);

  for (i = 0; i < ncolors; i++) {
     if (!XParseColor (display, default_cmap, name[i],
                    &exact_defs[i])) {
        fprintf(stderr, "basic: color name %s not in database", name[i]);
        exit(0);
     }

     /* set pixel value in struct to the allocated one */
     exact_defs[i].pixel = colors[i];
  }

   /* this sets the color of read/write cell */
  XStoreColors (display, default_cmap, exact_defs, ncolors);
  border_pixel = colors[0];
  background_pixel = colors[1];
  foreground_pixel = colors[2];
}
```

Le programme principal qui appelle cette fonction ***get_colors***() contient un appel à ***XQueryColor***() pour connaître les valeurs RGB des colorcells allouées par ***get_colors***() (fichiers séparés).

Le main fait également appel à ***XStoreColor***() pour modifier dynamiquement la couleur du texte qui est affiché dans la fenêtre sans avoir à le re-dessiner.

```
void main(argc, argv)
int argc;
```

```
char **argv;
{
   XColor color;
   unsigned short red, green, blue;
      .
      .
      .
   /* open display, etc... */

   color.pixel = foreground_pixel;
   XQueryColor(display, DefaultColormap(display, screen_num), &color);

   printf("red is %d, green is %d, blue is %d\n",
         color.red, color.green, color.blue);

   while (1)  {
      XNextEvent(display, &report);
      switch  (report.type) {
         .
         .
         .
         case ButtonPress:
            color.red += 5000;
            color.green -= 5000;
            color.blue += 3000;

            printf("red is %d, green is %d, blue is %d\n",
                  color.red, color.green, color.blue);

            XStoreColor(display,  DefaultColormap(display, screen_num),
                     &color);
            break;
         .
         .
         .
   }
}
```

4.1.10. Exemple d'utilisation des couleurs nommées et partagées.

On se propose d'allouer des couleurs dans la colormap système. On alloue la couleur rouge. Pour que la fenêtre en cours herite de la colormap système, il faut qu'à l'initialisation, rien ne soit spécifié concernant la colormap.

Exemple:

```
XCreateSimpleWindow(Dpy,RootWindow(Dpy,Scr),100,100,500,400,3,
BlackPixel(Dpy,Scr),WhitePixel(Dpy,Scr))
```

Avec la fonction ***XCreateSimpleWindow()*** la colormap est héritée de la fenêtre mère (ici, la RootWindow). Pour la fonction ***XCreateWindow()***, la colormap est héritée de la fenêtre mère si aucune colormap n'est spécifiée dans la structure ***XSetWindowAttributes*** passée en dernier parametre.

On doit récupérer l'identifieur de la colormap héritée:

```
Colormap cmap;

XGetWindowAttributes(Dpy, RootWindow, &attr);
Cmap = attr.colormap;
```

Deux façons de procéder: Premier exemple, on spécifie la couleur par

les trois composantes.

```
xcol.red = (255) << 8;
xcol.green = (0) << 8;
xcol.blue = (0) << 8;
if (!XAllocColor(Dpy, Cmap, &xcol))
    pixel = xcol.pixel;
else
    printf("Table de couleur système pleine\n");
```

La deuxième façon d'allouer la couleur est de l'appeller par son nom avec la fonction ***XAllocNamedColor()***, où l'on spécifie une entrée du fichier ***/usr/lib/X11/lib/rgb.txt***.

4.2. Visuals et Codage vidéo.

X est conçu pour manipuler des frame-buffers sous forme bitmap. Dans le plus simple des displays, le noir et blanc, il n'y a qu'un bit par pixel. Pour gérer les displays couleur ou monochrome (nuances de gris), plusieurs bits sont nécessaires pour coder un pixel. Il existe trois façons d'interpréter la valeur d'un pixel.

4.2.1. Codages du frame-buffer.

4.2.1.1. Codage en composantes statiques.

Dans les cartes graphiques capables d'afficher 16 millions de couleurs (24 bits sont nécessaires), la valeur du pixel est composée de trois composantes R, V, B, c'est la méthode appelée ***TrueColor***. Chaque composante s'étend de 0 à 255 (un octet non signé, donc 3*8 bits=24 bits).

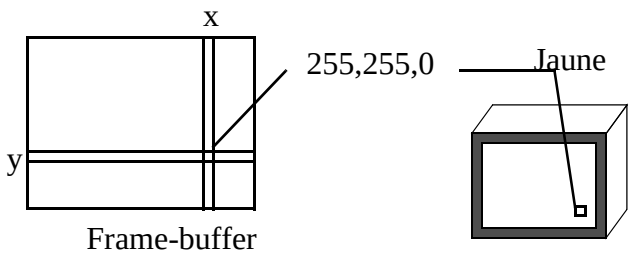


Figure 4: **Codage truecolor.**

Les cartes graphiques monochromes à nuances de gris codent la luminosité du point. Il n'y a qu'une seule composante (mode ***StaticGray***).

4.2.1.2. Codage couleur par colormap.

Exiger 3 octets pour chaque pixel demande une quantité de mémoire

assez grande pour la résolution des moniteurs courants.

On exploite souvent la deuxième solution qui consiste à n'utiliser que 8 bits (c'est un exemple) par pixels. Dans ce cas, la valeur d'un pixel n'exprime pas directement une couleur. C'est un index dans un tableau ou sont stockées des couleurs. Par exemple pour un display 8 bits, le frame-buffer est un champ 2D d'indices dans une table de couleurs.

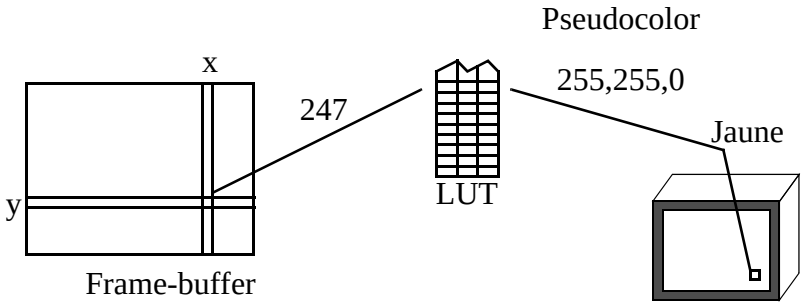


Figure 5: **Codage par colormap.**

Cette table porte le nom de colormap ou encore LUT (Loo Up Table). Dans le dernier cas on limite volontairement le nombres des couleurs disponibles (256 couleurs à trois composantes dans le cas d'un display 8 bits), mais on n'a besoin de trois fois moins de mémoire. Si l'on a le droit de changer les entrées (les trois composantes) de la colormap, on est en présence d'un visuel ***PseudoColor***. Si les entrées sont fixes (fixées par le hardware), c'est un visuel ***StaticColor***.

4.2.1.3. Codage par composantes remappées.

Ce codage est très proche de celui présenté au chapitre 4.2.1.1 . Sa nécessité tient essentiellement à deux faits.

- On peut vouloir recalibrer les composantes logiciellement.
- On veut pouvoir faire un inverse vidéo sur un écran rapidement.

Pour l'inverse-vidéo, le codage par palette est bien adapté: il suffit d'inverser les composantes de chaque entrée dans la colormap (256 opérations pour un display 8 bits). Avec le codage en composantes statiques, il est nécessaire d'inverser la valeur de tous les pixels (millions d'opérations).

Le codage par composantes remappées propose trois composantes par pixel. Mais chaque composante est un index dans une table. Ce n'est pas une

table de couleurs, mais en fait trois tables séparées.

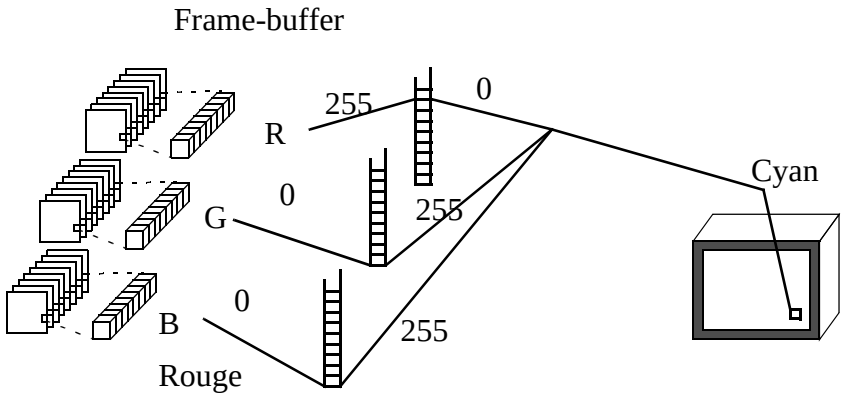


Figure 6: **Mode DirectColor.**

Finalement, les trois tables de mapping forme une seule colormap (avec des entrées à trois composantes), mais chaque composante indexe seulement son canal. C'est le mode ***DirectColor***.

Dans le cas d'un display à niveau de gris (1 composante et un table mono-composante), ce mode se nomme ***GrayScale***.

4.2.1.4. Résumé pour les modes sans colormap.

En ce qui concerne les modes sans colormap, on trouve le mode monochrome (nuances de gris) qui code la luminosité sur 1 (noir et blanc), 2 (4 niveaux), 4 (16 niveaux) ou 8 bits (256 niveaux). Le mode truecolor gère la couleur sans colormap. La précision du codage se fait généralement entre 12 (16 niveaux par composantes R, G et B) et 24 bits (8 bits soit 256 niveaux par composante). Le 24 bits est le plus répandu. Une autre déclinaison est également utilisée, c'est le 16 bits qui code deux composantes sur 5 bits (32 niveaux) et la troisième sur 6 bits. Il est équivalent au fameux mode 65535 couleurs de windows.

4.2.1.5. Résumé pour les modes avec colormap.

La taille de la LUT est définie par de le nombre de bitplanes disponibles. Généralement on a 8 bits et 256 couleurs. Les couleurs de la LUT sont définies sur trois composantes R,V et B si l'on est en couleur (8 bits pour chacune généralement), mais ce peut être une seule composante si l'on travaille en monochrome.

4.2.1.6. Résumé des résumés: Les visuals.

On sent bien avec ces exemples, la complexité de la gestion de la couleur pour X. On attribue des noms précis à ces classes de gestion de la

couleur.

Table 1: Classes de visuals.

Colormap	Lecture/Ecritue	Lecture seule
Monochrome/Gris	GrayScale	StaticGray
Colormap	PseudoColor	StaticColor
Décomposition	DirectColor	TrueColor

4.2.2. Comment X décrit le support couleur avec les Visuals.

Un visual décrit les caractèristiques d'une colormap destinée à être utilisée sur un type d'écran donné. Pratiquement, il s'agit d'un pointeur sur une structure de type ***Visual*** qui contient des informations concernant un moyen d'utiliser un écran donné.

Il est nécessaire de spécifier un visual lors de la création d'une colormap ou de la création d'une fenêtre, et le même visual doit être utilisé lors de ces deux création si la colormap va être associée à la fenêtre. La plupart des fenêtres héritent du visual; bien souvent, lors de la création d'une fenêtre on utilise la macro ***DefaultVisual***() qui retourne le visual de la RootWindow, et donc on utilise la colormap par défaut. Une fenêtre crée avec ***XCreateSimpleWindow()*** utilise la colormap par défaut.

La structure Visual est opaque. On ne peut accèder aux informations qu'elle renferme directement.

Rappel: un visual décrit une manière d'utiliser la couleur sur un écran, mais un écran peut supporter plusieurs visuals.

Par exemple, sur un système couleur, on peut utiliser à la fois un visual monochrome et un visual couleur. Obtenir des informations concernant les visuals supportés par un écran

Deux fonctions sont disponibles : ***XMatchVisualInfo()*** et ***XGetVisualInfo()***. Elles renvoient une structure de type ***XVisualInfo*** qui est publique (contrairement à la structure ***Visual*** qui est opaque).

Le champ class de la structure XVisualInfo correspond à une des six classes possibles: ***DirectColor***, ***GrayScale***, ***PseudoColor***, ***StaticColor***, StaticGray et ***TrueColor***.

On peut connaître les visuals supportés par un écran donné à l'aide du programme ***xdpyinfo***.

En pratique, on utilisera le plus souvent le ***DefaultVisual()*** qui est celui qui correspond le plus souvent aux besoins usuels et qui exploite au mieux le hardware dont on dispose.

Les visuals ***DirectColor***, ***GrayScale*** et ***PseudoColor*** ont des colormaps

modifiables (***Read/Write***).

Les visuals ***StaticColor***, ***StaticGray*** et ***TrueColor*** ont des colormaps immuables (***Read-Only***).

Attention: les colormaps modifiables (Read/Write) ont deux types de colorcells:

- Read/Write: la couleur de la colorcell peut être changée à n'importe quel moment par le client qui l'a allouée. Elle ne peut être partagée.
- Read-Only: allouée une fois par un client, peut être partagée mais plus modifiée.

Avantages des colormaps immuables:

- Elles peuvent être partagées.
- Le calcul direct de la pixel value est possible, sans passer par le serveur, puisque la correspondance entre la pixel value et la couleur est prévisible (il suffit de connaître la colormap).

Désavantages des colormaps immuables:

- La couleur désirée n'est peut-être pas disponible. Dans ce cas, impossible de la créer: les colorcells en Read-Only uniquement.

Avantages des colormaps modifiables:

- On peut avoir à la fois des colorcells immuables et modifiables.
- C'est ce qui rend les visuals PseudoColor et DirectColor les plus utiles.

Désavantages des colormaps modifiables:

- Les colorcells en Read/Write ne sont pas partageables.

REMARQUE: la colormap par défaut contient des colorcells en Read-Only et en Read-Write. Si une application ne trouve pas dans cette colormap toutes les couleurs dont elle a besoin, elle pourra modifier, si elles sont disponibles, les colorcells en Read/Write. Il faut néanmoins remarquer que ces colorcells ne seront plus disponibles pour les autres applications tant qu'elles ne seront pas libérées. Dans ce cas, la seule solution pour l'application en manque de couleurs consiste à allouer une nouvelle colormap (une colormap privée). C'est ce que font certaines applications comme XV.

4.3. Les pixmaps.

4.3.1. Introduction

Nous avons déjà parlé dans les premiers chapitres de ce cours de Pixmaps. Il s'agit en effet de "l'autre" drawable (avec les objets de type ***Window***) dans lequel on peut utiliser les fonctions de dessin de la Xlib comme ***XDrawLine(display, drawable, gc, x1, y1, x2, y2)*** .

Un pixmap est un buffer mémoire qui peut être assimilé à une fenêtre graphique virtuelle. On peut dessiner dedans, copier son contenu dans une fenêtre, dans un autre pixmap, effacer son contenu, etc…

Les pixmaps ont plusieurs utilités :

- Optimiser la gestion des événements Expose. Si on écrit une application qui affiche des images, il suffit de lire l'image dans un Pixmap et copier le contenu du Pixmap dans une fenêtre pour afficher l'image. Mieux, X11 permet d'éviter de redessiner totalement une fenêtre si seulement une partie de celle-ci doit être raffraichie : chaque événement Expose contient les coordonnées du rectangle à redessinner. Il suffit dans ce cas de recopier la zone correspondante du pixmap vers la fenêtre.
- Faire de l'animation avec double-buffer, sans scintillements. On dessine dans le Pixmap, on recopie dans la fenêtre, on redessine, on recopie, etc... C'est comme au cinéma!
- Créer une fenêtre ayant un motif en fond (attribut ***background_pixmap*** des fenêtres).
- Faire du remplissage avec des motifs. Nous allons voir qu'à l'aide des Pixmaps et des contextes graphiques il est possible de remplir un rectangle contenant par exemple un damier (avec la fonction ***XFillRectangle(display, drawable, gc, x, y, width, height)*** tout simplement).
- Sauvegarder sur disque le contenu d'une fenêtre….

4.3.2. Mettre un bitmap en fond d'une fenêtre.

Rappel : un bitmap est en noir et blanc, un pixmap peut comporter de nombreuses couleurs.

Pour créer un bitmap, le plus simple est d'utiliser le programme bitmap. Ce programme permet de dessiner à l'aide d'outil très simples des motifs ou

des curseurs et de les sauvegarder dans un fichier.

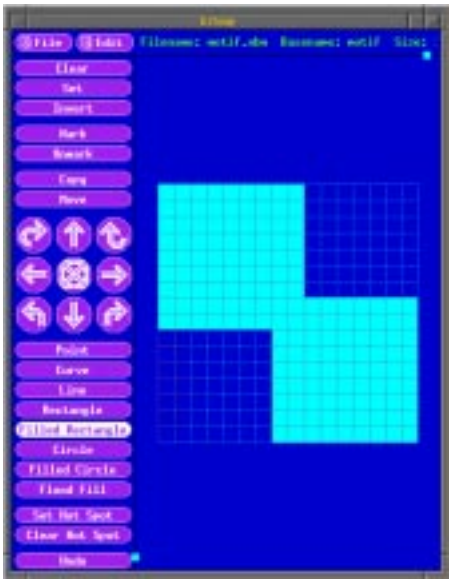


Figure 7: **Le programme bitmap.**

Par la suite on pourra soit lire directement le fichier à l'aide de la fonction ***XReadBitmapFile***() soit l'inclure directement dans le code avec un ***#include*** (déconseillé). Dans tous les cas, pour les manipuler il faudra stocker leur contenu dans une variable de type Pixmap.

La Xlib ne fournit pas de fonctions permettant de lire des motifs comportant plus de deux couleurs. Pour cela il faudra recourir à des librairies spécialisées comme la librairie ***Xpm*** ou la librairie ***ImageMagick***

4.3.2.1. Premier exemple : le bitmap est lu dans un fichier.

```
#include <stdio.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>

Display *dpy;
int screen;
Window root;
Visual   *visual;
int depth;
int fg, bg;

main()
{
  Window win;
  XSetWindowAttributes xswa;
  XEvent report;

  GC gc;
  XGCValues gcvalues;

  Pixmap motif;
  Pixmap motif_8bits;
  int val;
```

```
int motif_width, motif_height,x_hot,  y_hot;

if((dpy=XOpenDisplay(NULL))==NULL) {
  fprintf(stderr,"fenetre de base: ne peut pas se connecter ");
  fprintf(stderr,"au serveur %s\n",XDisplayName(NULL));
  exit(-1);
}

/* Initialisation des variables standards */

screen = DefaultScreen(dpy);
root = DefaultRootWindow(dpy);
visual = DefaultVisual(dpy, screen);
depth = DefaultDepth(dpy,screen);
fg = BlackPixel(dpy, screen);
bg = WhitePixel(dpy, screen);

/* Lecture du motif de 1 bit de profondeur (bitmap) */

val = XReadBitmapFile(dpy, root,
                "motif.xbm",              /* Nom du fichier */
                &motif_width, &motif_height, /* largeur et hauteur */
                &motif,                   /* le pixmap */
                &x_hot, &y_hot);          /* le "hot spot */
switch(val) {
case BitmapFileInvalid:
  fprintf(stderr, "le fichier motif.xbm n'est pas un bitmap valide\n");
  break;
case BitmapOpenFailed:
  fprintf(stderr, "le fichier motif.xbm n'a pu être lu\n");
  break;
case BitmapNoMemory:
  fprintf(stderr, "Pas assez de mémoire pour lire le fichier motif.xbm\n");
  break;
case BitmapSuccess:
  fprintf(stderr, "Fichier motif.xbm lu avec succes. Taille = %d %d, hotspot = %d %d\n",
        motif_width, motif_height, x_hot, y_hot);
  break;
}

/* Creation d'un second pixmap de la meme profondeur */

motif_8bits = XCreatePixmap(dpy,
                      root,
                      motif_width, motif_height,
                      depth);


/* La fonction suivante a besoin d'un GC */

gcvalues.foreground =  fg;
gcvalues.background =  bg;

gc =  XCreateGC(dpy,  RootWindow(dpy, screen),
            (GCForeground | GCBackground), &gcvalues);


/* On copie le bitmap dans le pixmap_8 bits */

val = XCopyPlane(dpy, motif, motif_8bits, gc, 0, 0, motif_width,
              motif_height, 0, 0, 1);

/* attributs de la fenetre. */

/* Le motif en fond. INCOMPATIBLE avec background_pixel !!!*/

xswa.background_pixmap = motif_8bits;

/* Les événements */

xswa.event_mask = ExposureMask|ButtonPressMask|ButtonReleaseMask|
  PointerMotionMask|KeyPressMask;

/* Couleur de la bordure */

xswa.border_pixel     = fg;
```

```
 win = XCreateWindow(dpy, root,
                     100, 100, 500, 500, 3,
                     depth,
                     InputOutput,
                     visual,
                     CWEventMask|CWBorderPixel|CWBackPixmap,
                     &xswa);
 XMapWindow(dpy,win);


 /* On libere les pixmaps car maintenant ils sont dans la memoire du serveur X11 */
 XFreePixmap(dpy, motif);
 XFreePixmap(dpy, motif_8bits);

 while(1) {
   XNextEvent(dpy, &report);
   ...
 }
}
```

Ce qu'il faut retenir de ce petit exemple :

- Un bitmap = deux couleurs. Pour le lire, il faut utiliser la fonction ***XReadBitmapFile***() qui alloue et initialise un Pixmap de 1 bit de profondeur. Pour pouvoir l'utiliser comme fond d'une fenêtre n'ayant pas la même profondeur, il faut recopier ce pixmap dans un autre pixmap ayant la même profondeur que la fenêtre. On utilise pour celà la fonction ***XCopyPlane***() en positionnant le dernier paramètre à 1 (un seul bitplane à copier).
- Ne pas oublier de libérer les Pixmaps alloués si on en a plus besoin. Sitôt la fenêtre créée, ses attributs sont dans la méloire du serveur. On libère donc les Pixmaps juste après à l'aide de ***XFreePixmap***().
- ATTENTION : si vous positionnez en plus l'attribut ***background_pixel*** de la fenêtre, l'attribut ***background_pixmap*** ne sera pas pris en compte. Ces deux attributs sont exclusifs mais rien ne vous empêche de les positionner. Faites attention car c'est le ***background_pixel*** qui gagne!

4.3.2.2. Deuxième exemple : le bitmap est inclu dans le source.

```
#include <stdio.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include "motif.xbm"

...

main()
{
```

```
...
 if((dpy=XOpenDisplay(NULL))==NULL) {
   fprintf(stderr,"fenetre de base: ne peut pas se connecter ");
   fprintf(stderr,"au serveur %s\n",XDisplayName(NULL));
   exit(-1);
 }

 /* Initialisation des variables standards */
 screen = DefaultScreen(dpy);
 ...
 motif = XCreateBitmapFromData(dpy, root,
                      motif_bits, motif_width, motif_height);

 /* Creation d'un second pixmap de la meme profondeur */
 motif_8bits = XCreatePixmap(dpy,
                      root,
                      motif_width, motif_height,
                      depth);
```

En gras, les modifications à apporter à l'exemple précédent.

Quelques remarques cependant :

- ***motif_bits***, ***motif_width*** et ***motif_height*** sont maintenant des macros qu'il ne faut pas déclarer. Elles proviennent du fichier ***motif.xbm*** qui est inclu. Voici à quoi il ressemble :

```
static unsigned char motif_bits[] = {
0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01,
0xff, 0x01, 0xff, 0xff, 0xff, 0xff, 0x80, 0xff, 0x80, 0xff, 0x80, 0xff,
0x80, 0xff, 0x80, 0xff, 0x80, 0xff, 0x80, 0xff
};
```

- L'intéret principal de cette méthode est qu'elle facilite la distribution de programmes compilés. Je la déconseille dans tous les autres cas.

4.3.3. Utiliser un bitmap comme motif de remplissage.

Une fois le pixmap alloué et initialisé (s'assurer qu'il a la même profondeur que la fenêtre, comme dans les deux exemples précédents), il est très facile de l'utiliser comme motif pour dessiner. Il suffit pour celà de créer un GC avec les attributs suivants positionnés :

- ***fill_style*** doit valoir ***FillTiled*** (d'autres subitilités sont disponibles en le positionnant avec les valeurs ***FillStippled*** ou ***FillOpaqueStippled***, voir le manuel).
- ***tile*** doit être initialisé avec l'ID du pixmap.

... ou de modifier un GC existant à l'aide des fonctions ***XSetFillStyle***() et ***XSetTile***().

4.3.3.1. Exemple d'utilisation d'un motif pour dessiner.

```
#include <stdio.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>

...

main()
{
 ...
/* Creation d'un second pixmap de la meme profondeur */
 motif_8bits = XCreatePixmap(dpy,
                        root,
                        motif_width, motif_height,
                        depth);
 gcvalues.foreground =  fg;
 gcvalues.background =  bg;

 gc =  XCreateGC(dpy,  RootWindow(dpy, screen),
            (GCForeground | GCBackground), &gcvalues);

 /* On copie le bitmap dans le pixmap_8 bits */
 val = XCopyPlane(dpy, motif, motif_8bits, gc, 0, 0, motif_width,
          motif_height, 0, 0, 1);
 ...
 /* creation de la fenêtre, etc... */

 ...
 XSetFillStyle(dpy, gc, FillTiled);
 XSetTile(dpy, gc, motif_8bits);

 XMapWindow(dpy,win);

 XFreePixmap(dpy, motif);
 XFreePixmap(dpy, motif_8bits);

 while(1) {
  XNextEvent(dpy, &report);
  switch (report.type) {
  case Expose:
    /* traitement des évènements de type Expose */

    break;
  case ButtonPress:
/*Quand on clique on dessine un rectangle rempli avec le motif */
    XFillRectangle(dpy, win, gc, report.xbutton.x,
              report.xbutton.y, 100, 100);
    break;
  case KeyPress:
    XCloseDisplay(dpy);
    exit(0);
    /* traitement des évènements de type KeyPress */

    break;
  case ConfigureNotify:
    /* traitement des évènements de type ConfigureNotify */

    break;
  }
 }
}
```

4.3.4. Utilisation d'un pixmap pour faire du double buffering.

Pas grand chose à expliquer car la méthode est très simple : on dessine dans un pixmap (et non pas dans la fenêtre), puis on copie d'un coup le

pixmap dans la fenêtre.

Le petit exemple ci-dessous montre un exemple d'utilisation d'un Pixmap en tant que double buffer. Il permet en outre de comparer la même animation (cent cercles concentriques qui avancent en diagonale à partir de la position cliquée) avec et sans double buffer.

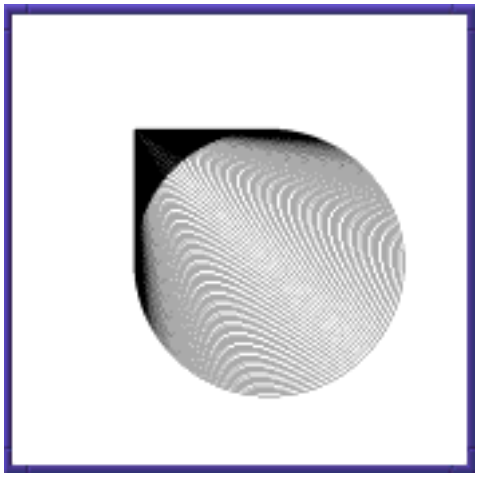


Figure 8: **Le programme d'exemple du double buffer.**

4.3.4.1. Programme d'exemple.

```
#include <stdio.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include "motif.xbm"

Display *dpy;
int screen;
Window root;
Visual *visual;
int depth;
int fg, bg;

main()
{
  Window win;
  int width = 500, height = 500;
  XSetWindowAttributes xswa;
  XEvent report;

  GC gc;
  XGCValues gcvalues;

  Pixmap db;
  int i,j;

  if((dpy=XOpenDisplay(NULL))==NULL) {
    fprintf(stderr,"fenetre de base: ne peut pas se connecter ");
    fprintf(stderr,"au serveur %s\n",XDisplayName(NULL));
    exit(-1);
  }

  /* Initialisation des variables standards */
```

```
screen = DefaultScreen(dpy);
root = DefaultRootWindow(dpy);
visual = DefaultVisual(dpy, screen);
depth = DefaultDepth(dpy,screen);
fg = BlackPixel(dpy, screen);
bg = WhitePixel(dpy, screen);

/* Creation du double buffer */

db = XCreatePixmap(dpy, root, width, height, depth);

gcvalues.foreground =  fg;
gcvalues.background =  bg;

gc =  XCreateGC(dpy,  RootWindow(dpy, screen),
            (GCForeground | GCBackground), &gcvalues);



/* attributs de la fenetre. */

/* Le motif en fond */

xswa.background_pixel = bg;

/* Les événements */

xswa.event_mask = ExposureMask|ButtonPressMask|ButtonReleaseMask|
  PointerMotionMask|KeyPressMask;

/* Couleur de la bordure et du background */

xswa.background_pixel = bg;
xswa.border_pixel     = fg;

win = XCreateWindow(dpy, root,
                    100, 100, 500, 500, 3,
                    depth,
                    InputOutput,
                    visual,
                    CWEventMask|CWBorderPixel|CWBackPixel,
                    &xswa);

XMapWindow(dpy,win);

while(1) {
  XNextEvent(dpy, &report);
  switch (report.type) {
  case Expose:
    /* traitement des évènements de type Expose */

    break;
  case ButtonPress:
    switch(report.xbutton.button) {
    case Button1:
      /* Animation avec double buffer */

      for(j=0; j < 100; j++) {
        XSetForeground(dpy, gc, bg);
        XFillRectangle(dpy, db, gc, 0, 0, width, height);
        XSetForeground(dpy, gc, fg);

        for(i = 0; i < 400; i+=4)
          XDrawArc(dpy, db, gc, report.xbutton.x+j, report.xbutton.y+j,
                i,i, 0, 23040);

        XCopyArea(dpy, db, win, gc, 0, 0, width, height, 0, 0);
         /* on force le rafraichissement de l'écran */

        XFlush(dpy);
      }

      break;
    case Button2:
      /* Animation sans double buffer */
```

```
      for(j=0; j < 100; j++) {
        XClearWindow(dpy, win);

        for(i = 0; i < 400; i+=4)
          XDrawArc(dpy, win, gc, report.xbutton.x+j, report.xbutton.y+j,
                i,i, 0, 23040);
      }
      break;
    }
    break;
  case KeyPress:
    XCloseDisplay(dpy);
    exit(0);
     /* traitement des évènements de type KeyPress */

    break;
  case ConfigureNotify:
     /* traitement des évènements de type ConfigureNotify */

    break;
  }
 }
}
```

4.3.4.2. Etude de l'exemple.

- Pour réaliser une animation en double buffer, on commence par effacer le contenu du double buffer en dessinant sur la totalité du pixmap un rectangle de la couleur du fond. Ensuite, on effectue le dessin proprement dit, non pas dans la fenêtre mais directement dans le pixmap :

```
XDrawArc(dpy, db, gc, report.xbutton.x+j, report.xbutton.y+j, eport.xbutton.y+j i, i, 0, 23040);
```

- On recopie le pixmap dans la fenêtre :

```
XCopyArea(dpy, db, win, gc, 0, 0, width, height, 0, 0);
```

- On force le serveur X11 à "flusher", c'est à dire à exécuter toutes les requêtes qui sont dans sa pile. N'oublions pas que habituellement les requêtes sont bufférisées. C'est l'équivalent du ***fflush***() de la librairie standard ***stdio***.

```
XFlush(dpy);
```

- Dans le cas normal, on efface la fenêtre, et on dessine directement dans la fenêtre. Le dessin étant très long, on a presque le temps de voir les cercles se dessiner. L'effet est désastreux.

4.3.5. Optimisation des Expose.

Si on dessine directement dans un Pixmap celà permet de gérer facilement le rafraichissement de la fenêtre. En effet, si cette dernière est recouverte par une autre fenetre puis découverte, il est très facile de la rafraichir puisqu'on dispose d'une copie de son contenu dans le pixmap.

4.3.5.1. Méthode “tout en un” :

Si l’on n’est pas à la recherche de performances ou si l’application n’est pas très gourmande en ressources, le plus simple est de recopier intégralement le contenu du pixmap dans la fenêtre, comme nous l’avons fait pour dans l’exemple du double buffer.

```
...

while(1) {
 XNextEvent(dpy, &report);
 switch (report.type) {
 case Expose:
   /* On efface tous les Expose de la pile */

   while(XCheckTypedEvent(dpy, Expose, &report));
   XCopyArea(dpy, db, win, gc, 0, 0, width, height, 0, 0);
    /* on force le rafraichissement de l’écran */

   XFlush(dpy);

   break;
 case ButtonPress:
...
```

Puisque on recopie l’intégralité du pixmap dans la fenêtre, il faut purger les événements Expose qui traînent encore dans la pile. Inutile de redessiner plusieurs fois la même chose!

Rappelons que plusieurs événements de type Expose peuvent être générés dans le cas d’une modification de la taille de la fenêtre ou d’un recouvrement partiel multiple, un événement Expose est généré pour chaque partie rectangulaire à redessiner.

Pour supprimer des événements d’un type donné de la pile des événements on utilise la fonction ***XCheckTypedEvent***()

4.3.5.2. Méthode “normale”.

On traite les Expose un à un en ne redessinant que les parties rectangulaires correspondantes.

```
...

while(1) {
 XNextEvent(dpy, &report);
 switch (report.type) {
 case Expose:
   XCopyArea(dpy, db, win, gc,
             report.xexpose.x, report.xexpose.y,
             report.xexpose.width, report.xexpose.height,
             report.xexpose.x, report.xexpose.y);
   break;
 case ButtonPress:
...
```

C’est effectivement plus simple, mais n’oublions pas que dans ce cas on va passer plusieurs fois dans le case dans le cas d’exposition multiple.

5. Les images.

5.1. Introduction.

Les images sont representées sous X11 par une structure de type ***XImage***:

```
typedef struct _XImage
{
    int width, height;        /* size of image */
    int xoffset;             /* number of pixels offset in X direction */
    int format;               /* XYBitmap, XYPixmap, ZPixmap */
    char *data;               /* pointer to image data */
    int byte_order;           /* data byte order, LSBFirst, MSBFirst */
    int depth;               /* depth of image */
.
.
.
    int bytes_per_line;       /* accelarator to next line */
    int bits_per_pixel;       /* bits per pixel (ZPixmap) */
    int (*put_pixel)();
    struct _XImage *(*sub_image)();
} XImage;
```

Il ne faut pas confondre ***Pixmap*** et ***XImage***. Le premier est un identifieur d'une zone mémoire dans la mémoire du serveur (l'équivalent d'une fenêtre), alors qu'une ***XImage*** est une zone de mémoire dans le client ou sont stockées des informations.

A la différence des Pixmaps qui se trouvent dans la mémoire du serveur, les XImages sont dans la mémoire du client (ou mieux, nous allons voir!). Ainsi, on peut les manipuler directement à coup de ***memset***(), ***memcopy***() et autres ****ximage++***!

Rappelons que les Pixmaps ne peuvent être manipulés qu'au travers de fonctions Xlib et donc au travers du protocole X11.

On peut allouer une XImage dans une zone de mémoire partagée directement visible par le serveur X.

5.2. Programmation.

Pour visualiser une XImage il faut recopier son contenu dans un drawable (Pixmap, Window). Xlib fournit deux fonctions d'interfaçage entre drawables et Images : ***XGetImage***() et ***XPutImage***().

Une ***XImage*** est un champ 2D de valeurs. Dans le cas des visuels à 256 couleurs, ces valeurs sont des indices dans la colormap.

Il existe trois formats de ***XImage***:

- ***XYBitmap***: la zone de mémoire décrite par une XImage de ce type est en fait un plan de bits (bitplane). L'état (allumé ou éteint) du pixel est donné par la position du bit concerné.
- ***XYPixmap***: La zone mémoire de ce mode est une concaténation de plusieurs bit-planes (8), ceci ressemble au codage des frame-buffers.
- ***ZPixmap***: le plus couramment utilisé, c'est en fait un mode chunky-pixel (voir annexes). Un octet de la zone mémoire est un index dans la colormap.

Concretement, il faut allouer une XImage avant de l'utiliser. Pour ceci on utilise la fonction: ***XImage *XCreateImage(Display *display, Visual *visual, unsigned int depth, int format, int offset, char *data, unsigned int width, unsigned int height, int bitmap_pad, int bytes_per_line)***

Un exemple:

```
XImage        *ximage;
char * Data;
int nppl=512, nl=512;

/*
    Data=malloc(nppl*nl);
    fread(data,…,file);/* chargement d'une image brute */
 */

ximage = XCreateImage(Dpy, visual, 8,ZPixmap,0,Data,nppl, nl,8, 0);
```

Les argument ***Dpy*** et ***visual*** sont classiques. Des renseignements sur la taille et la profondeur de l'image sont spécifiés. Attardons nous sur l'argument Data. Data pointe sur une zone de mémoire linéaire (de taille ***nppl*nl***). La structure XImage est donc une structure de controle du buffer d'octets (Data) que l'on passe en paramètre.

Cette image est affichée (mémoire copiée de la mémoire du client vers la zone mémoire de la fenêtre dans l'espace mémoire du serveur). La fonction ***XPutImage()*** réalise ce transfert.

5.2.1. Autres fonctions d'exploitation des XImages.

- ***XGetImage***(...) et ***XGetSubImage***(...)Remplit une XImage à partir d'un drawable (window ou pixmap).
- ***XPutImage***(display, image, drawable, x_src, y_src, x_dest, y_dest, width, height): Recopie une XImage dans

un drawable. Nécessaire pour rendre une XImage visible.

- ***XDestroyImage***(ximage): Libère la mémoire allouée pour l'XImage. Attention, si l'XImage a été créée avec XCreateImage, XGetImage ou XGetSubImage, cette fonction libère à la fois le buffer et la structure d'encapsulation.
- ***XGetPixel***(ximage, x, y) et ***XPutPixel***(ximage, x, y, pixel_value) : Macros permettant de lire ou d'écrire un pixel dans une XImage. ***XGetPixel***() renvoie la pixel-value du point passé en paramètre, ***XPutPixel***() permet de colorer un pixel avec une ***pixel_value***.

5.2.2. Exemples.

5.2.2.1. Allocation d'une image.

```
/* Alloue une XImage de width pixels de large * height pixels de haut */

    char *buffer = NULL;
    XImage *XIma = NULL;

/* On alloue donc le buffer qui va contenir les graphiques manuellement a l'aide d'un malloc
*/

    if (buffer=(char *) malloc(width * height*sizeof(char))
    {
        perror("malloc failed \n");
        exit(0);
    }

/* Allocation d'une XImage à partir du buffer, plus rapide pour le dessin et l'affichage qu'un
Pixmap */
    XIma=XCreateImage(display, visual ,8, ZPixmap ,0 ,(char *) buffer, width,height,8,0);

     ...

/* On remplit avec la pixel value 0 le point (x, y) */
    buffer[x*y+x] = 0;

/* On affiche le buffer dans une fenêtre */
     XPutImage(dpy, win, gc, XIma, 0, 0, 0, 0, width, height);
```

5.2.2.2. Commentaire.

Dans cet exemple, on alloue d'abord un buffer standard puis on l'encapsule dans une XImage. Le buffer est utilisé directement pour dessiner :

```
buffer[x*y+x] = 0;
```

On aurait pu aussi bien utiliser la fonction XPutPixel() de la manière suivante.

```
XPutPixel(XIma, x, y, 0);
```

Je pense que les deux écritures sont strictement équivalentes, cependant il est avantageux d'attaquer directement le buffer dans le cas d'utilisation de fonctions de type ***memset***() ou ***memcpy***(), utiles pour remplir des zones de mémoire rapidement.

Enfin, l'affichage se fait en recopiant l'XImage dans une fenêtre :

```
XPutImage(dpy, win, gc, XIma, 0, 0, 0, 0, width, height);
```

6. Annexes.

6.1. Codage de Frame-Buffer

6.1.1. Mode bitmap.

Dans le mode bitmap, la valeur du pixel est exprimée par un empilement de bitplanes dans le frame-buffer.

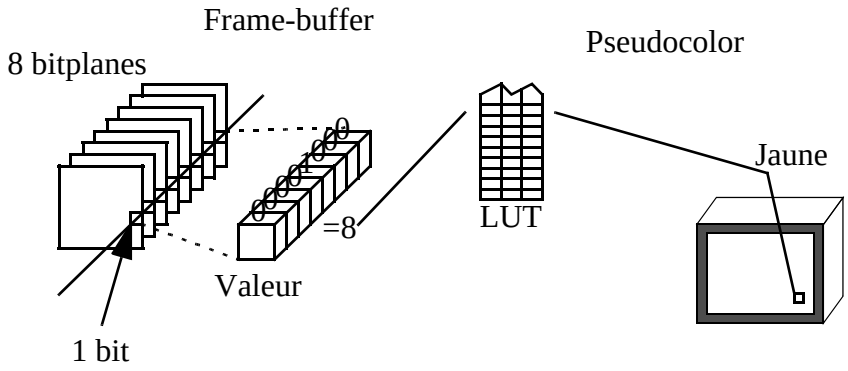


Figure 9: **Codage d'un pixel en mode bitmap.**

Ce mode bitmap possède plusieurs atouts. D'un part, il complètement dynamique du point de vue de l'allocation de la mémoire vidéo. En effet, si on veut disposer de seulement 32 couleurs, 5 plans suffisent. Un octet dans un bitplane code une partie des valeurs de 8 pixels contigus sur une ligne. D'autre part, plusieurs effets sont possibles sans trop d'efforts (suppression de certains plans, mélange de plans).

Dans l'exemple de la Figure 9, pour coder un mode ***truecolor***, il faudrait 24 bitplanes.

Néanmoins, une image vidéo composée de bitplanes a forcement une largeur multiple de 8 (très petit inconvénient).

Le deuxième inconvénient est le fait que pour accéder à la valeur d'un point, il faut aller consulter 1 bit dans plusieurs bitplanes.

Ce codage de frame-buffer est le plus couramment utilisé.

6.1.2. Mode chunky-pixel.

Ce mode de codage des valeurs de frame-buffer est utilisé dans les consoles de jeux-vidéo (très peu paramétrables).

Ici, le frame buffer est un champ 2D de valeurs. Alors qu'en mode bitmap, un octet codait une partie de 8 pixels contigus, le mode chunky-pixel propose un octet/un pixel.

Cela peut paraître plus naturel mais ce mode est peu répandu.

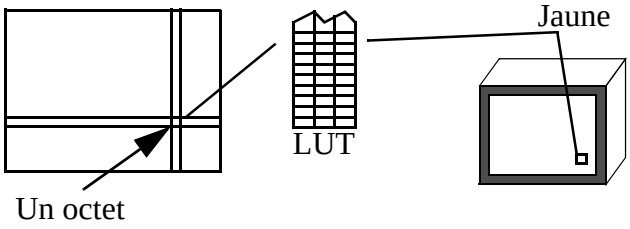


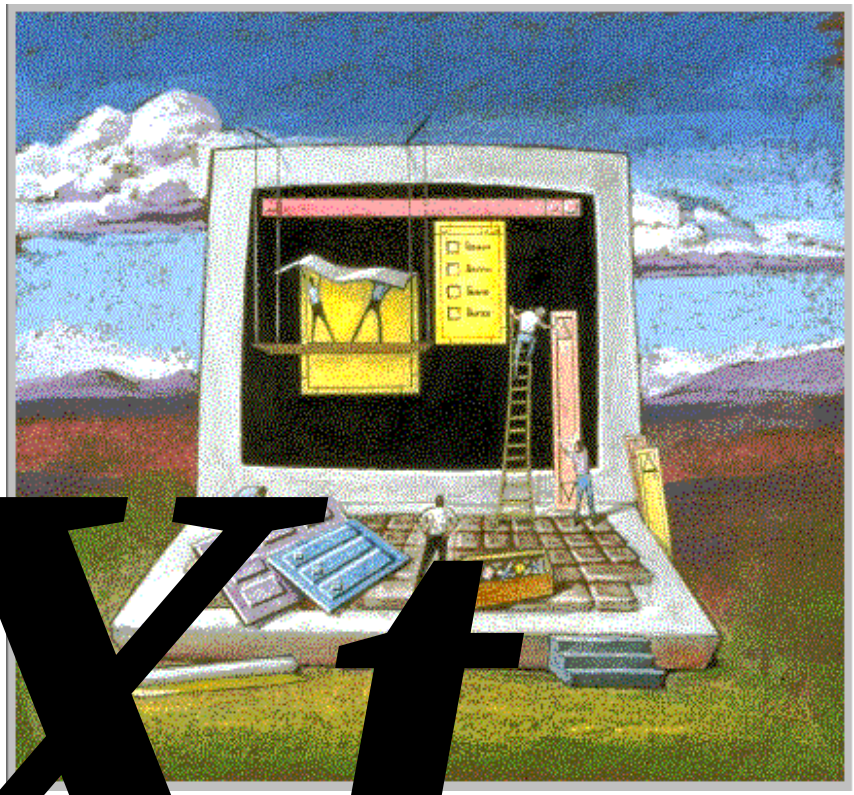
Figure 10: **Codage chunky-pixel.**

L'inconvénient de ce mode réside dans le fait que si l'on n'a pas un display 8 bits (un octet pour un pixel), une valeur peut être "à cheval" sur deux octets physiques en mémoire.

X-Intrinsics

Toolkit Intrinsics Xt

Franck Diard, Michel Buffa, François Abram, 1998
diard@i3s.unice.fr, http://www.essi.fr/~diard
abram@i3s.unice.fr, http://www.essi.fr/~abram
V 1.4. Mars 1998.



Xt

1. Introduction.

La librairie Xt (aussi appelée “librairie Intrinsics”, ou encore “les Intrinsics”) propose des fonctions spécialisées dans la construction d’interfaces graphiques au-dessus du système de fenêtrage X Window. Une boîte à outils graphiques, par exemple Motif, se compose de la librairie Xt et d’un ensemble de fonctions décrivant des boutons, menus, ascenseurs, etc... que l’on appelle plus généralement des Widgets. Les Intrinsics fournisssent les mécanismes de base nécessaires à la construction d’une large variétée de Widgets et sont construits au-dessus de la Xlib. Ils étendent les concepts abstraits de X Window (objets, évènements, etc...) mais demeurent indépendants du look des boîtes à outils qu’ils permettent de construire. ***Motif***, ***Athena***, ***Openlook*** ne se ressemblent pas mais utilisent pourtant tous les Intrinsics.

Les Intrinsics utilisent des techniques de programmation orientée objet permettant la construction homogène des éléments d’une interface graphique (Widgets): les programmeurs peuvent créer de nouveaux Widgets soit à partir de zéro, soit en dérivant des Widgets existants (mécanisme de surclassage).

Lorsque les Intrinsics furent conçus, la racine de la hierarchie des objets était une classe de Widgets nommée ***Core***. Dans la version 4 des Intrinsics, trois superclasses ont été crées au-dessus de la classe ***Core***. Le nom de la classe racine est aujourd’hui Object.

Les Intrinsics ont été crées dans deux buts précis :

- 1.Créer des Widgets.
- 2.Programmer des applications avec un toolkit.

Les programmeurs qui désirent créer de nouveaux Widgets vont utiliser la plupars des fonctions de la librairie Intrinsics pour créer tout types de Widgets, du plus simple (bouton poussoir par ex.) au plus compliqué (requester de choix de fichier).

Les programmeurs d’applications ne vont utiliser qu’un petit sous-ensemble des Intrinsics, en collaboration avec une librarie décrivant un ensemble de Widgets. Par exemple, pour créer une application à l’aide de la Toolkit Motif, il faudra appeler aussi bien des fonctions Motif que des fonctions des Intrinsics.

Les applications qui utilisent les Intrinsics doivent inclure **<*X11/Intrinsic.h*>** et **<*X11/StringDefs.h*>**, optionnellement **<*X11/Xatoms.h*>** et **<*X11/Shell.h*>**. Elles doivent aussi inclure un fichier pour chaque classe de Widget utilisée (par exemple, **<*X11/Xaw/Label.h*>** ou **<*X11/Xaw/Scrollbar.h*>**).

Les sources qui implémentent de nouveaux Widgets doivent inclure **<*X11/IntrinsicP.h*>** au lieu de **<*X11/Intrinsic.h*>**. Sur les systèmes Unix, la librairies des Intrinsics se nomme ***/usr/lib/X11/libXt.a***. Pour l'utiliser on effectue l'édition de liens avec l'option ***-lXt***.

2. Widgets.

2.1. Présentation.

L'abstraction fondamentale, l'objet de base d'une boîte à outils (un Toolkit) est le Widget. Un Widget est la combinaison d'une ou plusieurs fenêtres X (son look) et d'un comportement (entrées/sorties, sémantique d'affichage). Il est alloué dynamiquement et posséde un état. Certains Widgets affichent de l'information (par exemple du texte ou des graphiques), d'autres sont juste des containers pour d'autres Widgets (par exemple une barre de menu). Certains Widgets ne permettent que des sorties et ne réagiront pas aux évènements clavier ou souris, d'autres modifieront leur apparence en fonction des entrées ou exécuteront des fonctions utilisateur qui leur sont attachées (bouton poussoir).

Chaque Widget appartient exactement à une classe, statiquement allouée et initialisée, qui contient la liste des opérations possibles sur tous les Widgets lui appartenant. D'un point de vue logique, Une classe de Widget est un ensemble de procédures (avec les valeurs autorisées pour les paramètres de ces procédures) associées aux Widgets de la classe. Ces procédures peuvent être héritées des classes péres (un Widget d'une classe donnée posséde les atributs de sa classe mais également les attributs des classes englobantes).

D'un point de vue physique, une classe de Widgets est un pointeur vers une structure. Les éléments de cette structure sont constants pour tous les Widgets d'une même classe.

Une instance de Widget est composée de deux parties :

- Une structure de données qui contient des éléments spécifiques au Widget instancié.
- Une struture de classe qui contient des informations applicables à tous les Widgets de la classe.

La plus grande partie des entrées/sorties d'un Widget (police de caractères, couleur, taille, bordure, etc...) est customizable par l'utilisateur.

2.2. La librairie des Widgets Athena - Xaw

Nous l'avons vu, un toolkit X comporte deux parties distinctes : La librairie Xt Intrinsincs et un ensemble de Widgets. L'ensemble des Widgets Athena est un exemple d'implémentation de Widgets à partir des Intrinsincs. C'est le toolkit livré en standard avec les bandes de distribution de X.

Parceque les Intrinsics fournissent les mêmes fonctionnalités de base à tous les ensemble de Widgets, il est possible par exemple d'utiliser des Widgets Athena avec d'autres Widgets issus d'une autre boîte à outil, Motif par exemple. Un tel mélange n'est cependant pas recommandé car chaque ensemble de Widgets posséde son propre protocole et il se peut que toutes les fonctionnalités des Widgets utilisés ne soient pas disponibles.

L'ensemble des Widgets Athena est une librairie située une couche au-dessus des Intrinsics (eux même une couche au-dessus de la Xlib). Ils fournissent au développeur d'applications graphiques un ensemble cohérent d'objets permettant de répondre à la plupars de ses besoins.

Bien que les Intrinsics soient un standard issu d'un Consortium, il n'y a pas d'ensemble de Widgets standard.

Un toolkit X se compose de :

- Un ensemble de fonctions Intrinsincs pour construire des widgets.
- Un modéle, une architecture pour assembler des widgets.
- Un ensemble de widgets pour la programmation d'interfaces graphiques.

Alors que la majorité des fonctions des Intrinsincs sont destinées au programmeur de widgets, une partie d'entre elles sont utiles au programmeur d'applications graphiques (voir le X Toolkit Intrinsics - C Langage Interface). Le modèle architectural permet au programmeur de widgets de désigner de nouveaux widgets à l'aide des fonctions Intrinsics et en combinant les widgets existants. Un Toolkit X est formé d'un ensemble de widgets et de méthodes d'assemblage.

2.3. Terminologie.

En plus des termes déjà définis pour la programmation sous Xlib (***display***, ***window***, ***events***, etc...), il est nécessaire d'introduire un nouvel ensemble de termes propres à la programmation à l'aide d'un Toolkit.

- `les widgets sont des fenêtres X': Chaque widget est associé à une fenêtre X. L'ID de cette fenêtre est accessible à l'aide d'une fonction des Intrinsincs. Ainsi on

peut utiliser des fonctions de la Xlib directement, pour gèrer les entrées sorties d'un widget donné.

- `Encapsulation d'informations': Les données relatives à chaque widget et à ses sous-classes sont privées. AInsi, ces données ne sont jamais directement accessibles au programmeur d'application, elles sont encapsulées dans le module logiciel qui les implante. La librairie Xt fournit cependant des fonctions et macros pour accèder à certains attributs des widgets.
- `Sémantique et géométrie des widgets': La sémantique des widget est clairement séparée de leur géométrie. Les widgets sont concernées par la sémantique de l'application et le rôle qu'ils ont à jouer pour interagir avec l'utilisaateur, mais ils ont peu de contrôle sur leur taille, leur placement relatif par rapport aux autres widgets, etc... Il existe des mécanismes pour associer plusieurs widgets et faire des suggestions quant à leur géométrie (geometric managers).

2.4. Input Focus.

The Intrinsics define a resource on all Shell widgets that interact with the window manager called input. This resource requests the assistance of window manager in acquiring the input focus. The resource defaults to False in the Intrinsics, but is redefined to default to True when an application is using the Athena widget set. An application programmer may override this default and set the resource back to False if the application does not need the window manager to give it the input focus. See the ***X Toolkit Intrinsics - C Language Interface*** for details on the input resource.

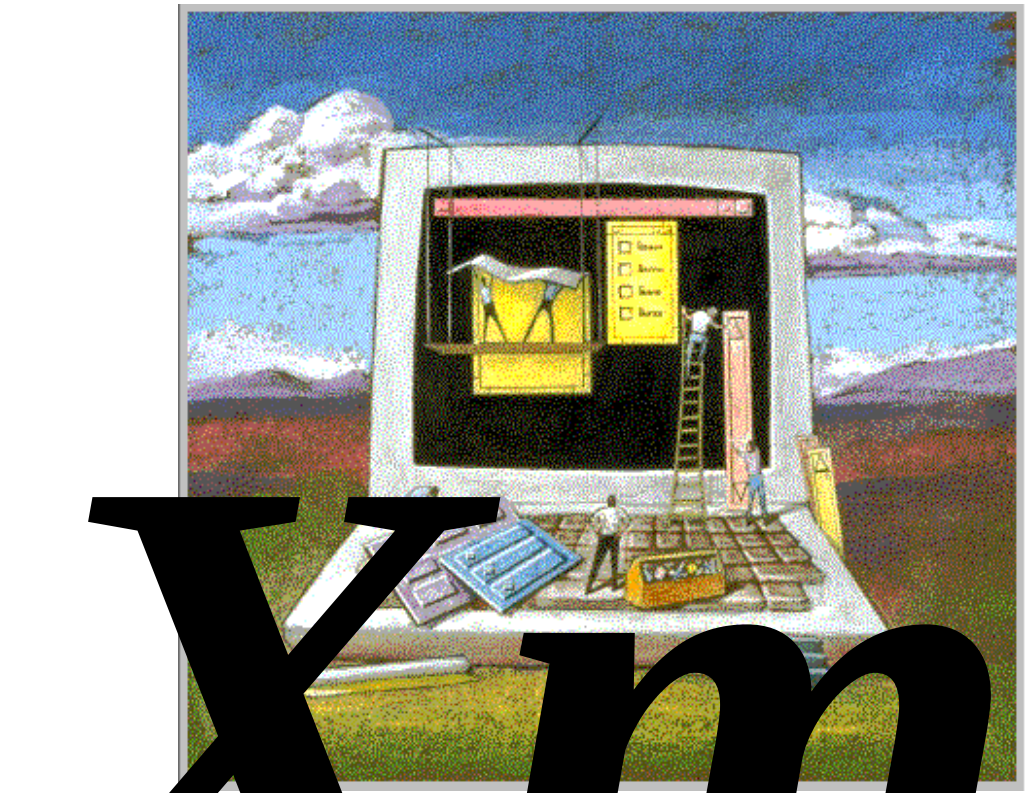
2.5. Using Widgets

Widgets serve as the primary tools for building a user interface or application environment. The Athena widget set consists of primitive widgets that contain no children (for example, a command button) and composite widgets which may contain one or more widget children (for example, a Box widget).

Motif

Toolkit Xm

Franck Diard, Michel Buffa, François Abram, 1998
diard@i3s.unice.fr, http://www.essi.fr/~diard
abram@i3s.unice.fr, http://www.essi.fr/~abram
V 1.4. Mars 1998.



Xm

1. Mise en œuvre de Motif.

1.1. Introduction.

Un Toolkit X fournit au programmeur de GUI (Graphic User Interface) une boîte à outil comprenant des éléments graphiques (Widgets) et un ensemble d'outils pour gérer leur comportement (fonctions, structures de données, etc...).

Il existe deux niveaux de Toolkits au-dessus de X :

- Le X Toolkit Intrinsincs présenté dans le chapitre précédent du cours, qui permet aux programmeurs de créer de nouveaux Widgets.
- Les third party Toolkits comme Athena ou MOTIF qui proposent un ensemble de Widgets au look and feel différents.

Si l'on utilise correctement les Widgets d'un Toolkit, cela simplifie considérablement la programmation d'une GUI. En outre, toutes les GUI développées avec un même Toolkit auront le même look and feel.

Lorsque nous écrirons des applications avec MOTIF nous aurons en sus des librairies MOTIF, à appeler certaines fonctions de la librairie Xt puisque MOTIF est construit sur cette dernière (il n'est cependant pas nécessaire de connaître en détail les mécanismes de la librairie Xt car MOTIF effectue la plus grande partie du travail d'interfaçage pour nous).

Remarque : MOTIF a été développé par Open Software Foundation (OSF) et le nom complet de MOTIF est en fait OSF/MOTIF.

1.2. Premiers pas avec MOTIF .

1.2.1. Notre premier programme.

Dans cette section, nous allons détailler un petit programme d'exemple, notre premier programme MOTIF.

Il ne va pas faire grand chose mais nous allons apprendre beaucoup en l'étudiant.

1.2.2. Que fait notre petit programme ?

Le programme push.c ouvre une fenêtre contenant un unique bouton poussoir. Ce bouton contient une chaîne de caractères : "Push_me". Lorsqu'on clique dessus (avec le bouton gauche de la souris), un message est affiché sur stdout (et non pas dans la fenêtre).

Pour sortir du programme, il faut tuer la fenêtre soit par ctrl-c dans le xterm d'oú on l'a lancée, soit à l'aide du window manager.

Nous verrons bientôt comment créer d'autres types de Widgets, des menus, etc...

1.2.3. Qu'allons-nous apprendre de ce petit programme ?

On apprend toujours beaucoup du premier programme que l'on écrit avec une nouvelle librairie. Ce petit programme est celui proposé dans le tutorial de la documentation MOTIF.

Nous allons voir (et comprendre) comment :

- Ecrire et compiler un programme MOTIF.
- S'effectuent les relations entre la Xlib, la librairie MOTIF et les Intrinsincs.
- Créer un Widget et configurer ses ressources.
- Gérer les événements.
- Appeler des fonctions à partir des événements.

1.2.4. Le programme push.c.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/*--------------------------*/
main(int argc, char **argv)
/*--------------------------*/
{

  Widget top_wid, button;
  XtAppContext  app;
  void pushed_fn();

  top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
                        &argc, argv, NULL, NULL);

  button = XmCreatePushButton(top_wid, "Push_me", NULL, 0);


  /* On dit à Xt de manager le bouton */
  XtManageChild(button);

  /* On attache un CallBack au bouton */
```

```
  XtAddCallback(button, XmNactivateCallback, activateCB, NULL);

  /* Affichage de la fenêtre principale (top_wid) et de tous ses enfants */
  XtRealizeWidget(top_wid);

  /* boucle de gestion des événements */
  XtAppMainLoop(app);
}

/*-----------------------------------------------*/
void activateCB(Widget w, XtPointer client_data,
           XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------*/
{
  printf("Don't Push Me!!\n");
}
```

1.2.5. L'appel de fonctions MOTIF, Xt et Xlib.

Lorsqu'on écrit une application MOTIF, on appelle explicitement des fonctions, on manipule des types de données appartenant aux librairies Xm, Xt et/ou à la Xlib.

En général, on utilise les fonctions Xlib pour dessiner ou effectuer une gestion plus fine des événements.

Toutes les fonctions et les types de données MOTIF commencent par ***Xm***. Par exemple, dans push.c, ***XmCreatePushButton***() est une fonction MOTIF.

Toutes les fonctions et types de données des Intrinsincs commencent par Xt. Par exemple, dans push.c, ***XtVaAppInitialize***() et ***XtManageChild***() sont des fonctions Xt.

Toutes les fonctions et les types de données de la Xlib commencent par X. Nous avons vu des exemples nombreux dans les premiers chapitres du cours.

Remarque : il y a quelques exceptions notoires comme Widget (Xt) ou Window (Xlib).

1.3. Compiler un programme MOTIF :

1.3.1. Fichiers à inclure :

Pour qu'une application MOTIF compile sans problèmes, il faut inclure les fichiers suivants : **<*Xm/Xm.h*>**. Obligatoire !

Un fichier par type de Widget utilisé. Chaque Widget possède son propre fichier include. Par exemple, dans push.c on a utilisé un bouton poussoir de type PushButton, il faut donc inclure <Xm/PushB.h>.

Remarque : ce n'est pas la peine d'inclure des fichiers pour la librairie Xt

car <***Xm/Xm.h***> les inclut déjà.

1.3.2. Edition de liens:

Il faut linker avec les librairies MOTIF, Xt et Xlib. On utilisera donc les options suivantes : ***-lXm -lXt -lX11*** pendant l'édition de liens.

Remarque : l'ordre d'inclusion de ces librairies est très important !

En général, les includes, librairies, manuels de MOTIF se trouvent dans les chemins standards de la librairie ***X11***.

1.4. Bases de la programmation MOTIF.

1.4.1. Initialisation de la toolkit :

L'initialisation de la toolkit Xt doit se faire avant toute chose !

Il existe plusieurs moyens de réaliser cette opération, celui proposé dans le petit programme push.c, à l'aide de la fonction ***XtVaAppInitialize***() est le plus commun. C'est celui que nous utiliserons dans la plupart de nos exemples.

L'appel de ***XtVaAppInitialize***() réalise les tâches suivantes :

- Ouverture d'une connexion avec le serveur X : ouverture du Display.
- Prise en compte des options X standards (-display, -geometry, etc...) sur la ligne de commande.

1.4.2. Lecture et prise en compte des ressources.

Une fenêtre top_level (container principal de l'application) est crée, et un pointeur retourné.

Description des paramètres de ***XtVaAppInitialize***() :

- Application context: Structure nécessaire à la librairie pour fonctionner correctement. On ne va pas s'en occuper car l'utilité de cette structure est réservée aux utilisateurs avancés. On se contentera toujours d'utiliser l'application context de la même manière que dans push.c tout au long des exemples que nous étudierons ensemble.
- Application class name: Une chaîne de caractère qui définit la classe de l'application. Par convention, le nom de la classe est le même que celui de l'application mais avec la première lettre en majuscule. Attention de ne pas vous tromper car le nom du fichier de ressources pour

l'aplication sera le même.

- Troisième et quatrième paramètres (Special X command line arguments) : Réservé aux utilisateurs avancés de MOTIF. L'utilisation de ces paramètres sort du cadre de ce cours, se réfèrer à la documentation Xt pour plus de détails. On se contentera de mettre le troisième paramètre à ***NULL*** et le quatrième à zéro.
- argc et argv: Contiennent les arguments de la ligne de commande. Standard...
- Septième et huitième paramètres (Fallback Ressources) : Sortent du cadre du cours. On mettra tout ça à ***NULL***!

1.4.3. Création d'un Widget

Avec MOTIF, il existe une fonction de création unique pour chaque type de Widget. Il est facile de deviner le nom de la fonction de création lorsqu'on connaît le type de Widget que l'on désire créer. La règle est la suivante :

Pour créer <widget name>, on utilise la fonction ***XmCreate***<widget name>.

Presque toutes les fonctions de création possèdent 4 paramètres :

- 1.Un pointeur vers le Widget père, top_wid dans le programme push.c.
- 2.Le nom du Widget, "Push_me" dans l'exemple. Ce nom servira à positionner les ressources du bouton dans le fichier de ressources.
- 3.Une liste d'attributs/ressources. NULL dans l'exemple, mais nous verrons plus tard l'utilité de ce paramètre.
- 4.Le nombre d'éléments dans la liste précédente.

Le troisième paramètre sert à initialiser les ressources des Widgets au moment de leur création (hauteur, largeur, couleur, etc…).

1.4.4. Manager un Widget :

Une fois créé, un Widget doit être pris en charge, il doit être managé. Cette prise en charge est réalisée par l'appel de la fonction ***XtManageChild***().

Lorsque un Widget est managé, tous les aspects liés à sa taille son pris en compte par son père.

Nous étudierons ce mécanisme plus en détail dans la suite du cours.

1.4.5. Gestion des événements, les Callbacks :

Principe de gestion des événements sous MOTIF :

Lorsqu'un Widget est créé il sait de lui-même comment répondre à certains événements : changer de taille lors d'une requête du Window Manager, changer son apparence lors d'un click souris (dans le cas d'un bouton poussoir), se redessiner si besoin est (gestion automatique des Expose events) etc…

Nous avons déjà vu dans la partie du cours consacrée à la Xlib les différents types d'événements et la complexité de leur prise en compte. Fort heureusement, Xt va grandement faciliter la tâche du programmeur car la plupart des Widgets possèdent des mécanismes simples et puissants pour prendre en compte les événements les plus courants.

Pour qu'un Widget utilise des fonctions spécifiées par le programmeur, il faut positionner certaines ressources du Widget que l'on appelle des ***Callback Resources***.

1.4.5.1. Les tables de translation :

Chaque Widget possède une table de translation qui définit la manière dont il va réagir à certains événements particuliers.

La liste complète des tables de translation de chaque Widget se trouve dans le manuel de référence Motif.

Voici un extrait de la table de translation du bouton poussoir :

```
BSelectPress: Arm()
BSelectClick: Activate(), Disarm()
```

BSelectPress correspond à un appui sur le bouton de gauche de la souris. L'action correspondante est un appel à la fonction interne Arm() qui modifie l'affichage du bouton pour simuler son enfoncement.

BSelectClick correspond à un click souris, c'est-à-dire lorsque le bouton de gauche est enfoncé puis relâché, alors l'action sera Activate() suivi de Disarm(), cette dernière redessine le bouton comme s'il avait repris son apparence "non enfoncé".

Les événements clavier peuvent également figurer dans la table de translation. Le programmeur peut spécifier des raccourcis clavier, prendre en compte l'appui de certaines touches aux fonctions particulières…

Par exemple : ***KActivate*** - typiquement la touche return (lorsque on entre un texte dans un sélécteur, <return> sera équivalent à presser le bouton Ok par exemple), KHelp, etc...

1.4.5.2. Spécifier des callbacks.

Les fonctions ***Arm***(), ***Activate***() ou ***Disarm***() sont des exemples de callbacks prédéfinis. Si l'on veut exécuter une fonction particulière lorsque l'utilisateur appuie sur un bouton, il faut attacher des callbacks aux Widgets.

Dans ***push.c*** la fonction ***activateCB***() est un callback. On l'attache au bouton poussoir à l'aide de la fonction XtAddCallBack() qui est la fonction la plus commune pour effectuer cette opération.

Elle possède 4 paramètres :

- 1.Un pointeur sur le Widget (bouton avec l'exemple de ***push.c***).
- 2.Le nom de la Callback Ressouce du Widget, dans notre exemple, on veut appeller ***activateCB***() chaque fois que le bouton est activé, c'est-à-dire chaque fois qu'on clique dessus (appui et relâche du bouton souris), on positionne donc la ressource ***XmNactivateCallback***.
- 3.Un pointeur sur la fonction utilisateur que l'on désire appeler.
- 4.Des paramètres d'appel que l'on appelle Client Data. Nous verrons des exemples d'utilisation de ce paramètre dans la suite du cours, mais pour le moment ne compliquons pas et mettons le à ***NULL***.

Dans cet exemple, lors d'un click souris sur le bouton, MOTIF va bien ajouter à la liste des actions l'action précisée par le programmeur, qui consiste à appeler la fonction ***activateCB***(), mais les mécanismes internes du Toolkit vont quand même exécuter les actions prédéfinies ***Activate***() et ***Disarm***(). Il s'agit d'un mécanisme d'ajout. Le programmeur peut très bien spécifier plus d'un callback pour un même type d'événement.

Par exemple, dans ***push.c***, si l'on voulait également appeler la fonction utilisateur quit(), on ajouterait la ligne suivante :

```
XtAddCallback(button, XmNdisarmCallback, quit, NULL);
```

1.4.5.3. Déclaration des fonctions de callback :

Regardons maintenant dans le programme ***push.c*** comment la fonction de callback activateCB() a été déclarée :

```
void activateCB(Widget w, XtPointer client_data,
                XmPushButtonCallbackStruct *cbs)
```

Les fonctions de callback possèdent trois paramètres :

- Un pointeur sur le(s) Widget(s) associé(s) à ce callback. Plusieurs Widgets peuvent appeler une même fonction de callback.
- Le second paramètre sert à passer des client data à la fonction de callback. Nous verrons plus tard dans le cours comment utiliser ce paramètre. Pour le moment laissons-le comme ça.
- Un pointeur sur une structure qui contient tous les renseignements relatifs à l'événement qui a provoqué l'appel de la fonction de callback, ainsi que des renseignements relatifs au type de Widget passé en 1er paramètre.

Par exemple, dans push.c, cette structure devra être castée par le type ***XmPushButtonCallbackStruct*** puisque le widget qui a provoqué l'appel est un bouton poussoir.

D'une maniére générale, la Callback Structure du widget **<*widget name*>** a la forme suivante :

```
typdef struct {
   int reason;
   XEvent *event;
   ... champs spécifiques au Widget
} Xm<widget name>CallbackStruct;
```

Le champ ***reason*** contient des informations sur le callback telles que "***arm***" ou "***disarm***" qui ont provoqué l'appel de la fonction de callback".

1.4.5.4. Affichage des Widgets et boucle de gestion des événements :

Nous avons presque terminé l'étude de notre premier programme. Il ne reste plus que deux étapes avant la fin, deux étapes essentielles que tout programnme MOTIF doit effectuer en dernier :

- Afficher, réaliser (realize) les Widgets. Ceci est fait en appelant la fonction des Intrinsincs XtRealizeWidget(). Nous passons comme paramètre de cette fonction un pointeur vers la fenêtre racine de notre application, le top level widget ***top_wid***. Ainsi, lorsque ce Widget sera affiché, tous ses enfants le seront aussi.
- Effectuer la gestion des événements. L'appel de XtAppMainLoop(app) le fait pour nous. Après cet appel, Xt prend le contrôle du programme, gère les événements,

appelle les fonctions de callback que nous avons définies,
etc...

2. Généralités sur les Widgets.

2.1. Introduction.

MOTIF possède deux grandes classes de Widgets: primitive et manager. Plusieurs choses à savoir :

- Ces deux grandes classes sont découpées en nombreuses sous-classes.
- Un Widget de la classe primitive est destiné a fonctionner de manière atomique : il ne contiendra pas d'autre Widget. L'exemple typique de Widget appartenant à cette classe est le bouton poussoir.
- Un Widget de la classe manager est destiné à servir de container et pourra contenir d'autres Widgets.

2.2. Les différents types de Widgets :

2.2.1. Les widgets de la classe primitive.

2.2.1.1. ArrowButton .

Une bouton contenant non pas un label mais un pixmap en forme de flêche. Une ressource permet de choisir le sens de la flêche.

Exemple d'ArrowButtons :

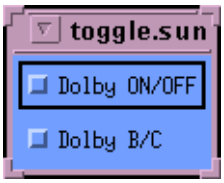
2.2.1.2. Label.

Cette classe posséde quatre sous-classes :

- 1.PushButton : un bouton poussoir contenant un texte. Voici un exemple de PushButtons :
-
- 2.DrawnButton : un bouton poussoir contenant un pixmap. Voici un exemple de DrawnButtons :

- 3.CascadeButton : Un bouton poussoir particulier utilisé pour faire des menus déroulants. Voici un exemple de CascadeButtons :

- 4.ToggleButton : Un bouton On/Off. Voici un exemple de ToggleButtons :

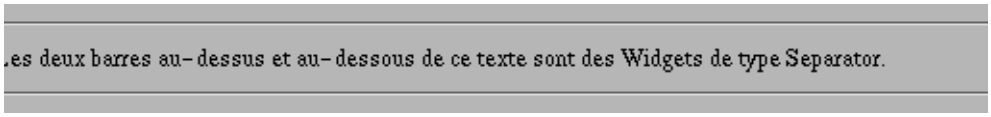


2.2.1.3. Scrollbar.

Permet d'afficher le contenu d'une fenêtre partiellement. A l'aide d'une Scrollbar l'utilisateur peut séléctionner la partie de la fenêtre qui est visible à l'écran. Voici un exemple de Scrollbar :

2.2.1.4. Separator.

Eléments graphiques permettant d'afficher des lignes verticales ou horizontales pour séparer différentes parties d'un menu par exemple. Voici un exemple de Separator :



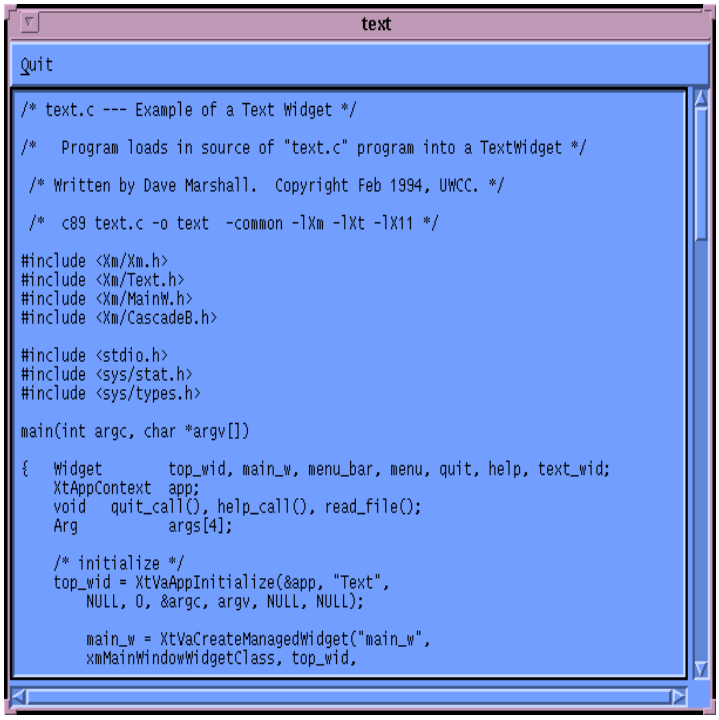
2.2.1.5. List.

Une boîte contenant une liste d'éléments que l'utilisateur peut choisir à la souris. Voici un exemple de List :

2.2.1.6. Text.

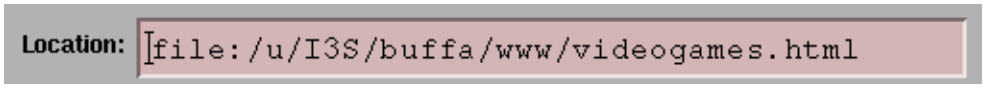
Ce Widget est un petit éditeur de texte à lui tout seul. Voici un exemple

de Text :



2.2.1.7. TextField.

Un éditeur de texte d'une seule ligne. Voici un exemple de TextField :



2.2.1.8. Les Gadgets.

Parfois, en lisant les manuels MOTIF, vous allez entendre parler de Gadgets. Les Gadgets sont des "Widgets simplifiés". Ils n'en existe que de trois types : ArrowButton, Label et Separator.

Ils ne disposent pas d'autant de ressources que les Widgets correspondants et leur gestion par les librairies ***Xm*** et ***Xt*** est plus rapide bien que leur comportement soit équivalent.

Nous ne nous attarderons pas sur les Gadgets car avec la puissance des machines actuelles et la complexité relativement faible des applications que nous allons développer dans ce cours, la différence entre Gadgets et Widgets est imperceptible.

2.2.2. Les Widgets de la classe Manager.

La classe Manager est divisée en de nombreuses sous-classes.

2.2.2.1. Frame.

Ce type de Widget permet d'encadrer d'autres Widgets en spécifiant la taille du cadre, son style, etc… Voici un exemple de Frame :

2.2.2.2. Scrolled Window.

Une fenêtre possédant des Scrollbars. On peut mettre dans une Scrolled Window d'autres Widgets dont la taille excède celle de la ScrolledWindow.

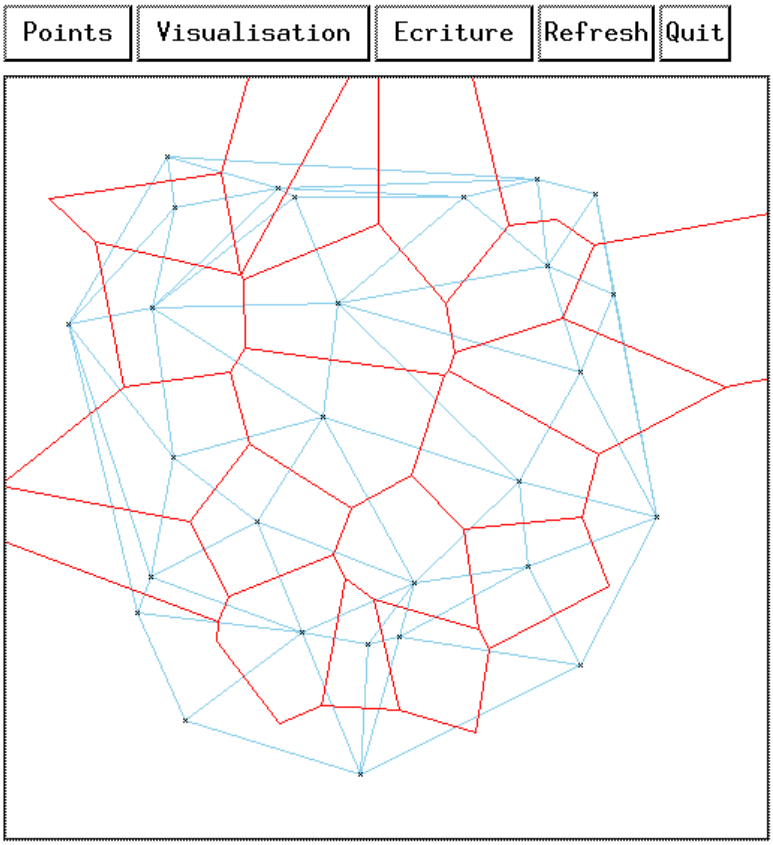
2.2.2.3. MainWindow

Un container de type TopLevel spécialement désigné pour contenir une application simple. On peut mettre dans une MainWindow plusieurs types de Widgets. Voici un exemple de MainWindow :

2.2.2.4. DrawingArea.

On peut dessiner dans ce Widget. Remarque : MOTIF ne fournit aucune fonction de dessin. Il faudra faire appel à des fonctions de la Xlib. Voici un

exemple de DrawingArea :



2.2.2.5. PanedWindow.

Un container pour ranger verticalement plusieurs Widgets.

2.2.2.6. RowColumn.

Widget gèrant automatiquement le positionnement de plusieurs Widgets en lignes et colonnes.

2.2.2.7. Scale.

Un ascenseur horizontal ou vertical permettant à l'utilisateur de spécifier à la souris une valeur appartenant à un intervalle borné linéaire.

2.2.2.8. BulletinBoard.

Cette classe de Widgets possède deux sous-classes :

- 1.Form : même utilité que RowColumn: permet de

contrôler le positionnement des Widgets créés à l'intérieur de ce Widget. Ce contrôle est plus puissant mais plus complexe qu'avec les RowColumns.

- 2.Dialog: Il existe deux classes de Dialog Widgets :
  - 1.MessageBox qui permet d'afficher un message dans une "boîte" (une fenêtre de type TopLevel) pour informer l'utilisateur.
  - 2.SelectionBox qui permet l'interaction avec l'utilisateur. MOTIF fournit un Widget de type Command pour la saisie de commandes utilisateur et un Widget de type ***FileSelectionBox*** pour la séléction des fichiers.

3. Spécification des ressources.

3.1. Le fichier app-defaults.

Le mécanisme de gestion des ressources d'une application permet de personnaliser les widgets sans avoir a recompiler. Les ressources peuvent être spécifiées à plusieurs endroits: sur la ligne de commande, dans le fichier ***$HOME/.Xdefaults*** de l'utilisateur, dans un fichier utilisateur dédié à l'application (par exemple dans le répertoire ***$HOME/.app-defaults***), ou encore dans un fichier de ressources par defaut (dans le répertoire ***/usr/lib/app-defaults***).

Chaque ressource d'un widget possède une valeur par défaut déterminée par sa classe. Il est cependant possible de modifier cette valeur par défaut et/ou rendre la ressource configurable par l'utilisateur.

Prenons l'exemple du bouton poussoir du cours précédent (le programme ***push.c***).

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/*-------------------------*/
main(int argc, char **argv)
/*-------------------------*/
{

  Widget top_wid, button;
  XtAppContext  app;
  void pushed_fn();

  top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
                    &argc, argv, NULL, NULL);

  button = XmCreatePushButton(top_wid, "Push_me", NULL, 0);


  /* On dit à Xt de manager le bouton */
  XtManageChild(button);

  /* On attache un CallBack au bouton */
  XtAddCallback(button, XmNactivateCallback, activateCB, NULL);

  /* Affichage de la fenêtre principale (top_wid) et de tous ses enfants */
  XtRealizeWidget(top_wid);

  /* boucle de gestion des événements */
  XtAppMainLoop(app);
}

/*-----------------------------------------------*/
```

```
void activateCB(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)
/*-------------------------------------------------*/
{
  printf("Don't Push Me!!\n");
}
```

Dans ce programme, la chaîne de caractère affichée dans le bouton est égale à "Push_me", car par défaut, dans le cas d'un bouton poussoir, le label est égal au nom du widget. ATTENTION: ceci n'est vrai que dans le cas des widgets appartenant à la classe Label, et n'est pas vérifié en ce qui concerne les autres ressources de notre bouton poussoir.

Xt propose un système très pratique pour spécifier des valeurs par défaut aux ressources utilisées par les widgets d'une application: les fichiers de ressources par défaut, que l'on trouve dans le répertoire ***/usr/lib/X11/app-defaults*** (sous Unix).

Le fichier de ressources par défaut d'une application devra être égal au nom de la classe de l'application, deuxième paramètre de la fonction ***XtVaAppInitialize***(). Dans notre exemple, le fichier devra s'appeler ***Push***.

Par convention, la classe d'une application sera égale au nom de l'application, mais avec la première lettre en majuscule. Si l'application commence par un X, on capitalisera les deux premières lettres (voilà pourquoi le fichier de ressources de l'application xterm s'appelle XTerm).

4. Le Widget de type RowColumn.

4.1. Présentation.

Il s'agit du plus simple des "container widgets" car il permet aisément de gèrer le positionnement de ses widgets fils.

Dans un RowColumn, les widgets sont positionnés de la manière suivante :

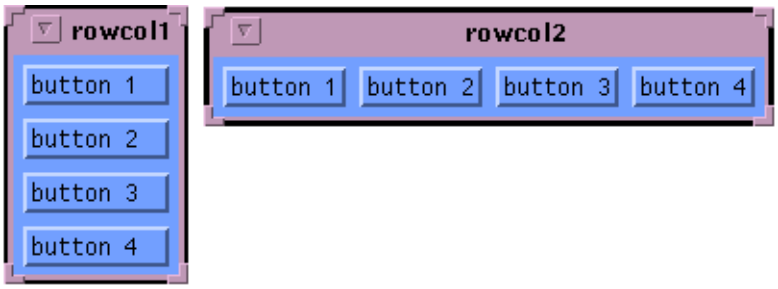
- Les widgets fils sont alignés horizontalement ou verticalement dans l'ordre de création. La ressource ***XmNorientation*** spécifie le sens de l'alignement, et possède ***XmVERTICAL*** comme valeur par défaut, ***XmHORIZONTAL*** est l'autre valeur possible.
- Les widgets peuvent être alignés sur plusieurs lignes ou colones selon la valeur de XmNorientation. La ressource XmNnumColumns spécifie le nombre de lignes/colonnes (uniquement si la ressource XmNpacking vaut XmPACK_COLUMNS).
- Les widgets fils doivent avoir la même taille. Si ce n'est pas le cas ou bien si la taille de ces widgets n'est pas spécifiée par leurs propres ressources, alors on peut demander au RowColumn de "forcer" leur taille. Ceci peut être fait de plusieurs manières selon la valeur de la ressource XmNpacking:
  - XmNpacking vaut XmPACK_TIGHT (valeur par défaut) : les widgets auront toutes la même largeur égale à la largeur du plus large des fils. Si la taille de la RowColumn change, la taille des fils changera en conséquence.
  - XmNpacking vaut XmPACK_COLUMNS : tous les widgets fils auront la même taille et seront alignés sur le nombre de lignes/colonnes spécifié par la ressource XmNnumColumns.
  - XmNpacking vaut XmPACK_NONE : le RowColumn ne va pas essayer de changer la taille ou la position des

widgets fils.

Examinons maintenant quelques exemples...

4.2. Les programmes rowcol1.c et rowcol2.c

Ces deux programmes créent une petite boîte qui contient 4 boutons poussoirs (alignés verticalement dans rowcol1.c et horizontalement dans rowcol2.c). Le résultat est le suivant :



4.2.1. rowcol1.c :

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>

main(argc, argv)
int argc;
char *argv[];
{
   Widget top_widget, row_column, button1, button2, button3, button4;
   XtAppContext app;
   int n;
   Arg args[10];

   top_widget = XtVaAppInitialize(&app, “rowcol1”, NULL, 0,
                        &argc, argv, NULL, NULL);

   n=0;
   row_column = XmCreateRowColumn (top_widget, “row_column”, args, n);
   XtManageChild(row_column);

   n=0;
   button1 = XmCreatePushButton(row_column, “button1”, args, n);
   XtManageChild(button1);

   n=0;
   button2 = XmCreatePushButton(row_column, “button2”, args, n);
   XtManageChild(button2);

   n=0;
   button3 = XmCreatePushButton(row_column, “button3”, args, n);
   XtManageChild(button3);

   n=0;
   button4 = XmCreatePushButton(row_column, “button4”, args, n);
   XtManageChild(button4);

   XtRealizeWidget(top_widget);
   XtAppMainLoop(app);
```

```
}
```

4.2.2. rowcol2.c :

Remplacer juste les lignes de création du RowColumn par :

```
n=0;
XtSetArg(args[n], XmNorientation, XmHORIZONTAL); n++;
row_column = XmCreateRowColumn (top_widget, “row_column”,
                        args, n);
XtManageChild(row_column);
```

Seule la valeur de la ressource ***XmNorientation*** a changé. Au lieu de valoir ***XmVERTICAL*** qui est la valeur par défaut, comme dans rowcol1.c, elle vaut ici XmHORIZONTAL. Le reste du code est le même.

Remarque : pour compiler sans erreur, nous avons inclu le fichier **<*Xm/RowColumn.h*>** .

Il existe bien d’autres ressources pour les widgets de type RowColumn, mais nous avons vu les trois plus importantes : ***XmNorientation***, ***XmNpacking*** et ***XmNnumColumns***.

5. *Le widget de type Form*

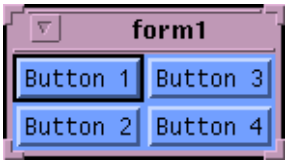
5.1. Introduction .

Après le RowColumn relativement simple à utiliser, le widget de type Form est l'autre widget "container" proposé par Motif.

Par rapport au RowColumn, le widget de type Form permet un contrôle plus précis et plus puissant du positionnement géométrique des widgets fils, au prix d'une certaine complexité d'utilisation. Avec un Form, les widgets fils peuvent avoir des tailles différentes et on peut leur donner des attachements relatifs les uns par rapport aux autres.

Le positionnement des widgets fils dans un Form Widget peut être réalisé de plusieurs manières.

Examinons tout d'abord trois petits programmes ***form1.c***, ***form2.c*** et ***form3.c*** qui utilisent trois approches différentes pour produire le même résultat :



5.2. Attachement simple des widgets fils.

Remarque : lorsqu'un widget est créé dans une Form, il hérite de nouveaux attributs qu'il ne possède pas naturellement.

Ces attributs concernent son positionnement à l'intérieur de la Form. Parmi ceux-ci, certains attributs servent à attacher le widget à ses widgets voisins dans la Form, ou bien directement aux bords de la Form.

Les ressources concernant l'attachement sont les suivantes :

- ***XmNtopAttachment*** spécifie à quoi est attaché le widget par le haut.
- ***XmNbottomAttachment*** spécifie à quoi est attaché le widget par le bas.

- ***XmNleftAttachment*** spécifie à quoi est attaché le widget par la gauche.
- ***XmNrightAttachment*** spécifie à quoi est attaché le widget par la droite.

Les valeurs possibles sont :

- ***XmATTACH_FORM*** pour attacher le widget à un bord interieur de la Form.
- ***XmATTACH_WIDGET*** pour attacher le widget à un des autres widgets qui se trouve dans la Form. Il faudra alors positionner une autre resource pour spécifier le widget auquel on s'attache :
  - ***XmNtopWidget*** pour un attachement par le haut.
  - ***XmNbottomWidget*** pour un attachement par le bas.
  - ***XmNleftWidget*** pour un attachement par la gauche.
  - ***XmNrightWidget*** pour un attachement par la droite.

Dans l'exemple suivant les widgets fils sont placés dans une Form en précisant pour chaque widget son attachement soit à un ou plusieurs côtés de la Form, soit à d'autres widgets.

5.3. Le programme form1.c :

On créé quatre boutons dans une Form :

- bouton1 est attaché par le haut et la gauche à la form.
- bouton2 est attaché par le bas et la gauche à la form, par le haut à bouton1.
- bouton3 est attaché par le haut et la droite à la form, par la gauche à bouton1.
- bouton4 est attaché par le bas et la droite à la form, par le haut à bouton3 et par la gauche à bouton2.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>

main (int argc, char **argv)
{
  XtAppContext app;
```

```
 Widget top_widget, form, button1, button2, button3, button4;
 int n=0;
 Arg args[10];

 top_widget = XtVaAppInitialize(&app, “form1”, NULL, 0,
                        &argc, argv, NULL, NULL);

 n=0;
 form = XmCreateForm(top_widget, “form”, args, n);
 XtManageChild(form);

 n=0;
 /* bouton 1 : bords haut et gauche attachés à la Form */
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
 button1 = XmCreatePushButton(form, “button1”, args, n);
 XtManageChild(button1);

 n=0;
 /* bouton 2: bords haut  attaché au bouton 1, bas et gauche
    à la Form */
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_WIDGET); n++;
 XtSetArg(args[n], XmNtopWidget, button1); n++;
 XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
 button2 = XmCreatePushButton(form, “button2”, args, n);
 XtManageChild(button2);

 n=0;
 /* bouton 3: bords gauche  attaché au bouton 1, haut et
    droite à la Form */
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
 XtSetArg(args[n], XmNleftWidget, button1); n++;
 button3 = XmCreatePushButton(form, “button3”, args, n);
 XtManageChild(button3);

 n=0;
 /* bouton 4: bords haut attaché au bouton 3, gauche au bouton 2,
    bas et droite à la Form */
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_WIDGET); n++;
 XtSetArg(args[n], XmNtopWidget, button3); n++;
 XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
 XtSetArg(args[n], XmNleftWidget, button2); n++;
 XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
 button4 = XmCreatePushButton(form, “button4”, args, n);
 XtManageChild(button4);

 XtRealizeWidget(top_widget);
 XtAppMainLoop(app);
}
```

Remarque : pour compiler sans erreur il faut inclure le fichier <Xm/Form.h>.

5.4. Attachement à des positions prédéfinies dans la Form :

Il est possible de positionner un widget à un endroit particulier dans une Form sans l’attacher à un autre widget. En effet, MOTIF suppose qu’une Form est découpée en segments verticaux et horizontaux formant une grille régulière. La position d’un widget pourra ainsi être spécifiée en indiquant un nombre de segments à partir du coin en haut à gauche de la Form.

Par défaut une Form possède 100 divisions horizontales et verticales. On peut changer cette valeur à l’aide de la ressource ***XmNfractionBase***.

La position d'un côté particulier d'un widget dans une Form sera précisée à l'aide des ressources d'attachement vues précédemment ***XmNtopAttachment***, ***XmNbottomAttachment***, etc..., simplement on leur donnera une nouvelle valeur : ***XmATTACH_POSITION***.

Il faudra ensuite positionner la ressource correspondante ***XmNtopPosition***, ***XmNbottomPosition***, etc… à la valeur désirée (un entier).

5.5. Le programme form2.c :

Dans cet exemple nous allons indiquer la position des différents boutons de la manière suivante :

- bouton1 sera attaché par le haut et la gauche à la Form, et ses bords droits et bas s'arrêteront à la position 50 (comme il y a 100 positions, ce bouton occupera exactement le quart haut et gauche de la Form).
- bouton2 sera attaché par le bas et la gauche à la Form, et ses bords haut et droit s'arrêteront à la position 50.
- bouton3 sera attaché par le haut et la droite à la Form, et ses bords bas et gauche s'arrêteront à la position 50.
- bouton4 sera attaché par le bas et la droite à la Form, et ses bords haut et gauche s'arrêteront à la position 50.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>


main (int argc, char **argv)
{
  XtAppContext app;
  Widget top_widget, form, button1, button2, button3, button4;
  int n=0;
  Arg args[10];

  top_widget = XtVaAppInitialize(&app, “form1”, NULL, 0,
                        &argc, argv, NULL, NULL);

  n=0;
  form = XmCreateForm(top_widget, “form”, args, n);
  XtManageChild(form);

  n=0;
  /* bouton 1 : bords haut et gauche attachés à la Form,
     bords droit et bas a la position 50/100 */
  XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNrightPosition, 50); n++;
  XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNbottomPosition, 50); n++;
  button1 = XmCreatePushButton(form, “button1”, args, n);
  XtManageChild(button1);
```

```
 n=0;
 /* bouton 2: bords bas et gauche attachés à  la Form,
    droit et haut à la position 50/100  */
 XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNrightPosition, 50); n++;
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNtopPosition, 50); n++;
 button2 = XmCreatePushButton(form, "button2", args, n);
 XtManageChild(button2);

 n=0;
 /* bouton 3: bords haut et droit attachés à la Form,
    bas et gauche à la position 50/100 */

 XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNleftPosition, 50); n++;
 XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNbottomPosition, 50); n++;
 button3 = XmCreatePushButton(form, "button3", args, n);
 XtManageChild(button3);

 n=0;
 /* bouton 4: bas et droit attachés à la Form,
    haut et gauche à la position 50/100 */
 XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
 XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNleftPosition, 50); n++;
 XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
 XtSetArg(args[n], XmNtopPosition, 50); n++;
 button4 = XmCreatePushButton(form, "button4", args, n);
 XtManageChild(button4);

 XtRealizeWidget(top_widget);
 XtAppMainLoop(app);
}
```

Remarque : essayez de modifier à l'aide du window manager la taille de la fenêtre correspondant aux programmes form1 et form2 obtenus à partir des sources présentées ici.

5.6. Attachements opposés :

L'attachement opposé est un autre moyen d'attacher les bords d'un widget.

On réalise un tel attachement en positionnant comme auparavant les ressources ***XmntopAttachment***, ***XmNbottomAttachment***, etc... mais cette fois-ci on leur donne la valeur ***XmATTACH_OPPOSITE_WIDGET***. On précisera le widget auquel on s'attache à l'aide de la ressource ***XmNwidget*** déjà étudiée.

Remarque : avec ce type d'attachement "opposé", on attache les côtés "similaires". Au lieu de dire "Je vais attacher le bord droit de mon widget au bord gauche du widget voisin", on va dire "J'attache le bord droit de mon widget au bord droit d'un autre widget de manière à ce qu'ils soient alignés.

5.7. Le programme form3.c :

Cet exemple produit exactement le même résultat que le programme form2.c. Seules les ressources des widgets button2 et button4 ont été modifiées. Pour button2 on a indiqué que l'on voulait son bord droit aligné avec celui de button1. Pour button4, on a indiqué que l'on voulait son bord gauche aligné avec celui de button3.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>


main (int argc, char **argv)
{
  XtAppContext app;
  Widget top_widget, form, button1, button2, button3, button4;
  int n=0;
  Arg args[10];

  top_widget = XtVaAppInitialize(&app, "form1", NULL, 0,
                          &argc, argv, NULL, NULL);

  n=0;
  form = XmCreateForm(top_widget, "form", args, n);
  XtManageChild(form);

  n=0;
  /* bouton 1 : bords haut et gauche attachés à la Form,
     bords droit et bas a la position 50/100 */
  XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNrightPosition, 50); n++;
  XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNbottomPosition, 50); n++;
  button1 = XmCreatePushButton(form, "button1", args, n);
  XtManageChild(button1);

  n=0;
  /* bouton 2: bords bas et gauche attachés à  la Form,
     droit et haut à la position 50/100  */
  XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNrightAttachment, XmATTACH_OPPOSITE_WIDGET); n++;
  XtSetArg(args[n], XmNrightWidget, button1); n++;
  XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNtopPosition, 50); n++;
  button2 = XmCreatePushButton(form, "button2", args, n);
  XtManageChild(button2);

  n=0;
  /* bouton 3: bords haut et droit attachés à la Form,
     bas et gauche à la position 50/100 */

  XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNleftPosition, 50); n++;
  XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNbottomPosition, 50); n++;
  button3 = XmCreatePushButton(form, "button3", args, n);
  XtManageChild(button3);

  n=0;
  /* bouton 4: bas et droit attachés à la Form,
     haut et gauche à la position 50/100 */
```

```
  XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
  XtSetArg(args[n], XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET); n++;
  XtSetArg(args[n], XmNleftWidget, button3); n++;
  XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
  XtSetArg(args[n], XmNtopPosition, 50); n++;
  button4 = XmCreatePushButton(form, "button4", args, n);
  XtManageChild(button4);

  XtRealizeWidget(top_widget);
  XtAppMainLoop(app);
}
```

5.8. Un exemple plus complet :

Nous terminerons l'étude des widgets de type Form avec cet exemple. Nous allons créer une Form contenant deux types de widgets différents. Nous allons également utiliser des fonctions de callbacks.

Ce petit programme s'appelle arrows.c. Il crée quatre widgets de type ArrowButton autour d'un PushButton labellé "Quit" :

Lorsqu'on clique sur les boutons ils affichent simplement un message sur stdout.

Ce petit programme utilise une astuce : la ressource ***XmNfractionBase*** de la Form est positionnée à 3. Ceci crée une grille de 3 x 3 cases qui est plus facile à manipuler dans ce cas précis.

Le programme arrows.c :

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/ArrowB.h>
#include <Xm/Form.h>

void north(), south(), east(), west(), quitb();

/*--------------------------------------------*/
main (int argc, char **argv)
/*--------------------------------------------*/
{
  XtAppContext app;
  Widget top_widget, form, arrow1, arrow2, arrow3, arrow4, quit;
  int n=0;
  Arg args[10];

  top_widget = XtVaAppInitialize(&app, "form1", NULL, 0,
                        &argc, argv, NULL, NULL);

  n=0;
  XtSetArg(args[n], XmNfractionBase, 3); n++;
```

```
form = XmCreateForm(top_widget, “form”, args, n);
XtManageChild(form);

n=0;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNtop, 0); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNbottomPosition, 1); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNleftPosition, 1); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNrightPosition, 2); n++;
XtSetArg(args[n], XmNarrowDirection,   XmARROW_UP); n++;
arrow1 = XmCreateArrowButton(form, “arrow1”, args, n);
XtAddCallback(arrow1, XmNactivateCallback, north, NULL);
XtManageChild(arrow1);


n=0;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNtopPosition, 1); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNbottomPosition, 2); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNleftPosition, 0); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNrightPosition, 1); n++;
XtSetArg(args[n], XmNarrowDirection,   XmARROW_LEFT); n++;
arrow2 = XmCreateArrowButton(form, “arrow2”, args, n);
XtAddCallback(arrow2, XmNactivateCallback, west, NULL);
XtManageChild(arrow2);

n=0;
/* bouton 3: bords haut et droit attachés à la Form,
   bas et gauche à la position 50/100 */

XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNtopPosition, 1); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNbottomPosition, 2); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNleftPosition, 2); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNrightPosition, 3); n++;
XtSetArg(args[n], XmNarrowDirection,   XmARROW_RIGHT); n++;
arrow3 = XmCreateArrowButton(form, “arrow3”, args, n);
XtAddCallback(arrow3, XmNactivateCallback, east, NULL);
XtManageChild(arrow3);

n=0;
/* bouton 4: bas et droit attachés à la Form,
   haut et gauche à la position 50/100 */
XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNtopPosition, 2); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNbottomPosition, 3); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNleftPosition, 1); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNrightPosition, 2); n++;
XtSetArg(args[n], XmNarrowDirection,   XmARROW_DOWN); n++;
arrow4 = XmCreateArrowButton(form, “arrow4”, args, n);
XtAddCallback(arrow4, XmNactivateCallback, south, NULL);
XtManageChild(arrow4);

n=0;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNtopPosition, 1); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNbottomPosition, 2); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNleftPosition, 1); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg(args[n], XmNrightPosition, 2); n++;
```

```
  quit = XmCreatePushButton(form, “quit”, args, n);
  XtAddCallback(quit, XmNactivateCallback, quitb, NULL);
  XtManageChild(quit);

  XtRealizeWidget(top_widget);
  XtAppMainLoop(app);
}

/* CALLBACKS */
/*---------------------------------------------*/
void north(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)
/*---------------------------------------------*/
{
  printf(“Going North\n”);
}

/*---------------------------------------------*/
void west(Widget w, XtPointer client_data,
         XmPushButtonCallbackStruct *cbs)
/*---------------------------------------------*/
{
  printf(“Going West\n”);
}

/*---------------------------------------------*/
void east(Widget w, XtPointer client_data,
         XmPushButtonCallbackStruct *cbs)
/*---------------------------------------------*/
{
  printf(“Going East\n”);
}

/*---------------------------------------------*/
void south(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

/*---------------------------------------------*/
{
  printf(“Going South\n”);
}

/*---------------------------------------------*/
void quitb(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)
/*---------------------------------------------*/

{
  printf(“quit button pressed\n”);
  exit(0);
}
```

6. Le Widget de type MainWindow, les Menus

6.1. Introduction .

Dans cette section nous allons étudier un autre type de “container”, le widget de type MainWindow. Nous étudierons particulièrement les relations entre les widgets de ce type et les différents menus. Nous verrons qu’il est également possible de mettre d’autre types de widgets dans une MainWindow.

6.2. Le widget de type MainWindow.

Les MainWindow sont utilisés comme top level “container” pour des applications simples.

Une MainWindow consiste en général en une MenuBar (barre de menu) située en haut de la fenêtre de l’application, et une work area en dessous pouvant contenir n’importe quel type de widgets. Voici un exemple d’application bâtie sur une MainWindow :

On peut mettre dans une barre de menu :

- Des boutons.
- Des menus déroulants ou Pulldown menus, qui peuvent à leur tour contenir des sous-menus déroulants.

En plus de la work area, le widget de type MainWindow gère une zone appelée command area dans laquelle l’utilisateur peut entrer des commandes (du texte sur une ligne), et une message area dans laquelle l’application peut

écrire.

Pour le moment, étudions les menus!

6.3. Le widget MenuBar.

La création d'un vértitable menu comme on en retrouve dans la plupart des applications est une tâche relativement complexe. C'est pourquoi nous allons découper l'étude des menus en deux étapes.

Dans un premier temps nous allons voir comment créer une barre de menu simple contenant seulement des boutons.

Dans un second temps nous verrons comment ajouter à cette barre de menu des menus déroulants.

Les boutons que va contenir une barre de menu ou un menu déroulants sont d'un type particulier : le type ***CascadeButton***. Vous comprendrez bien vite pourquoi on les a nommé de la sorte.

Etudions maintenant un petit exemple.

6.4. Le programme menu_cascade.c :

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/RowColumn.h>

void quit_call(), help_call(); /* callbacks */

/*--------------------------*/
main(int argc, char *argv[])
/*--------------------------*/
{
  Widget top_wid, main_win, menu_bar, quit, help;
  XtAppContext app;
  int n=0;
  Arg args[10];

  /* create application, main and menubar widgets */

  top_wid = XtVaAppInitialize(&app, "menu_cascade",
                        NULL, 0, &argc, argv, NULL, NULL);

  n=0;
  main_win = XmCreateMainWindow(top_wid, "main_window", args, n);
  XtManageChild(main_win);

  n=0;
  menu_bar = XmCreateMenuBar(main_win, "main_list", args, n);
  XtManageChild(menu_bar);


  /* create quit widget + callback */
  n=0;
  XtSetArg(args[n], XmNmnemonic, 'Q'); n++;
  quit = XmCreateCascadeButton(menu_bar, "Quit", args, n); n++;
  XtManageChild(quit);
  XtAddCallback(quit, XmNactivateCallback, quit_call, NULL);
```

```
   /* create help widget + callback */

  n=0;
  XtSetArg(args[n], XmNmnemonic, 'H'); n++;
  help = XmCreateCascadeButton(menu_bar, "Help", args, n); n++;
  XtManageChild(help);

  XtAddCallback(help, XmNactivateCallback, help_call, NULL);


  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*---------------*/
void quit_call()
/*---------------*/

{
  printf("On sort du programme!\n");
  exit(0);
}

/*---------------*/
void help_call()
/*---------------*/

{
  printf("Désolé je ne peux pas vous aider\n");
}
```

Les premières lignes de ce programme devraient maintenant vous être familières.

Dans la fenêtre top_level nous créons une MainWindow (variable main_win), nous créons aussi une MenuBar (variable menu_bar) comme enfant de main_win, et finalement nous créons deux CascadeButton (quit et help) dans la MenuBar.

En général, un widget de type CascadeButton sert à indiquer le nom d'un menu déroulant. Lorsqu'on clique dessus le menu se déroule en cascade, c'est pourquoi on les appelle des CascadeButton. Ils se conduisent exactement comme des PushButton et peuvent avoir des callbacks (ressource ***XmNactivateCallback***).

Un CascadeButton peut très bien ne pas avoir de menu déroulant. Il se conduit alors exactement comme un bouton poussoir. C'est le cas dans le programme menu_cascade.c. Dans ce programme d'exemple, nous avons créé deux boutons auxquels nous avons attaché des callbacks très simples.

Raccourcis clavier : nous avons installé pour chacun des boutons un raccourcis clavier équivalent à un click souris sur le bouton. Dans l'exemple, si on tape meta-q on sort du programme comme si on avait cliqué sur le bouton quit. Idem pour le bouton help : l'appui de meta-h est équivalent à clicker dessus.

Nous avons utilisé pour celà la Ressource ***XmNmnemonic***. Son utilisation est très simple (voir ***menu_cascade.c***).

Autre exemple d'école :

On va tout d'abord créer une MainWindow contenant une MenuBar et

un widget de type Frame dans la WorkArea.

```
#include <Xm/Xm.h>
#include <Xm/RepType.h>

#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/RowColumn.h>
#include <Xm/Frame.h>

/*---------------------------*/
main(int argc, char *argv[])
/*---------------------------*/
{
  XtAppContext app_context;
  Widget topLevel, mainWindow, menuBar, frame;
  Widget fileButton, fileMenu, quit, help, helpButton, helpMenu, helpBox;

  /* On indique que l'on desire utiliser le langage par defaut
     (anglais). Cette fonction initialise le support international de
     l'application. Les parametres inidiquent que l'on prend les
     valeurs par defaut */
  XtSetLanguageProc(NULL, (XtLanguageProc) NULL, NULL);

  topLevel = XtVaAppInitialize(&app_context, "XMainWIndow",
                        NULL, 0, &argc, argv, NULL, NULL);

  /* On cree la main window */
  mainWindow = XtVaCreateManagedWidget("mainWindow",
                                xmMainWindowWidgetClass,
                                topLevel,
                                NULL);

  /* On cree la barre de menus. On utilise une fonction Xm ici car
     la fonction XmCreateMenuBar effectue des operations qui n'ont pas
     d'equivalent avec Xt */
  menuBar = XmCreateMenuBar(mainWindow, "menuBar", NULL, 0);
  XtManageChild(menuBar);


  /* On cree une frame dans la work area de la main window */
  frame = XtVaCreateManagedWidget("frame",
                             xmFrameWidgetClass,
                             mainWindow,
                             NULL);

  /* On positionne les parties de la main window */
  XmMainWindowSetAreas(mainWindow, menuBar, NULL, NULL, NULL, frame);

  XtRealizeWidget(topLevel);
  XtAppMainLoop(app_context);
}
```

Exercice: créér le fichier de ressources pour cette application tel que la MainWindow soit large de 200 pixels, et la Frame fasse 300 x 300 pixels. La

WorkingArea de la MainWindow étant en fait une ScrolledWindow, on pourra positionner la ressource scrollingPolicy de la MainWindow pour faire apparaître des ScrollBars.

Nous étudierons plus en détails les ScrolledWindows et les ScrollBars dans la suite du cours.

On va maintenant ajouter des menus à cette application.

6.5. Création d'un menu.

Motif fournit trois types de menus : Popup, Pulldown et Option. Les Popups menus apparaissent à l'écran lorsqu'on presse une touche particulière, une combinaison de touches, ou un bouton de la souris dans un widget. Les Pulldown menus et les Option menus possèdent des boutons à l'ecran qui déclenchent leur apparition. Les Pulldown menus sont des menus déroulants (sous le bouton qui a déclenché leur déroulement) alors que les Option menus apparaissent "sur le bouton", parfois en montant.

Les Option menus se souviennent de la dernière séléction. Les menus accrochés à une MenuBar sont des PulldownMenus.

6.6. Etapes.

- 1.Créer un CascadeButton comme fils de la MenuBar. C'est le bouton qui va déclencher l'apparition du menu.
- 2.Créer un PulldownMenu vide, fils de la MenuBar, qui va contenir les différents items. On utilisera la fonction XmCreatePulldownMenu(). Attention: il ne faut pas le manager puisque le menu apparaît à la demande. C'est le CascadeButton qui s'occupera de ce travail pour nous! Pour information, un PulldownMenu est en réalité une RowColumn, tout comme la MenuBar.
- 3.Créer des PushButtons comme fils du Pulldown.
- 4.Indiquer au CascadeButton le menu qu'il doit dérouler. Cette action est réalisée en positionnant la ressource ***XmNsubMenuId*** du CascadeButton avec l'identificateur du ***PullDownMenu***.

Code de la nouvelle version :

```
#include <Xm/Xm.h>
#include <Xm/RepType.h>

#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/RowColumn.h>
```

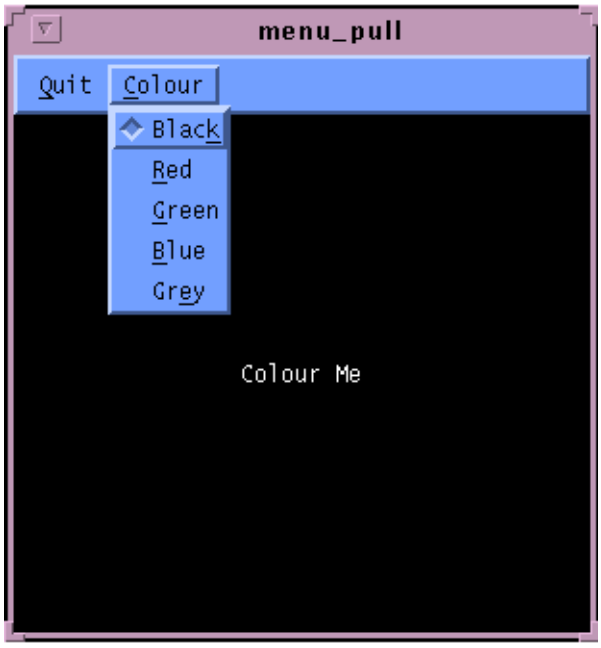
```
#include <Xm/Frame.h>
#include <Xm/PushB.h>

void QuitCB(Widget w, XtPointer client_data,
        XmPushButtonCallbackStruct *cbs);

/*---------------------------*/
main(int argc, char *argv[])
/*---------------------------*/
{
    .
    .
    .
    .
  /* Creation du menu:
     1) On cree un cascade bouton File dans la menubar */
  fileButton = XtVaCreateManagedWidget("fileButton",
                                xmCascadeButtonWidgetClass,
                                menuBar,
                                NULL);

  /* 2) Creation du PullDownMenu vide. ATTENTION: on utilise une
     fonction Xm! Remarquez que le menu n'est pas manage !*/
  fileMenu = XmCreatePulldownMenu(menuBar,
                             "fileMenu",
                             NULL,
                             0);

  /* 3) Creation d'un bouton quit dans le menu File */
  quit = XtVaCreateManagedWidget("quit",
                            xmPushButtonWidgetClass,
                            fileMenu,
                            NULL);

  /* 4) On indique au CascadeButton le menu qu'il doit derouler */
  XtVaSetValues(fileButton,
              XmNsubMenuId, fileMenu,
              NULL);

  /* On cree un callback pour le bouton quit */
  XtAddCallback(quit, XmNactivateCallback, QuitCB, 0);


  XtRealizeWidget(topLevel);
  XtAppMainLoop(app_context);
}


/*-----------------------------------------------*/
void QuitCB(Widget w, XtPointer client_data,
        XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------*/
{
  exit(0);
}
```

6.7. Exercices.

Essayez de créer un des sous-menus. Il suffit de mettre un CascadeButton dans le Pulldown Menu et de recommencer le processus de création d'un menu que l'on attacchera á ce bouton.

Crééz une Form dans la Frame et positionnez des objets à l'intérieur. Observez le comportement des ScrollBars.

6.8. Un exemple un peu plus compliqué.



Nous allons développer le programme menu_pull.c qui va contenir deux menus déroulants. Le premier, “Quitter”, va contenir un seul item (une seule entrée) qui nous permettra de sortir du programme.

Le second menu, “Couleur” proposera 5 items qui permettront de changer la couleur du fond du Label widget attaché à la MainWindow.

Le début du programme ressemble à l’exemple précédent : on crée un top_level, on crée une MainWindow dans le top_level, une MenuBar dans la MainWindow, des CascadeButton dans la MenuBar, et ensuite, plein de trucs nouveaux ! Alors etudiez-bien ce petit bout de code sans vous affoler, et rendez-vous après ces quelques lignes de source C !

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/Label.h>
#include <Xm/RowColumn.h>

Widget top_wid, label, main_w;

String colours[] = { “Black”,  “Red”, “Green”, “Blue”, “Grey”};

Display *display; /* xlib id of display */

Colormap cmap;


/*---------------*/
main(argc, argv)
/*---------------*/
int argc;
char *argv[];
```

```
{
 Widget  menubar, menu, quit_w, widget;
 XtAppContext app;


 XColor back, fore, spare;
 XmString  quit, colour, red, green, blue, black, grey, label_str;
 void quit_call(), colour_call();

 int n = 0;
 Arg args[2];

 /* Initialize toolkit */
 top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                    &argc, argv, NULL, NULL);


 /* MainWindow will contain a MenuBar and a Label */
 n=0;
 XtSetArg(args[n], XmNwidth, 300); n++;
 XtSetArg(args[n], XmNheight, 300); n++;
 main_w = XmCreateMainWindow(top_wid, "main_window", args, n);
 XtManageChild(main_w);


 /* Create a simple MenuBar that contains three menus */
 quit = XmStringCreateSimple("Quit");
 colour = XmStringCreateSimple("Colour");


 menubar = XmVaCreateSimpleMenuBar(main_w, "menubar",
                          XmVaCASCADEBUTTON, quit, 'Q',
                          XmVaCASCADEBUTTON, colour, 'C',
                          NULL);


 XmStringFree(colour); /* finished with this so free */


 /* First menu is the quit menu -- callback is quit_call() */
 quit_w = XmVaCreateSimplePulldownMenu(menubar, "quit_menu",
                            0, quit_call,
                            XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
                            NULL);
 XmStringFree(quit);


 /* Second menu is the color menu -- callback
    is colour_call() */
 black = XmStringCreateSimple(colours[0]);
 red = XmStringCreateSimple(colours[1]);
 green = XmStringCreateSimple(colours[2]);
 blue = XmStringCreateSimple(colours[3]);
 grey = XmStringCreateSimple(colours[4]);


 menu = XmVaCreateSimplePulldownMenu(menubar, "edit_menu",
      1, colour_call,
      XmVaRADIOBUTTON, black, 'k', NULL, NULL,
      XmVaRADIOBUTTON, red, 'R', NULL, NULL,
      XmVaRADIOBUTTON, green, 'G', NULL, NULL,
      XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
      XmVaRADIOBUTTON, grey, 'e', NULL, NULL,
      /* RowColumn resources to enforce */
      XmNradioBehavior, True,
      /* radio behavior in Menu */
      XmNradioAlwaysOne, True,
      NULL);

 XmStringFree(black);
 XmStringFree(red);
 XmStringFree(green);
 XmStringFree(blue);
```

```
  /* Initialize menu so that “black” is selected. */
  if (widget = XtNameToWidget(menu, “button_0”))
    XtVaSetValues(widget, XmNset, True, NULL);

  XtManageChild(menubar);

  /* create a label text widget that will
     be “work area” we colour */
  label_str = XmStringCreateSimple(“Colour Me”);

  label = XtVaCreateManagedWidget(“main_window”,
                          xmLabelWidgetClass,   main_w,
                          XmNlabelString, label_str,
                          NULL);

  XmStringFree(label_str);

  /* set the label as the “work area”
     of the main window */
  XtVaSetValues(main_w,
              XmNmenuBar,    menubar,
              XmNworkWindow, label,
              NULL);


  /* background pixel to black foreground to white */
  cmap = DefaultColormapOfScreen(XtScreen(label));
  display = XtDisplay(label);

  XAllocNamedColor(display, cmap, colours[0], &back, &spare);
  XAllocNamedColor(display, cmap, “white”, &fore, &spare);

  XtSetArg(args[n],XmNbackground, back.pixel);
  ++n;
  XtSetArg(args[n],XmNforeground, fore.pixel);
  ++n;
  XtSetValues(label, args, n);

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*-------------------------*/
void quit_call(w, item_no)
/*-------------------------*/
/* Any item the user selects from the File menu calls
   this function. It will “Quit” (item_no == 0).
*/
Widget w;              /* menu item that was selected */
int item_no;           /* the index into the menu */
{
  if (item_no == 0)     /* the “quit” item */
    exit(0);
}

/*-------------------------*/
void colour_call(w, item_no)
/*-------------------------*/
/* called from any of the “Colour” menu items.
   Change the color of the label widget.
   Note: we have to use dynamic setting with setargs()..
*/
Widget w;              /* menu item that was selected */
int item_no;           /* the index into the menu */
{
  int n =0;
  Arg args[1];

  XColor xcolour, spare; /* xlib color struct */

  if (XAllocNamedColor(display, cmap, colours[item_no],
                  &xcolour, &spare) == 0)
    return;
```

```
  XtSetArg(args[n],XmNbackground, xcolour.pixel);
  ++n;
  XtSetValues(label, args, n);
}
```

Bon. Respirez-un grand coup... Hmmmmmffffff ! Pas de panique !

Beaucoup de nouveautés ici, des fonctions bizarres qui ressemblent à celles déjà étudiées mais légèrement différentes, de la couleur (oui, de la couleur !), des chaînes de caractères utilisées de manière bizarre, etc… Nous allons étudier tout ceci calmement...

La fonction ***XmVaCreateSimpleMenuBar***(): dans cet exemple, nous avons créé la MenuBar à l'aide de la fonction ***XmVaCreateSimpleMenuBar***() qui est plus facile à utiliser que la fonction ***XmCreateMenuBar***() de l'exemple précédent. Elle permet d'indiquer les CascadeButton que l'on veut mettre dans la barre de menu dès la création de cette dernière.

Cette fonction possède trois types d'arguments :

- Un pointeur sur le widget père.
- Un nom de type chaîne de caractères.
- Une liste de CascadeButton terminée par NULL.

Trois paramètres sont nécessaires pour spécifier un CascadeButton :

- 1.Le nom de la ressource : XmVaCASCADEBUTTON permet d'initialiser le texte qui apparait sur le bouton (son label).
- 2.La valeur du label qui doit être de type XmString. On a converti une chaîne de caractère normale à l'aide de la fonction XmStringCreateSimple() qui est détaillée dans la partie du cours dédiée aux chaînes de caractères sous MOTIF.
- 3.Un raccourcis clavier.

Important : il faut prendre l'habitude de libèrer systématiquement la place occupée par une ***XmString*** dès qu'on en a plus besoin. Cette opération est réalisée en appelant la fonction ***XmStringFree***().

Remarque : les fonctions contenant "Va" après "Xt" ou "Xm" créent des widgets qui n'ont pas besoin d'être managés (pas besoin d'appel à ***XtManageChild***()) la fonction ***XmVaCreateSimplePulldownMenu***() :

Pour créer un menu déroulant, il faut créer un widget de type ***PulldownMenu***. Pour attacher un PulldownMenu à une barre de menu, il suffit de lui donner comme père un des CascadeButton de la barre de menu.

Regardons comment nous avons créé le menu déroulant Quit :

```
quit_w = XmVaCreateSimplePulldownMenu(menubar, “quit_menu”,
                    0, quit_call,
                    XmVaPUSHBUTTON, quit, ‘Q’, NULL, NULL,
                    NULL);
```

Etudions ensemble les paramètres de la fonction ***XmVaCreateSimplePulldownMenu***() :

- Un pointeur sur le widget père (menu_bar dans cet exemple).
- Un nom de type chaîne de caractères.
- Un entier qui est l’identificateur du CascadeButton auquel le PulldownMenu va s’attacher. Ici la valeur de cet Id est 0, le menu va donc s’attacher au CascadeButton Quit. Si cet Id avait valu 1, le menu se serait attaché au deuxième bouton, c’est-a-dire au bouton Colour.
- Le quatrième paramètre spécifie un callback associé à n’importe quel xhoix dans ce menu. Tous les choix appeleront le même callback. Remarque : on ne spécifie pas la fonction de callback à l’aide de XtAddCallback() dans ce cas.
- On a ensuite une liste terminée par NULL qui spécifie les différents choix dans le menu. Chaque élément de cette liste comprend cinq paramètres :
  - 1.XmVaPUSHBUTTON
  - 2.Un élément de type XmString qui va contenir le label du choix correspondant dans le menu.
  - 3.Un raccourcis clavier.
  - 4.Les deux derniers paramètres sont réservés aux utilisateurs avancés de MOTIF et sortent du cadre de ce cours.

Le menu Colour est créé de manière similaire si ce n’est qu’il possède cinq choix au lieu d’un seul dans le menu Quit.

6.9. Les callbacks des menus.

Il reste à étudier une dernière chose : comment traiter les choix faits par l’utilisateur lors de l’exécution de notre programme ?

Chaque PullDown menu possède sa propre fonction de callback. Les fonctions de callback des menus possèdent deux paramètres :

- Un pointeur sur le widget qui a provoqué l'appel de la fonction de callback.
- Un index correspondant au numéro du choix que l'utilisateur a fait.

Ainsi, dans la fonction de callback du menu Quit, ***quit_call***(), une seule séléction est possible (la variable ***item_no*** doit être égale à zéro).

Dans la fonction de ***call*** du menu Colour, ***colour_call***(), la valeur du choix va provoquer une action (changer la couleur du fond de widget de type Label qui se trouve dans la work area de la MainWindow.

Ne vous affolez pas à propos de la couleur, nous l'etudierons en détail dans les prochains cours. Pour comprendre l'exemple, il suffit de savoir que la structure de données xcolour, de type XColor stocke la valeur de la couleur que nous désirons dans le champ xcolour.pixel après allocation par l'appel de ***XAllocNamedColor***(). Il suffit ensuite de positionner la ressource ***XmNbackgound*** du widget de type Label à l'aide des fonctions ***XtSetArg***() et ***XtSetValues***().

7. Les widgets de type Dialog.

7.1. Introduction.

7.1.1. Rôle des Dialog widgets.

Les widgets de type Dialog permettent le dialogue entre l'application et l'utilisateur. Les widgets de type Dialog "poppent" une fenêtre contenant un message et peuvent selon les cas demander à l'utilisateur d'entrer du texte au clavier.

Ils peuvent fournir en standard trois boutons poussoirs :

- 1.Un bouton poussoir Ok qui ferme la fenêtre de dialogue et valide l'entrée clavier effectuée par l'utilisateur, s'il y en a eu une.
- 2.Un bouton poussoir Cancel qui ferme la fenêtre de dialogue et ne valide pas l'entrée clavier.
- 3.Un bouton poussoir Help qui sert à afficher une aide en ligne pour l'utilisateur.

Exemple typique de Dialog widget : le message affiché par Xemacs lorsque vous n'avez pas sauvegardé vos fichier avant de compiler, le message affiché par Netscape "Unable to locate file", etc…

7.1.2. Les différents types de Dialog widgets :

Les Dialogs jouent de nombreux rôles : afficher de l'information, un message d'erreur, des warnings,

proposer l'entrée d'une commande. Ainsi, pour chacun de ces usages MOTIF propose un type de widgetdifférent :

- BulletinBoardDialog: Permet la création de widgets de dialogue customisés.
- ErrorDialog: Affichage de messages d'erreur.
- SelectionDialog: Sélection d'un item parmi une liste d'options.

- FileSelectionDialog: Widget spécialisé dans le choix de fichiers et/ou de répertoires.
- InformationDialog: Affichage de messages d'information, documentation en ligne, etc...
- PromptDialog: Permet la saisie clavier de données utilisateur.
- QuestionDialog: Demande à l'utilisateur d'effectuer un choix de type Oui/Non.
- WarningDialog: Affichage de messages d'alerte.
- WorkingDialog: Avertit l'utilisateur que l'application est en train d'effectuer une opération.

Pour créer un Dialog, il faut utiliser des fonctions de type ***XmCreate...Dialog***(). Par exemple, ***XmCreateFileSelectionBoxDialog***(). Pour faire apparaître un Dialog à l'écran, il suffit de le manager à l'aide de la fonction ***XtManageChild***().

7.2. Le widget de type WarningDialog.

Ce widget sert à signaler une erreur à l'utilisateur. Il peut s'agir d'une erreur du programme lui-même ou d'une erreur de manipulation de la part de l'utilisateur.

Un exemple typique d'utilisation : lorsque l'utilisateur clicke sur le bouton "Quit" d'une application, il vaut mieux lui demander s'il est bien sûr de vouloir effectuer cette action (surtout s'il n'a pas sauvegardé le travail en cours).

Nous allons bientôt étudier ensemble le petit programme dialog1.c qui

met en place un tel mécanisme :

Remarque : tout ce que le WarningDialog contient est une chaîne de caractères de type XmString qui informe l'utilisateur sur la nature du message.

Différentes étapes pour créer ce WarningDialog widget :

- Initialiser la variable de type XmString avec la valeur du message d'information. En l'occurence "Are you sure you want to quit?".
- Initialiser la ressource XmNmessageString du WarningDialog widget avec la variable précédente.
- Créer le WarningDialog widget à l'aide de la fonction ***XmCreateWarningDialog***().
- Ajouter des fonctions de callback aux boutons Ok, Cancel et Help du WarningDialog widget.

Dans un premier temps on se contentera d'ajouter un callback au bouton ***Ok*** en positionnant la ressource ***XmNokCallback*** du WarningDialog widget.

Manager le widget à l'aide de la fonction XtManageChild() afin de l'afficher.

L'utilisation de la fonction XtPopup() permet de l'afficher en le "poppant" à l'écran.

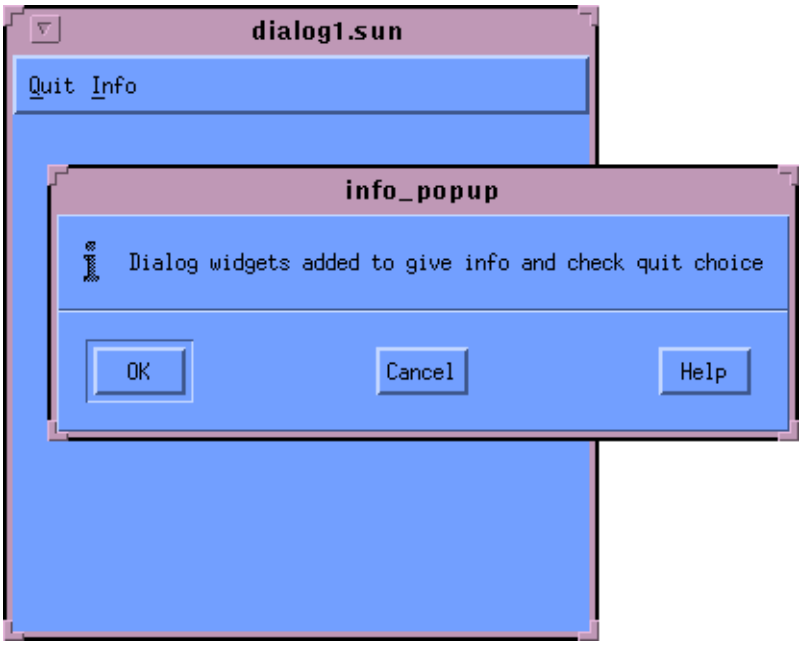
7.3. Le widget de type InformationDialog.

Ce widget ressemble énormément à celui de type WarningDialog. La seule différence est le dessin de l'icône qui est affiché juste avant le message (Un grand "i" au lieu d'un point d'exclamation).

En général les InformationDialog sont utilisés pour afficher une petite aide en ligne ou bien des informations concernant l'application (par exemple un menu contenant une entrée "about" pourra popper un InformationDialog widget contenant le numéro de la version, les auteurs, etc...").

Pour créer un InformationDialog, on procède de la même manière que pour le ***WarningDialog***, excepté que l'on utilise la fonction ***XmCreateInformationDialog***().

Exemple d'InformationDialog widget :



Le programme dialog1.c :

Ce petit programme contient des widgets de type InformationDialog et WarningDialog.

Etant donné que ces widgets sont dérivés de la classe MessageBox, nous devons inclure le fichier <Xm/MessageB.h>.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
```

```
/*---------------------------*/
main(int argc, char *argv[])
/*---------------------------*/
{
  XtAppContext app;
  Widget top_wid, main_w, menu_bar, info, quit;


  /* callback for the pushbuttons.  pops up dialog */
  void info_pop_up(), quit_pop_up();

  top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                        &argc, argv, NULL, NULL);

  main_w = XtVaCreateManagedWidget("main_window",
                            xmMainWindowWidgetClass,   top_wid,
                            XmNheight, 300,
                            XmNwidth,300,
                            NULL);

  menu_bar = XmCreateMenuBar(main_w, "main_list",
                        NULL, 0);
  XtManageChild(menu_bar);


  /* create quit widget + callback */

  quit = XtVaCreateManagedWidget( "Quit",
                            xmCascadeButtonWidgetClass, menu_bar,
                            XmNmnemonic, 'Q',
                            NULL);

  /* Callback has data passed to */
  XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
            "Are you sure you want to quit?");



  /* create help widget + callback */
  info = XtVaCreateManagedWidget( "Info",
                            xmCascadeButtonWidgetClass, menu_bar,
                            XmNmnemonic, 'I',
                            NULL);

  XtAddCallback(info, XmNactivateCallback, info_pop_up,
            "Dialog widgets added to give info and check quit choice");

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*-----------------------------------------------------*/
void info_pop_up(Widget cascade_button, char *text,
              XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------------*/
{
  Widget dialog;
  XmString xm_string;
  extern void info_activate();
  Arg args[1];
  int n;

  /* label the dialog */
  xm_string = XmStringCreateSimple(text);

  n=0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* Create the InformationDialog as child
     of cascade_button passed in */
  dialog = XmCreateInformationDialog(cascade_button,
                              "info", args, n);

  XmStringFree(xm_string);
```

```
  XtAddCallback(dialog, XmNokCallback, info_activate,
                NULL);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*-----------------------------------------------------*/
void quit_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------------*/
{
  Widget dialog;
  XmString xm_string;
  void quit_activate();
  Arg args[1];
  int n;

  /* label the dialog */
  xm_string = XmStringCreateSimple(text);

  n=0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* Create the WarningDialog */
  dialog = XmCreateWarningDialog(cascade_button, "quit",
                                 args, n);

  XmStringFree(xm_string);

  XtAddCallback(dialog, XmNokCallback, quit_activate,
                NULL);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/* callback routines for dialogs */

/*-----------------------------------------------------*/
void info_activate(Widget dialog)
/*-----------------------------------------------------*/
{
  printf("Info Ok was pressed.\n");
}


/*-----------------------------------------------------*/
void quit_activate(Widget dialog)
/*-----------------------------------------------------*/
{
  printf("Quit Ok was pressed.\n");
  exit(0);
}
```

7.4. Les widgets de type ErrorDialog, WorkingDialog et QuestionDialog.

Ces trois widgets fonctionnent de la même manière que les widgets de type WarningDialog ou InformationDialog.

A titre d'exercice, modifiez le programme ***dialog1.c*** pour créer ces widgets et voir à quoi ils ressemblent.

Par défaut lorsque vous créez un des Dialog widget étudiés jusqu'à présent, MOTIF propose les trois boutons Ok, Cancel et Help.

Cependant dans certains cas on aimerait bien se débarraser d'un ou deux de ces boutons; par exemple ne conserver que le bouton Ok.

Pour supprimer un bouton :

- Utiliser la fonction ***XmMessageBoxGetChild (dialog, Xm_<type du bouton>_BUTTON)*** pour obtenir un pointeur vers l'ID du bouton que l'on veut supprimer. Les différents types de boutons sont :
  - ***XmDIALOG_OK_BUTTON***
  - ***XmDIALOG_CANCEL_BUTTON***
  - ***XmDIALOG_HELP_BUTTON***
- Unmanager le bouton dont on a récupéré l'ID à l'aide de la fonction XtUnmanageChild().

Par exemple, dans le programme ***dialog1.c***, si l'on ne veut conserver que le bouton Ok dans l'InformationDialog, on ajoutera les lignes suivantes justes après l'appel de la fonction ***XmCreateInformationDialog***() :

```
Widget button;
.
.
.
dialog = XmCreateInformationDialog(cascade_button, "info", args, 1);
button = XmMessageBoxGetChild(dialog,
                    XmDIALOG_CANCEL_BUTTON);
XtUnmanageChild(button);
button = XmMessageBoxGetChild(dialog,
                    XmDIALOG_HELP_BUTTON);
XtUnmanageChild(button);
```

7.5. Le widget de type PromptDialog :

Le PromptDialog widget invite l'utilisateur à entrer des données au clavier.

Pour créer un PromptDialog, on utilise la fonction ***XmCreatePromptDialog***().

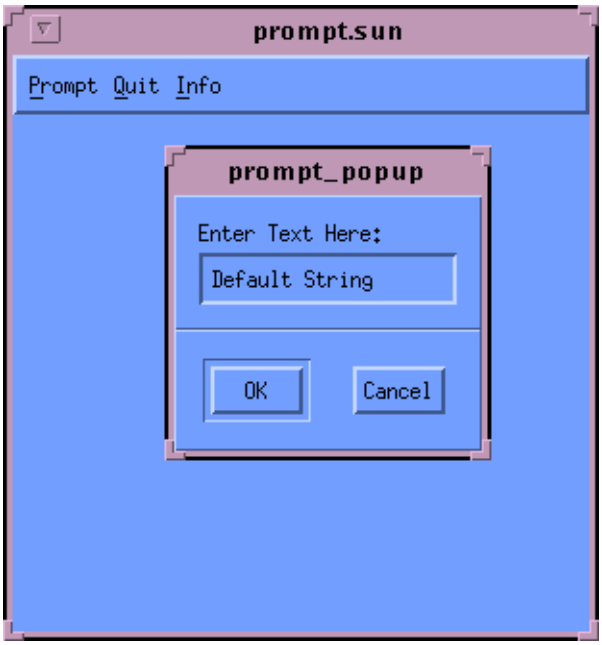
Ce widget possède deux ressources importantes :

- ***XmNselectionLabelString*** : le message d'invitation.
- ***XmNtextString*** : valeur par défaut des données. Si l'utilisateur clicke sur le bouton Ok sans avoir rien entré au clavier, la valeur par défaut de sa saisie sera celle spécifiée par la ressource ***XmNtextString***.

Remarque : le PromptDialog est basé sur la classe SelectionBox, pour compiler sans erreur il faut donc inclure le fichier <***Xm/SelectionB.h***>.

Nous allons ensemble étudier le petit programme ***prompt.c***, extension du programme ***dialog2.c*** étudié précédemment.

Exemple de PromptDialog : celui créé par le programme prompt.c :



Le programme prompt.c :

Nous ajoutons un nouveau menu “Prompt” à l’exemple précédent (***dialog2.c***). Ce menu ne contient qu’un CascadeButton qui décenche l’affichage du PromptWidget.

Le texte saisi par l’utilisateur sera affiché dans un InformationDialog créé par la fonction de callback du PromptDialog, la fonction ***prompt_activate***().

Une fonction de callback d’un PromptDialog possède la structure suivante :

```
void prompt_callback(Widget widget,
                     XtPointer client_data,
                     XmSelectionBoxCallbackStruct *selection)
```

En général, nous sommes surtout intéressés par la chaîne de caractères qui a été saisie dans le PromptDialog. Cette information est située dans le champ value (type ***XmString***) de la structure ***XmSelectionBoxCallbackStruct***.

Dans la fonction de callback du PromptDialog, ***prompt_activate***() (voir ci-dessus l’exemple de code), cette valeur se trouve dans selection->value. La fonction utilise cette valeur pour mettre à jour la ressource ***XmNmessageString*** de l’InformationDialog.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/SelectioB.h>
#include <Xm/RowColumn.h>

void ScrubDial(Widget, int);

Widget top_wid;

/*-------------------------*/
main(int argc, char *argv[])
/*-------------------------*/
{

  XtAppContext app;
  Widget main_w, menu_bar, info, prompt, quit;
  void info_pop_up(), quit_pop_up(), prompt_pop_up();


  top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                        &argc, argv, NULL, NULL);

  main_w = XtVaCreateManagedWidget("main_window",
                           xmMainWindowWidgetClass, top_wid,
                           XmNheight, 300,
                           XmNwidth,300,
                           NULL);


  menu_bar = XmCreateMenuBar(main_w, "main_list",
                       NULL, 0);
  XtManageChild(menu_bar);

  /* create prompt widget + callback */
  prompt = XtVaCreateManagedWidget( "Prompt",
                            xmCascadeButtonWidgetClass, menu_bar,
                            XmNmnemonic, 'P',
                            NULL);

  /* Callback has data passed to */
  XtAddCallback(prompt, XmNactivateCallback,
            prompt_pop_up, NULL);



  /* create quit widget + callback */
  quit = XtVaCreateManagedWidget( "Quit",
                           xmCascadeButtonWidgetClass, menu_bar,
                           XmNmnemonic, 'Q',
                           NULL);


  /* Callback has data passed to */
  XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
            "Are you sure you want to quit?");



  /* create help widget + callback */

  info = XtVaCreateManagedWidget( "Info",
                           xmCascadeButtonWidgetClass, menu_bar,
                           XmNmnemonic, 'I',
                           NULL);

  XtAddCallback(info, XmNactivateCallback, info_pop_up,
            "Select Prompt Option To Get Program Going.");

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}
```

```
/*--------------------------------------------------------*/
void prompt_pop_up(Widget cascade_button, char *text,
                XmPushButtonCallbackStruct *cbs)
/*--------------------------------------------------------*/
{
  Widget dialog, remove;
  XmString xm_string1, xm_string2;
  void prompt_activate();
  Arg args[3];
  int n;

  /* label the dialog */
  xm_string1 = XmStringCreateSimple("Enter Text Here:");
  /* default text string  */
  xm_string2 = XmStringCreateSimple("Default String");

  n=0;
  XtSetArg(args[n], XmNselectionLabelString, xm_string1); n++;
  XtSetArg(args[n], XmNtextString, xm_string2); n++;
   /* set up default button for cancel callback */
  XtSetArg(args[n], XmNdefaultButtonType,
         XmDIALOG_CANCEL_BUTTON);  n++;

  /* Create the WarningDialog */
  dialog = XmCreatePromptDialog(cascade_button, "prompt",
                          args, n);

  XmStringFree(xm_string1);
  XmStringFree(xm_string2);


  XtAddCallback(dialog, XmNokCallback, prompt_activate,
              NULL);
  /* Scrub Prompt Help Button */
  remove = XmSelectionBoxGetChild(dialog,
                           XmDIALOG_HELP_BUTTON);

  XtUnmanageChild(remove);  /* scrub HELP button */

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*--------------------------------------------------------*/
void info_pop_up(Widget cascade_button, char *text,
                XmPushButtonCallbackStruct *cbs)
/*--------------------------------------------------------*/
{
  Widget dialog;
  XmString xm_string;
  extern void info_activate();
  Arg args[2];
  int n;

  /* label the dialog */
  xm_string = XmStringCreateSimple(text);

  n=0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* set up default button for OK callback */
  XtSetArg(args[n], XmNdefaultButtonType,
         XmDIALOG_OK_BUTTON); n++;


  /* Create the InformationDialog as child of
      cascade_button passed in */
  dialog = XmCreateInformationDialog(cascade_button,
                              "info", args, n);

  ScrubDial(dialog, XmDIALOG_CANCEL_BUTTON);
  ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

  XmStringFree(xm_string);
```

```
  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*-----------------------------------------------------*/
void quit_pop_up(Widget cascade_button, char *text,
              XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------------*/
{
  Widget dialog;
  XmString xm_string;
  void quit_activate();
  Arg args[1];
  int n;

  /* label the dialog */
  xm_string = XmStringCreateSimple(text);

  n=0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* set up default button for cancel callback */
  XtSetArg(args[n], XmNdefaultButtonType,
          XmDIALOG_CANCEL_BUTTON); n++;
  /* Create the WarningDialog */
  dialog = XmCreateWarningDialog(cascade_button, "quit",
                              args, n);

  ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

  XmStringFree(xm_string);

  XtAddCallback(dialog, XmNokCallback, quit_activate,
               NULL);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*-------------------------------------*/
void ScrubDial(Widget wid, int dial)
/*-------------------------------------*/
/* routine to remove a DialButton from a Dialog */
{
  Widget remove;

  remove = XmMessageBoxGetChild(wid, dial);
  XtUnmanageChild(remove);
}

/*-------------------------------------------------------------*/
void prompt_activate(Widget widget, XtPointer client_data,
                  XmSelectionBoxCallbackStruct *selection)
/*-------------------------------------------------------------*/
/* callback function for Prompt activate */
{
  Widget dialog;
  Arg args[2];
  XmString xm_string;
  int n;

  /* compose InformationDialog output string */
  /* selection->value holds XmString entered to prompt */

  xm_string = XmStringCreateSimple("You typed: ");
  xm_string = XmStringConcat(xm_string,selection->value);

  n=0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* set up default button for OK callback */
  XtSetArg(args[n], XmNdefaultButtonType,
          XmDIALOG_OK_BUTTON);  n++;
  /* Create the InformationDialog to echo
      string grabbed from prompt */
```

```
  dialog = XmCreateInformationDialog(top_wid,
                              "prompt_message", args, n);

  ScrubDial(dialog, XmDIALOG_CANCEL_BUTTON);
  ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*---------------------------------*/
void quit_activate(Widget dialog)
/*---------------------------------*/
/* callback routines for quit ok dialog */
{
  printf("Quit Ok was pressed.\n");
  exit(0);
}
```

7.6. Widgets de type SelectionDialog et FileSelectionDialog.

7.6.1. Présentation.

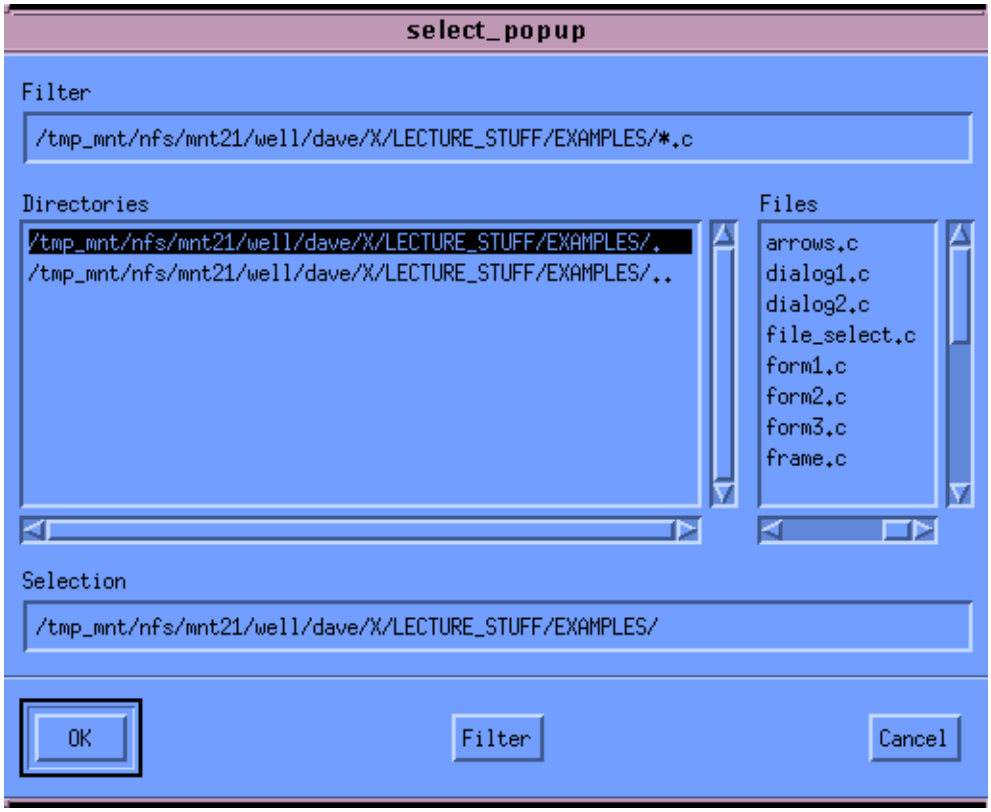
Ces deux widgets permettent à l'utilisateur d'effectuer un ou plusieurs choix dans une liste d'items.

Nous allons étudier le FileSelectionDialog car bien que plus complexe à utiliser que le SelectionDialog son usage est plus fréquent.

Le FileSelectionDialog est spécialisé dans la séléction de fichier(s) dans un répertoire. On le rencontre dans toutes les applications qui nécessitent la lecture/écriture de fichiers.

Le widget de type SelectionDialog est moins spécialisé et permet de séléctionner des objets moins spécifiques que des fichiers.

Exemple de FileSelectionDialog:



Pour créer un FileSelectionDialog, on utilise la fonction ***XmCreateFileSelectionDialog***(). Pour compiler sans erreur, il faut inclure le fichier ***<Xm/FileSB.h>*** car le widget de type FileSelectionDialog est basé sur la classe FileSelectionBox.

Remarque : si on utilise des fonctions de manipulation des SelectionDialog telles que ***XmSelectionBoxGetChild***() nécessaire si on veut enlever un des boutons par défaut d'un ***FileSelectionDialog*** (voir programme ***file_select.c***), nous devons aussi inclure <Xm/SelectioB.h> pour ne pas qu'il y ait d'erreur de compilation .

7.6.2. Ressources des FileSelectionDialog.

Un FileSelectionDialog possède de nombreuses ressources que l'on peut positionner pour contrôler la recherche de fichiers, utiliser des méta-caractères tels que '*' pour ne proposer que les fichiers ayant un suffixe donné, etc... En général ces ressources sont du type XmString.

- ***XmNdirectory***: Le répertoire oú se trouvent les fichier que l'on va proposer à l'utilisateur. Par défault : le répertoire courant.

- ***XmNdirLabelString***: Texte affiché au-dessus de la boîte du FileSelectionDialog contenant la liste des noms de répertoires.
- ***XmNdirMask***: Masque utilisé pour filtrer certains noms de fichiers. Par exemple, dans le programme ***file_select.c*** ce masque vaut ****.c*** afin de ne proposer que des fichiers sources C dans le FileSelctionDialog.
- ***XmNdirSpec*** : Le nom du fichier choisi par défaut.
- ***XmNfileListLabelString***: Texte affiché au-dessus de la boîte du FileSelectionDialog contenant la liste des noms de fichiers.
- ***XmNfileTypeMask***: Type des fichiers devant être listés. Valeurs possibles : ***XmFILE_REGULAR***, ***XmFILE_DIRECTORY, XmFILE_TYPE_ANY.***
- ***XmNfilterLabelString***: Texte affiché au-dessus de la boîte du FileSelectionDialog contenant le filtre sur les noms de fichiers.

Certaines de ces resssources peuvent être modifiées pendant l'exécution du programme par l'utilisateur, en saisissant du texte directement dans les fenêtres correspondantes du FileSelectionDialog.

7.6.3. Le programme file_select.c.

Ce petit programme ouvre un FileSelectionDialog avec un filtre pour ne proposer que des fichiers sources C (valeur du filtre ***XmNdirMask*** : ****.c***). Lorsqu'un fichier est séléctionné son listing est imprimé sur ***stdout***.

```
#include

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/SelectioB.h>
#include <Xm/PushB.h>
#include <Xm/FileSB.h>
#include <Xm/RowColumn.h>

void ScrubDial(Widget, int);

Widget top_wid;

/*----------------------------*/
main(int argc, char *argv[])
/*----------------------------*/
{
  XtAppContext app;
  Widget main_w, menu_bar, file_select, quit;
  void  quit_pop_up(), select_pop_up(); /* callbacks */
```

```
  top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                        &argc, argv, NULL, NULL);

  main_w = XtVaCreateManagedWidget("main_window",
       xmMainWindowWidgetClass,   top_wid,
       XmNheight, 300,
       XmNwidth,300,
       NULL);


  menu_bar = XmCreateMenuBar(main_w, "main_list",
                        NULL, 0);
  XtManageChild(menu_bar);


  /* create prompt widget + callback */

  file_select = XtVaCreateManagedWidget( "Select",
       xmCascadeButtonWidgetClass, menu_bar,
       XmNmnemonic, 'S',
       NULL);


  /* Callback has data passed to */
  XtAddCallback(file_select, XmNactivateCallback,
             select_pop_up, NULL);

  /* create quit widget + callback */
  quit = XtVaCreateManagedWidget( "Quit",
       xmCascadeButtonWidgetClass, menu_bar,
       XmNmnemonic, 'Q',
       NULL);


  /* Callback has data passed to */
  XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
               "Are you sure you want to quit?");

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*-------------------------------------------------------*/
void select_pop_up(Widget cascade_button, char *text,
                XmPushButtonCallbackStruct *cbs)
/*-------------------------------------------------------*/
{
  Widget dialog, remove;
  void select_activate(), cancel();
  XmString mask;
  Arg args[1];
  int n;

  /* Create the FileSelectionDialog */

  mask  = XmStringCreateSimple("*.c");

  n=0;
  XtSetArg(args[n], XmNdirMask, mask); n++;
  dialog = XmCreateFileSelectionDialog(cascade_button,
                                "select", args, n);

  XtAddCallback(dialog, XmNokCallback, select_activate,
             NULL);
  XtAddCallback(dialog, XmNcancelCallback, cancel, NULL);

  remove = XmSelectionBoxGetChild(dialog,
                             XmDIALOG_HELP_BUTTON);

  XtUnmanageChild(remove); /* delete HELP BUTTON */


  XtManageChild(dialog);
```

```
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*-----------------------------------------------------*/
void quit_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)
/*-----------------------------------------------------*/
{
  Widget dialog;
  XmString xm_string;
  void quit_activate();
  Arg args[2];
  int n;

  /* label the dialog */
  xm_string = XmStringCreateSimple(text);

  n = 0;
  XtSetArg(args[n], XmNmessageString, xm_string); n++;
  /* set up default button for cancel callback */
  XtSetArg(args[n], XmNdefaultButtonType,
           XmDIALOG_CANCEL_BUTTON); n++;
  /* Create the WarningDialog */
  dialog = XmCreateWarningDialog(cascade_button, "quit",
                                 args, n);

  ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

  XmStringFree(xm_string);

  XtAddCallback(dialog, XmNokCallback, quit_activate,
                NULL);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);
}

/*-----------------------------------*/
void ScrubDial(Widget wid, int dial)
/*-----------------------------------*/
/* routine to remove a DialButton from a Dialog */
{
  Widget remove;

  remove = XmMessageBoxGetChild(wid, dial);
  XtUnmanageChild(remove);
}

/*-------------------------------------------------------*/
void select_activate(Widget widget, caddr_t client_data,
                     XmFileSelectionBoxCallbackStruct *selection)
/*-------------------------------------------------------*/
/* callback function for Prompt activate */
/* function opens file (text) and prints to stdout */
{
  FILE *fp, *fopen();
  char *filename, line[200];
  void error();

  XmStringGetLtoR(selection->value,
                  XmSTRING_DEFAULT_CHARSET, &filename);


  if ((fp = fopen(filename,"r")) == NULL)
    error("CANNOT OPEN FILE", filename);
  else {
    while (!feof(fp)) {
      fgets(line,200,fp);
      printf("%s\n",line);
    }
    fclose(fp);
  }
}
```

```
/*-----------------------------------------------*/
void cancel(Widget widget, caddr_t client_data,
XmFileSelectionBoxCallbackStruct *selection)
/*-----------------------------------------------*/
{
  XtUnmanageChild(widget);  /* undisplay widget */
}

/*-------------------------------*/
void error(char *s1, char *s2)
/*-------------------------------*/
/* prints error to stdout */
{
  /* EXERCICE:

     REECRIRE CETTE ROUTINE POUR AFFICHER LE TEXTE DANS UN
     ErrorDialog widget. */

  printf("%s: %s\n", s1, s2);
  exit(-1);
}

/*---------------------------------*/
void quit_activate(Widget dialog)
/*---------------------------------*/
{
  /* callback routines for quit ok dialog */
  printf("Quit Ok was pressed.\n");
  exit(0);
}
```

8. Les widgets de type Text.

8.1. Introduction :

MOTIF fournit deux types de Text widgets:

- XmText: Ce widget est un petit éditeur de texte multilignes.
- XmTextField: Un éditeur comportant une seule ligne.

Dans certaines applications, l'édition de texte occupe une place très importante. Le fait que MOTIF fournisse de puissants outils de ce type facilite grandement la vie du programmeur. Par exemple, dans Netscape, lorsque vous utilisez l'option "View Source" une fenêtre contenant un éditeur de texte simplifié vous propose d'éditer le code HTML de la page courante. Il s'agit d'un Text widget de MOTIF!

Remarque : si on couple les Text widgets avec d'autres widgets déjà étudiés comme le FileSelectionDialog, on peut aisément écrire son propre éditeur de texte (il ne sera pas ridicule, surtout comparé à ***textedit***!)

Nous étudierons surtout le widget Text étant donné que le widget de type TextField n'en est qu'une version très simplifiée. (De plus, si on positionne la ressource ***XmNeditMode*** du widget Text avec la valeur ***XmSINGLE_LINE_EDIT***, on transforme le widget Text en widget de type TextField)

8.2. Création d'un Widget de type Text

Il existe plusieurs manières de créer un widget de type Text.

Nous pouvons utiliser les fonctions courantes maintenant bien assimilées ***XmCreateText***() ou ***XtVaCreateManagedWidget***().

Il ne faudra pas oublier de mettre le widget de type Text dans une ScrolledWindow car le texte édité pourra excèder la taille de la fenêtre. MOTIF fournit un méta widget de type ***ScrolledText*** qui comprend un Text widget dans une ScrolledWindow. On peut le créer à l'aide de la fonction ***XmCreateScrolledText***().

Ressources d'un Text widget :

- ***XmNrows***: Nombre de lignes visibles.
- ***XmNcolumns***: Nombre de colonnes visibles.
- ***XmNeditable***: Valeurs possibles : True ou False selon que l'édition du texte est autorisée ou pas.
- ***XmNscrollHorizontal***: Valeurs possibles : True ou False selon que l'on autorise le scrolling horizontal du texte lorsque la longueur des lignes excède la largeur de la fenêtre d'édition.
- ***XmNscrollVertical***: Valeurs possibles : True ou False selon que l'on autorise le scrolling vertical du texte lorsque le texte ne contient plus dans la page en hauteur.
- ***XmNeditMode***: Valeurs possibles : ***XmSINGLE_LINE_EDIT*** ou ***XmMULTI_LINE_EDIT*** selon que l'on désir éditer du texte sur une ou plusieurs lignes.

8.3. Mettre du texte dans un Text widget.

La fonction ***XmTextSetString***() met une chaîne de caractère ordinaire (au sens du langage C) dans un Text widget.

Elle accepte deux paramètres :

- 1.L'ID du Text widget.
- 2.La chaîne de caractères à éditer.

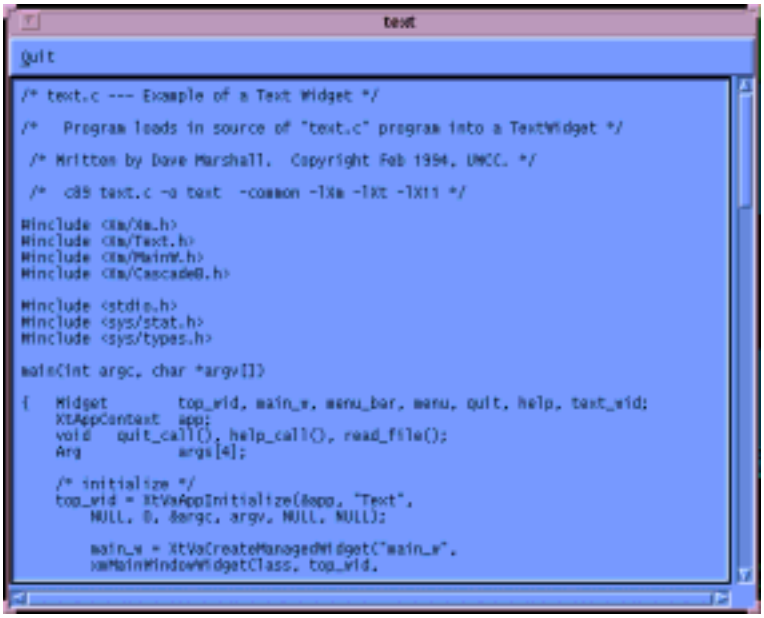
Le petit programme ***text.c*** présente un exemple d'utilisation de cette fonction.

8.4. Le programme *text.c*.

Ce programme est un petit exemple d'utilisation de Text widget. Il crée un widget de type ***ScrolledText*** et affiche à l'intérieur son propre source C : le contenu du programme ***text.c***.

L'édition n'est pas autorisée (lors de la création, la ressource ***XmNeditable*** vaut ***False***).

Voici un dump d'écran de l'exécution de ce petit programme:



Remarque : pour compiler sans erreur il faut inclure le fichier ***<Xm/Text.h>***.

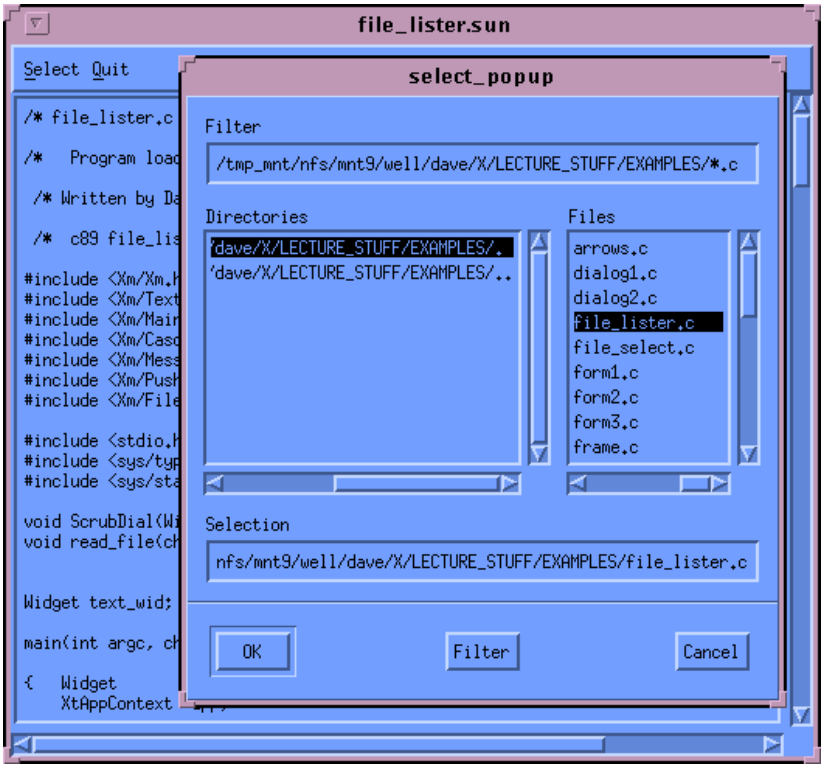
```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/RowColumn.h>

/*--------------------------*/
main(int argc, char *argv[])
/*--------------------------*/
{
  Widget       top_wid, main_w, menu_bar,
               menu, quit, help, text_wid;
  XtAppContext  app;
  void         quit_call(), help_call(), read_file();
  Arg          args[10];
  int n;

  /* initialize */
  top_wid = XtVaAppInitialize(&app, "Text",
                  NULL, 0, &argc, argv, NULL, NULL);


  main_w = XtVaCreateManagedWidget("main_w",
                  xmMainWindowWidgetClass, top_wid,
                  /* XmNscrollingPolicy,   XmVARIABLE, */
                  NULL);

  menu_bar = XmCreateMenuBar(main_w, "main_list",
                  NULL, 0);
  XtManageChild(menu_bar);

  /* create quit widget + callback */
```

```
  quit = XtVaCreateManagedWidget( "Quit",
                    xmCascadeButtonWidgetClass, menu_bar,
                    XmNmnemonic, 'Q',
                    NULL);

  XtAddCallback(quit, XmNactivateCallback,
              quit_call, NULL);

  /* Create ScrolledText -- this is work area for the
     MainWindow */

  n=0;
  XtSetArg(args[n], XmNrows,      30); n++;
  XtSetArg(args[n], XmNcolumns,   80); n++;
  XtSetArg(args[n], XmNeditable,  False); n++;
  XtSetArg(args[n], XmNeditMode,  XmMULTI_LINE_EDIT); n++;
  text_wid = XmCreateScrolledText(main_w, "text_wid",
                              args, n);
  XtManageChild(text_wid);

  /*  read file and put data in text widget */
  read_file(text_wid);

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*----------------------------*/
void read_file(Widget  text_wid)
/*----------------------------*/
{
  static char *filename = "text.c";
  char  *text;
  struct stat statb;
  FILE *fp;

  /* check file is a regular text file and open it */

  if ((stat(filename, &statb) == -1)
      || !(fp = fopen(filename, "r"))) {
    fprintf(stderr, "Cannot open file: %s\n", filename);
    XtFree(filename);
    return;
  }

  /* Map file text in the TextWidget */

  if (!(text = XtMalloc((unsigned)(statb.st_size+1)))) {
    fprintf(stderr, "Can't alloc enough space for %s",
            filename);
    XtFree(filename);
    fclose(fp);
    return;
  }


  if (!fread(text, sizeof(char), statb.st_size+1, fp))
    fprintf(stderr, "File read error\n");

  text[statb.st_size] = 0; /* be sure to NULL-terminate */

  /* insert file contents in TextWidget */
  XmTextSetString(text_wid, text);

  /* free memory
   */
  XtFree(text);
  XtFree(filename);
  fclose(fp);
}

/*--------------*/
void quit_call()
/*--------------*/
```

```
{
  printf("Quitting program\n");
  exit(0);
}
```

EXERCICE : à partir du programme text.c, écrire le programme ***file_lister.c*** permettant à l'aide d'un FileSelectionDialog de lister dans un ScrolledText widget le contenu de n'importe quel programme source C.

Dump d'écran du programme (à écrire) file_lister.c :



8.5. Edition de texte dans un Text Widget

8.5.1. Remplacement de texte :

La fonction ***XmTextReplace***() permet de remplacer tout ou partie du texte qui se trouve dans un TextWidget.

Paramètres de cette fonction :

- ***Widget***: L'ID du Text widget.
- ***StartPosition*** (de type ***XmTextPosition***): Un long int (défini dans <Xm/Text.h>) mesuré en bytes. Indique le point d'insertion du texte.

- ***EndPosition*** (de type ***XmTextPosition***): Même type que le paramètre précédent.
- NewText: La chaîne de caractères (au sens du langage C) qui va remplacer le texte situé entre les deux positions ***StartPosition*** et ***EndPosition***.

Quelques remarques :

Si ***StartPosition*** est égal à ***EndPosition*** alors le texte est inséré juste après le point d'insertion.

Le texte est remplacé uniquement entre les deux positions spécifiées par ***StartPosition*** et ***EndPosition***.

8.5.2. Insertion de texte :

Pour insèrer du texte, il faut utiliser la fonction ***XmTextInsert***().
Elle accepte trois paramètres :

- ***Widget***: L'ID du Text widget.
- ***InsertPosition*** (de type ***XmTextPosition***): Un ***long int*** (défini dans <***Xm/Text.h***>) mesuré en bytes. Indique le point d'insertion du texte.
- ***String***: La chaîne de caractères (au sens du langage C) à insèrer.

8.5.3. Rechercher un texte.

Pour rechercher un texte dans un Text widget, il faut utiliser la fonction ***XmTextFindString***().

Cette fonction possède les paramètres suivants :

- ***Widget***: L'ID du Text widget.
- ***StartPosition*** (de type ***XmTextPosition***): Un ***long int*** (défini dans <***Xm/Text.h***> mesuré en bytes. Indique la position dans le texte à partir de laquelle la recherche va s'effectuer.
- ***String***: La chaîne de caractères que l'on recherche.
- ***SearchDirection***: Sens de la recherche. Valeurs possibles : ***XmTEXT_FORWARD*** ou ***XmTEXT_BACKWARD***.
- ***Position***: Un pointeur de type ***XmTextPosition*** retourné par la fonction de recherche. Indique à quelle position la chaîne de caractères a été trouvée.

XmTextFindString() renvoie une valeur de type ***Boolean*** : ***True*** si la chaîne a été trouvée dans le texte, ***False*** sinon.

8.5.4. Sauvegarder le contenu d'un Text widget dans un fichier :

La fonction ***XmTextGetString***() permet d'obtenir le contenu d'un Text widget. Ainsi, pour sauvegarder le contenu d'un Text widget dans un fichier, il suffit simplement de :

- Appeler XmTextGetString()
- fprintf() le texte dans un fichier.

8.5.5. Contrôle de l'affichage du texte :

Pour afficher le texte à une position donnée, il faut utiliser la fonction ***XmTextShowPosition***().

Pour positionner une ligne du texte en haut de la fenêtre du ScrolledText widget, il faut utiliser la fonction ***XmTextSetTopCharacter***().

Pour positionner le point d'insertion (le curseur) à une position donnée, il faut utiliser la fonction ***XmTextSetInsertionPosition***().

Par défaut, un Text widget contient des fonctions d'édition de texte très simples. Si l'on désire un meilleur contrôle sur l'édition, il faut utiliser des fonctions de callback.

MOTIF fournit de nombreuses ressources pour les callbacks des Text widgets.

Voici les principales :

- ***XmNactivateCallback***: Uniquement pour les TextField ou les Text widgets d'une seule ligne. Ce callback est appelé lorsque l'utilisateur appuie sur la touche Enter.
- ***XmNverifyCallback***: Appelé avant un changement dans le texte.
- ***XmNvalueChangedCallback***: Appelé après un changement dans le texte.
- ***XmNmotionCallback***: Appelé lorsque l'utilisateur a déplacé le curseur ou bien lorsqu'il a effectué une sélection à la souris.
- ***XmNfocusCallback***: Appelé lorsque le text widget obtient le focus du clavier.
- ***XmNfocusCallback***: Appelé lorsque le text widget perd le focus du clavier.

Les verifyCallbacks peuvent être utilisés pour la saisie de mots de passe par exemple…

9. Les widgets de type DrawingArea.

9.1. Introduction :

Le widget de type DrawingArea permet de dessiner à l'aide des fonctions graphique de la Xlib que nous avons étudiées.

Pour créer une DrawingArea on utilisera au choix XmCreateDrawingArea() ou ***XtVaCreateManagedWidget***() avec comme classe ***XmDrawingAreaWidgetClass***.

Pour compiler sans erreurs il faut inclure le fichier <***Xm/DrawingA.h***>.

Il existe aussi un widget de type ***DrawnButton*** qui est la combinaison d'une DrawingArea et d'un PushButton. Si on utilise des DrawnButton dans une application, il faut inclure <***Xm/DrawnB.h***>.

9.2. Ressources et callbacks des widget de type DrawingArea.

En général on place les DrawingArea à l'intérieur d'un container widget : une Form, une MainWindow, une ScrolledWindow, etc... si bien qu'il est rare que l'on doive spécifier sa taille car cette dernière est héritée du widget père. On peut cependant positionner les ressources ***XmNheight*** et ***XmNwidth*** pour fixer la taille.

La ressource ***XmNresizePolicy*** permet de règler les autorisations de changement de taille par le window manager. Trois valeurs possibles : ***XmRESIZE_ANY***, ***XmRESIZE_GROW*** et ***XmRESIZE_NONE***.

Par défaut on peut associer trois types de callbacks à une DrawingArea :

- ***XmNexposeCallback***: Ce callback est appelé lorsqu'une partie de la DrawingArea doit être redessinnée.
- ***XmNresizeCallback***: Si la taille de la DrawingArea change ce callback est appelé.
- ***XmNinputCallback***: Si une entrée clavier ou un click souris est effectué dans la DrawingArea, ce callback est appelé.

9.3. Utilisation pratique d'une DrawingArea.

Nous allons maintenant réaliser du dessin 2D dans une application MOTIF.

Toutes les opérations graphiques seront réalisées à l'aide de fonctions de la Xlib, il va falloir récurpèrer des informations de bas niveau à partir des widgets MOTIF possèdant un de haut niveau d'abstraction. Nous allons étudier de petits programmes d'exemples. Tous procèdent de la même manière:

- 1.Création des différents widgets de l'application, y compris bien sûr une DrawingArea.
- 2.Obtention d'informations nécessaires aux fonctions Xlib : Display, pointeurs sur l'ID de la fenêtre X de la DrawingArea, etc…
- 3.Dessin dans la fenêtre X.

9.3.1. Premier programme d'exemple: *draw.c*.

Ce petit programme illustre les principes de base de l'utilisation conjointe de fonctions Xlib, Xt et MOTIF.

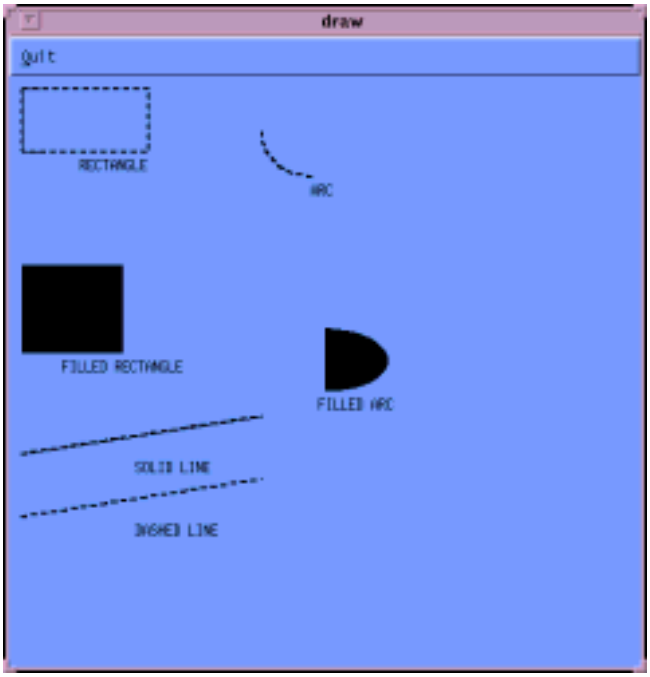
Il crée une DrawingArea, draw, dans une MainWindow. Les IDs du Display et du Screen (au sens Xlib) sont obtenus à l'aide des fonctions ***XtDisplay***() et ***XtScreen***().

Le contexte graphique est crée avec comme une couleur de foreground noire, obtenue à l'aide de la macro Xlib BlackPixelOfScreen() (remarque : on aurait pu aussi utiliser BlackPixel(), l'une prend un screen en argument, l'autre un display).

Le contexte graphique est placé dan la ressource XmNuserData de la DrawingArea, qui sert à passer des informations de la même manière que le paramètre ***ClientData*** des fonctions de callback.

Pour récupèrer la valeur de la ressource XmNuserData dans la fonction

de callback, on utilisera ***XtGetValues***().




```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>
#include <Xm/RowColumn.h>
/* XLIB Data */

Display *display;
Screen *screen_ptr;

/*--------------------------*/
main(int argc, char *argv[])
/*--------------------------*/
{
  Widget top_wid, main_w, menu_bar, draw, quit;
  XtAppContext app;
  XGCValues gcv;
  GC gc;
  void quit_call(), draw_cbk();

  top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
                        &argc, argv, NULL,
                        XmNwidth,  500,
                        XmNheight, 500,
                        NULL);

  /* create a main window */
  main_w = XtVaCreateManagedWidget("main_window",
                           xmMainWindowWidgetClass, top_wid,
                           NULL);

  /* create a menu bar */
  menu_bar = XmCreateMenuBar(main_w, "main_list",
                       NULL, 0);
  XtManageChild(menu_bar);

  /* create quit widget in menu bar + callback */
```

```
  quit = XtVaCreateManagedWidget( “Quit”,
                          xmCascadeButtonWidgetClass, menu_bar,
                          XmNmnemonic, ‘Q’,
                          NULL);

  XtAddCallback(quit, XmNactivateCallback, quit_call,
            NULL);

  /* Create a DrawingArea widget. */
  draw = XtVaCreateWidget(“draw”,
                    xmDrawingAreaWidgetClass, main_w,
                    NULL);

  /* get XLib Display Screen and Window ID’s for draw */
  display = XtDisplay(draw);
  screen_ptr = XtScreen(draw);


  /* set the MenuBar and the DrawingArea as menu bar and the
     “work area” of main window. Like that they are partly managed
     by the MainWindow */
  XtVaSetValues(main_w,
            XmNmenuBar,    menu_bar,
            XmNworkWindow, draw,
            NULL);

  /* add callback for exposure event */
  XtAddCallback(draw, XmNexposeCallback, draw_cbk, NULL);


  /* Create a GC. Attach GC to the DrawingArea’s
     XmNuserData.
     NOTE : This is a useful method to pass data */
  gcv.foreground = BlackPixelOfScreen(screen_ptr);
  gc = XCreateGC(display,
              RootWindowOfScreen(screen_ptr), GCForeground, &gcv);
  XtVaSetValues(draw, XmNuserData, gc, NULL);

  /* because we changed some ressources after the Managed Creation */
  XtManageChild(draw);

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/* CALL BACKS */

/*--------------*/
void quit_call()
/*--------------*/
{
  printf(“Quitting program\n”);
  exit(0);
}

/*-------------------------------------------------*/
void draw_cbk(Widget w, XtPointer data,
            XmDrawingAreaCallbackStruct *cbk)
/*-------------------------------------------------*/
/*  DrawingArea Callback. NOTE: cbk->reason says type of
    callback event */
{
  char  str1[25];
  int len1, width1, font_height;
  unsigned int width, height;
  int  x, y, angle1, angle2, x_end, y_end;
  unsigned int line_width = 1;
  int line_style = LineSolid;
  int cap_style = CapRound;
  int join_style = JoinRound;
  XFontStruct *font_info;
  XEvent *event = cbk->event;
  void load_font();
  GC gc;
```

```
Window win = XtWindow(w);


if (cbk->reason != XmCR_EXPOSE) {
  /* Should NEVER HAPPEN for this program */
  printf("X is screwed up!!\n");
  exit(0);
}

/* get font info */
load_font(&font_info);
font_height = font_info->ascent + font_info->descent;


/* get gc from Drawing Area user data */
XtVaGetValues(w, XmNuserData, &gc, NULL);

/* Draw A Rectangle */
x = y = 10;
width = 100;
height = 50;

XDrawRectangle(display, win, gc, x, y, width, height);

strcpy(str1,"RECTANGLE");
len1 = strlen(str1);
y += height + font_height + 1;

if ((x = (x + width/2) - len1/2) < 0 )
  x = 10;

XDrawString(display, win, gc, x, y, str1, len1);

/* Draw a filled rectangle */
x = 10; y = 150;
width = 80;
height = 70;

XFillRectangle(display, win, gc, x, y, width, height);

strcpy(str1,"FILLED RECTANGLE");
len1 = strlen(str1);
y += height + font_height + 1;
if ((x = (x + width/2) - len1/2) < 0 )
  x = 10;

XDrawString(display, win, gc, x, y, str1, len1);


/* draw an arc */
x = 200; y = 10;
width = 80;
height = 70;
angle1 = 180 * 64; /* 180 degrees */
angle2 = 90 * 64;  /* 90 degrees */

XDrawArc(display, win, gc, x, y, width, height,
        angle1, angle2);


strcpy(str1,"ARC");
len1 = strlen(str1);
y += height + font_height + 1;
if ((x = (x + width/2) - len1/2) < 0 )
  x = 200;

XDrawString(display, win, gc, x, y, str1, len1);


/* draw a filled arc */
x = 200; y = 200;
width = 100;
height = 50;
angle1 = 270 * 64; /* 270 degrees */
```

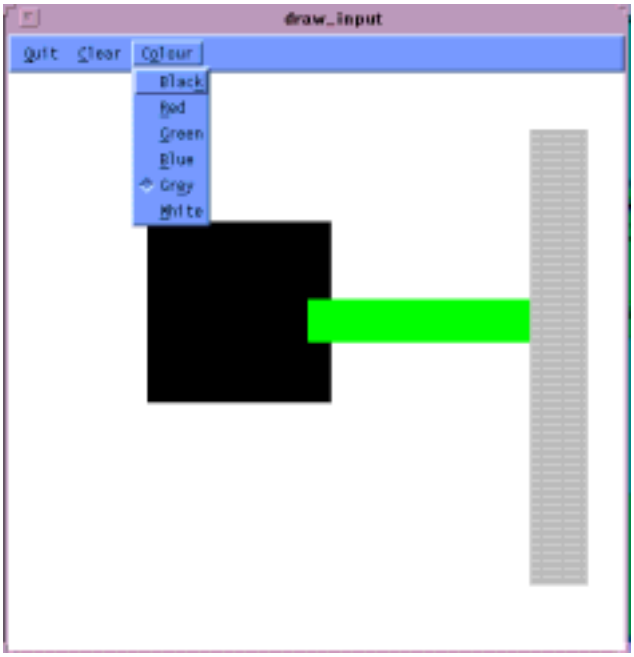
```
  angle2 = 180 * 64; /* 180 degrees */

  XFillArc(display, win, gc, x, y, width, height,
        angle1, angle2);

  strcpy(str1,"FILLED ARC");
  len1 = strlen(str1);
  y += height + font_height + 1;
  if ((x = (x + width/2) - len1/2) < 0 )
    x = 200;

  XDrawString(display, win, gc, x, y, str1, len1);


  /* SOLID LINE */
  x = 10; y = 300;
  /* start and end points of line */
  x_end = 200; y_end = y - 30;
  XDrawLine(display, win, gc, x, y, x_end, y_end);

  strcpy(str1,"SOLID LINE");
  len1 = strlen(str1);
  y += font_height + 1;
  if ((x = (x + x_end)/2 - len1/2) < 0 )
    x = 10;

  XDrawString(display, win, gc, x, y, str1, len1);


  /* DASHED LINE */
  line_style = LineOnOffDash;
  line_width = 2;

  /* set line attributes */
  XSetLineAttributes(display, gc, line_width, line_style,
                cap_style, join_style);

  x = 10; y = 350;
  /* start and end points of line */
  x_end = 200; y_end = y - 30;
  XDrawLine(display, win, gc, x, y, x_end, y_end);

  strcpy(str1,"DASHED LINE");
  len1 = strlen(str1);
  y += font_height + 1;
  if ((x = (x + x_end)/2 - len1/2) < 0 ) x = 10;

  XDrawString(display, win, gc, x, y, str1, len1);
}

/*--------------------------------------*/
void load_font(XFontStruct **font_info)
/*--------------------------------------*/
{
  char *fontname = "fixed";
  XFontStruct *XLoadQueryFont();

  /* load and get font info structure  */

  if (( *font_info = XLoadQueryFont(display, fontname)) == NULL) {
    /* error - quit early */
    printf("%s: Cannot load  %s font\n", "draw.c",
          fontname);
    exit(-1);
  }
}
```

9.3.2. Deuxième exemple : le programme draw_input.c

Avec cet exemple, nous allons voir comment traiter les entrées/sorties dans une DrawingArea.

Nous reprenons le programme menu_pull.c déjà étudié lors du cours sur les MainWindow et les menus, et le modifions pour intégrer une DrawingArea. Ce programme permet le dessin de rectangles "à la Mac Draw", le menu "colors" servant à sélectionner la couleur du rectangle que l'on veut dessiner.

Dump d'écran de draw_input.c:



Nous avons déjà vu pendant les TDs Xlib comment il fallait procèder pour dessiner des rectangles "élastiques". Il faut traiter trois types d'événements : l'appui d'un bouton de la souris, le déplacement de la souris et le relâchage du bouton souris.

Les DrawingArea widgets possèdent deux ressources permettant d'appeler des callbacks en cas d'entrées clavier ou souris, mais ils ne possèdent pas de ressources permettant de traiter le déplacement de la souris.

La solution consiste à enrichir la table de translations de la DrawingArea.

Rappel: chaque widget possède une table de translations qui se compose de :

- Une liste d'événements susceptible de se produire dans le widget.
- Des actions associées à ces événements.

Extrait d'une table de translation (celle de ***netscape***, prise dans son fichier de ressources ***/usr1/local/lib/X11/app-defaults/Netscape***) :

```
        .
  <Btn1Down>:     ArmLink()
  <Btn2Down>:     ArmLink()
  .
```

Pour ajouter le traitement de l'événement "déplacement de la souris bouton 1 appuyé", il faut ajouter la translations suivante :

```
  <Btn1Motion>: draw_cbk(motion)
```

où ***draw_cbk***() est la fonction de callback qui effectue le dessin des rectangles. Chaque fois que la souris sera déplacée bouton 1 appuyé dans la DrawingArea, ***draw_cbk***() sera appelée. De plus, il est possible de passer à la fonction de callback des paramètres de type String! Avec l'exemple ci-dessus, la chaîne de caractères "motion" sera passée (plus de détails sur le traitement de ces paramètres seront donnés page suivante).

Bien que l'on puisse traiter les autres événements souris avec les méthodes d'ajout de callback standards (***XtAddCallBack***(), etc...), nous allons nous servir de la table de translation pour traiter l'appui et le relâchement du bouton 1 de la souris. La table de translation finale ressemble à ceci :

```
  <Btn1Motion>: draw_cbk(motion)
  <Btn1Up>: draw_cbk(up)
  <Btn1Down>: draw_cbk(down)
```

En pratique, dans un programme MOTIF, nous initialisons d'abord la table de translations dans une variable de type ***String***, puis l'on attache cette table à la DrawingArea en positionnant la ressource ***XmNtranslations*** avec comme valeur le résultat de ***XtParseTranslationTable***(***String translations***).

Il faut également, lors de la création de la DrawingArea, indiquer que nous avons enrichi la table de translations par défaut avec les actions associées aux nouvelles translations. Ceci est réalisé à l'aide de la fonction ***XtAppAddActions***(). Ainsi il sera possible de postionner ces nouvelles translations dans un fichier de ressources, l'application reconnaîtra les nouveaux événements.

Avec notre exemple :

```
XtActionsRec actions; ... actions.string = "draw_cbk"; actions.proc = draw_cbk;
XtAppAddActions(app, &actions, 1);
```

Voici l'en-tête de la fonction de callback:

```
void  draw_cbk(Widget w, XButtonEvent *event,
          String *args, int *num_args)
{
  .
  .
  .
}
```

Les paramètres sont passés dans la variable ***args***, le nombre de paramètres dans ***num_args***. Comme nous ne passons qu'un seul paramètre avec la table de translations correspondant à cet exemple, nous testerons juste la valeur de la variable ***args[0]***. Les différentes valeurs ***up***, ***motion*** et ***down*** nous renseigneront sur la nature de l'événement qui a déclenché l'appel de la fonction de callback.

Listing complet du programme draw_input.c :

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>
#include <Xm/RowColumn.h>

GC gc;
XGCValues gcv;
Widget draw;
String colours[] = { "Black",  "Red", "Green", "Blue",
                "Grey", "White"};
long int fill_pixel = 1; /* stores current colour
                    of fill - black default */
Display *display; /* xlib id of display */
Colormap cmap;

/*----------------------------*/
main(int argc, char *argv[])
/*----------------------------*/
{
  Widget top_wid, main_w, menu_bar, quit, clear, colour;
  XtAppContext app;
  XmString  quits, clears, colourss, red, green,
          blue, black, grey, white;
  void quit_call(), clear_call(), colour_call(),
      draw_cbk();

  XtActionsRec actions;
  String translations =
    "<Btn1Motion>: draw_cbk(motion)  \n\
     <Btn1Down>: draw_cbk(down)  \n\
     <Btn1Up>: draw_cbk(up) ";

  top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
                        &argc, argv, NULL,
                        XmNwidth,  500,
                        XmNheight, 500,
                        NULL);

  /* Create MainWindow */
  main_w = XtVaCreateManagedWidget("main_window",
                          xmMainWindowWidgetClass,   top_wid,
                          XmNwidth, 500,
                          XmNheight, 500,
                          NULL);

  /* Create a simple MenuBar that contains three menus */
  quits = XmStringCreateSimple("Quit");
  clears = XmStringCreateSimple("Clear");
  colourss = XmStringCreateSimple("Colour");


  menu_bar = XmVaCreateSimpleMenuBar(main_w, "main_list",
                            XmVaCASCADEBUTTON, quits, 'Q',
                            XmVaCASCADEBUTTON, clears, 'C',
                            XmVaCASCADEBUTTON, colourss, 'o',
                            NULL);
  XtManageChild(menu_bar);
```

```
  /* First menu is quit menu -- callback is quit_call() */
  XmVaCreateSimplePulldownMenu(menu_bar, "quit_menu", 0,
        quit_call, XmVaPUSHBUTTON, quits, 'Q', NULL, NULL,
        NULL);
  XmStringFree(quits);

  /* Second menu is clear menu -- callback is clear_call() */

  XmVaCreateSimplePulldownMenu(menu_bar, "clear_menu", 1,
        clear_call, XmVaPUSHBUTTON, clears, 'C', NULL, NULL,
        NULL);
  XmStringFree(clears);


  /* create colour pull down menu */

  black = XmStringCreateSimple(colours[0]);
  red = XmStringCreateSimple(colours[1]);
  green = XmStringCreateSimple(colours[2]);
  blue = XmStringCreateSimple(colours[3]);
  grey = XmStringCreateSimple(colours[4]);
  white = XmStringCreateSimple(colours[5]);


  colour = XmVaCreateSimplePulldownMenu(menu_bar,
            "edit_menu", 2, colour_call,
            XmVaRADIOBUTTON, black, 'k', NULL, NULL,
            XmVaRADIOBUTTON, red, 'R', NULL, NULL,
            XmVaRADIOBUTTON, green, 'G', NULL, NULL,
            XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
            XmVaRADIOBUTTON, grey, 'e', NULL, NULL,
            XmVaRADIOBUTTON, white, 'W', NULL, NULL,
            XmNradioBehavior, True,
            /* RowColumn resources to enforce */
            XmNradioAlwaysOne, True,
            /* radio behavior in Menu */
            NULL);

  XmStringFree(black);
  XmStringFree(red);
  XmStringFree(green);
  XmStringFree(blue);
  XmStringFree(grey);
  XmStringFree(white);


  /* Create a DrawingArea widget. */
  /* make new actions */
  actions.string = "draw_cbk";
  actions.proc = draw_cbk;
  XtAppAddActions(app, &actions, 1);

  draw = XtVaCreateWidget("draw",
          xmDrawingAreaWidgetClass, main_w,
          XmNtranslations, XtParseTranslationTable(translations),
          XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
          NULL);


  cmap = DefaultColormapOfScreen(XtScreen(draw));
  display = XtDisplay(draw);

/* set the DrawingArea as the "work area" of main window */
  XtVaSetValues(main_w,
             XmNmenuBar,    menu_bar,
             XmNworkWindow, draw,
             NULL);


/* Create a GC. Attach GC to DrawingArea's XmNuserData. */
  gcv.foreground = BlackPixelOfScreen(XtScreen(draw));
  gc = XCreateGC(XtDisplay(draw),
```

```
                RootWindowOfScreen(XtScreen(draw)),
                GCForeground, &gcv);

  XtManageChild(draw);
  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}


/* CALL BACKS */

/*-------------*/
void quit_call()
/*-------------*/
{
  printf("Quitting program\n");
  exit(0);
}

/*---------------*/
void clear_call()
/*---------------*/
/* clear work area */
{
  XClearWindow(display, XtWindow(draw));
}

/*---------------------------*/
void colour_call(w, item_no)
/*---------------------------*/
/* called from any of the "Colour" menu items.
   Change the color of the
   label widget.
   Note: we have to use dynamic setting with setargs()..
 */
Widget w;     /* menu item that was selected */
int item_no;  /* the index into the menu */
{
  int n =0;
  Arg args[1];

  XColor xcolour, spare; /* xlib color struct */


  if (XAllocNamedColor(display, cmap, colours[item_no],
                  &xcolour, &spare) == 0)
    return;

  /* remember new colour */
  fill_pixel = xcolour.pixel;
}

/*  DrawingArea Callback.*/

/*-----------------------------------------------*/
void  draw_cbk(Widget w, XButtonEvent *event,
            String *args, int *num_args)
/*-----------------------------------------------*/
{
  static Position x, y, last_x, last_y;
  Position width, height;

  int line_style;
  unsigned int line_width = 1;
  int cap_style = CapRound;
  int join_style = JoinRound;

  if (strcmp(args[0], "down") == 0) {
    /* anchor initial point (save its value) */
    x = event->x;
    y = event->y;
  }
  else
    if (strcmp(args[0], "motion") == 0) {
```

```
    /* draw “ghost” box to show where it could go */
    /* undraw last box */

    line_style = LineOnOffDash;
    /* set line attributes */

    XSetLineAttributes(event->display, gc,
         line_width, line_style, cap_style, join_style);

    gcv.foreground
      = WhitePixelOfScreen(XtScreen(w));

    XSetForeground(event->display, gc,
                gcv.foreground);

    XSetFunction(event->display, gc, GXinvert);


    XDrawLine(event->display, event->window, gc,
            x, y, last_x, y);
    XDrawLine(event->display, event->window, gc,
            last_x, y, last_x, last_y);
    XDrawLine(event->display, event->window, gc,
            last_x, last_y, x, last_y);
    XDrawLine(event->display, event->window, gc,
            x, last_y, x, y);


    /* Draw New Box */
    gcv.foreground
      = BlackPixelOfScreen(XtScreen(w));
    XSetForeground(event->display, gc,
                gcv.foreground);

    XDrawLine(event->display, event->window, gc,
            x, y, event->x, y);
    XDrawLine(event->display, event->window, gc,
            event->x, y, event->x, event->y);
    XDrawLine(event->display, event->window, gc,
            event->x, event->y, x, event->y);
    XDrawLine(event->display, event->window, gc,
            x, event->y, x, y);
  }
  else
    if (strcmp(args[0], “up”) == 0) {
      /* draw full line */

      XSetFunction(event->display, gc, GXcopy);

      line_style = LineSolid;

      /* set line attributes */

      XSetLineAttributes(event->display, gc,
      line_width, line_style, cap_style, join_style);

      XSetForeground(event->display, gc, fill_pixel);


      XDrawLine(event->display, event->window, gc,
              x, y, event->x, y);
      XDrawLine(event->display, event->window, gc,
              event->x, y, event->x, event->y);
      XDrawLine(event->display, event->window, gc,
              event->x, event->y, x, event->y);
      XDrawLine(event->display, event->window, gc,
              x, event->y, x, y);

      width = event->x - x;
      height = event->y - y;
      XFillRectangle(event->display, event->window,
                  gc, x, y, width, height);
    }
  last_x = event->x;
```

```
  last_y = event->y;
}
```

10. Les widgets de type List.

10.1. Introduction.

Les widgets de type List permettent à l'utilisateur de choisir un ou plusieurs éléments appartenant à une liste d'éléments!

Nous avons déjà rencontré inconsciemment des widgets de type List, par exemple lors de l'étude des FileSelectionDialog. Les deux boîtes contenant les noms des répertoires et les noms des fichiers d'un FileSelectionDialog sont des widgets de type List.

La manipulation des List est similaire à celle des PulldownMenus. Cependant, avec un widget de type List :

- Il est possible d'ajouter des éléments dans la liste et aussi d'en enlever.
- Les List widgets peuvent être "scrollés".
- Il existe plusieurs modes de sélection des éléments.

Voici un exemple de widget de type List:

10.2. Modes de sélection des éléments d'une liste.

Quatre modes de sélection sont disponibles, on choisit l'un des modes en positionnant la ressource ***XmNselectionPolicy*** avec l'une de ces valeurs :

- ***XmSINGLE_SELECT***: Un seul item de la liste peut être sélectionné.
- ***XmBROWSE_SELECT***: Similaire à ***XmSINGLE_SELECT*** sauf qu'une valeur par défaut peut être choisie. L'interaction avec l'utilisateur est par conséquent légèrement différente.
- ***XmMULTIPLE_SELECT***: Plusieurs sélections sont possibles. Un élément est choisi en clickant dessus. Si on clicke à nouveau sur un élément déjà choisi, on annule sa sélection.
- ***XmEXTENDED_SELECT***: Similaire au mode précédent sauf que l'on peut sélectionner plusieurs éléments contigus en clickant avec la souris sur un élément puis en faisant glisser la souris sur d'autres éléments en maintenant le bouton enfoncé.

10.3. Principes de base d'utilisation des List widget.

10.3.1. Création widgets de type List ou ScrolledList.

Pour créer un List widget on utilisera les fonctions habituelles ***XmCreateList***() ou ***XtVaCreateManagedWidget***() avec comme classe ***xmListWidgetClass***.

Le plus souvent on désire créer des listes possèdant des barres de scrolling. On utilisera alors la fonction ***XmCreateScrolledList***().

10.3.2. Principales ressources.

- ***XmNitemCount***: Nombre d'éléments dans la liste.
- ***XmNitems***: La liste des éléments. Elle doit être de type ***XmStringTable***, qui est un type correspondant à un tableau unidimmensionnel d'objets de type ***XmString***.
- ***XmNselectionPolicy***: Mode de sélection des éléments. Voir plus haut.
- ***XmNvisibleItemCount***: Nombre d'éléments visibles dans la fenêtre du widget de type List. Cette ressource détermine en fait la hauteur du widget.
- ***XmNdisplayScrollBarPolicy***: Cette ressource permet de sélectionner l'apparition de la barre de scrolling. Deux valeurs sont possibles :
  - 1.***XmSTATIC*** : une barre de scrolling verticale sera toujours présente sur le bord de la boîte contenant les éléments de la liste.
  - 2.***XmAS_NEEDED*** : la barre de scrolling ne sera affichée que si tous les éléments ne contiennent pas dans la boîte d'affichage.
- ***XmNlistSizePolicy***: Cette ressource permet de sélectionner les différents modes de retaillage du widget de type List. Trois valeurs sont possibles :
  - 1.***XmCONSTANT*** : la taille est constante.
  - 2.***XmRESIZE_IF_POSSIBLE*** : la taille peut varier si possible, c'est-à-dire si le widget parent le permet.
  - 3.***XmVARIABLE***.
- ***XmNselectedItemCount***: Nombre d'éléments séléctionnés par défaut.

- ***XmNselectedItems***: La liste des éléments séléctionnés.

10.3.3. Ajout et retrait d'éléments:

Pour ajouter un élément dans un List widget, on utilise la fonction MOTIF ***XmListAddItem***(), qui accepte trois arguments :

- Un pointeur sur le widget de type List auquel on veut ajouter un élément.
- L'élément, de type ***XmString***.
- La position où l'on veut insérer l'élément. Il s'agit d'un entier (un int). Cette position commence avec la valeur 1, la position 0 servant à désigner la dernière position dans la liste.

Une autre fonction, ***XmListAddItemUnselected***() possède exactement la même syntaxe. Elle garantit qu`un élément n'est jamais séléctionné lorsqu'il est ajouté à la liste, ce qui n'est pas toujours le cas avec la fonction ***XmListAddItem***().

Pour supprimer un seul élément d'une liste il faut utiliser la fonction ***XmListDeleteItem***(***Widget***, ***XmString***).

Pour supprimer plusieurs éléments d'une liste il faut utiliser la fonction ***XmListDeleteItems***(***Widget***, ***XmString****).

Si la position du ou des éléments(s) à supprimer est connue, on peut utiliser également la fonction ***XmListDeletePos***(***Widget w***, ***int num***, ***int Pos***). Elle permet de détruire num éléments à partir de la position pos.

Pour détruire tous les éléments d'une liste, on utilisera la fonction ***XmListDeleteAllItems***(***Widget***).

10.3.4. Sélection d'éléments dans une liste.

Pour sélectionner des éléments dans une liste on utilisera les fonctions ***XmListSelectItem***(***Widget***, ***XmString***, ***Boolean***) et ***XmListSelectPos***(***Widget***, ***int***, ***Boolean***).

Si la valeur de type ***Boolean*** est égale à ***True***, alors la fonction de callback de la liste sera appelée lors de la sélection (Les fonctions de callback sont détaillées dans la section suivante).

Pour désélectionner des éléments d'une liste, on utilisera une des fonctions : ***XmListDeselectItem***(***Widget***, ***XmString***), ***XmListDeselectPos***(***Widget***, ***int***) ou ***XmListDeselectAllItems***(***Widget***).

10.3.5. Obtenir des informations sur une liste.

Etant donné que les widgets de type List évoluent dynamiquement (on ajoute et on enlève des éléments de la liste pendant l'exécution de l'application), il peut être nécessaire de connaître à un moment donné la valeur de certains attributs de la liste, par exemple le nombre d'éléments se trouvant dans la liste, l'ensemble des éléments sélectionnés, etc...

On obtient toutes ces informations à l'aide de la fonction ***XtGetValues***(), déjà étudiée, qui permet de consulter la valeur des ressources d'un widget. En effet, les ressourrrces des List widgets sont mises à jour dynamiquement pendant l'exécution de l'application.

10.3.6. Les fonctions de callbacks :

Il existe une ressource de callback pour chaque mode de sélection :***XmNsingleSelectionCallback***, ***XmNmultipleSelectionCallback***, ***XmNbrowseSelectionCallback*** et ***XmNextendedSelectionCallback***; et une ressource de callback par default : ***XmNdefaultActionCallback***.

La fonction de callback par défaut est toujours appelée après la fonction de callback correspondant à la sélection d'un ou plusieurs éléments.

Le callback de sélection est déclenché lors d'un double click sur un élément.

Les fonctions de callback pour les List widgets ont la forme suivante :

```
list_cbk(Widget w, XtPointerData data, XmListCallbackStruct *cbk)
```

Les champs intéressants de la structure ***XmListCallbackStruct*** sont les suivants :

- item : l'élément sélectionné (cas d'une sélection simple), de type XmString.
- item_position : la position de l'élément sélectionné dans la liste.
- selected_items : liste des éléments sélectionnés (cas d'une sélection multiple), il s'agit d'un tableau de XmString.
- selected_item_count : nombre d'éléments sélectionnés (taille du tableau selected_items).
- selected_item_positions : tableau contenant les positions des éléments sélectionnés.

Vous pouvez vous référer à la doc MOTIF ou bien regarder le contenu de <***XM/List.h***> pour obtenir plus d'informations.

10.4. Petit programme d'exemple : list.c

Ce petit programme crée une liste de noms de couleurs. La sélection des différents éléments change la couleur de fond du List widget, en utilisant une méthode similaire à celle déjà étudiée avec le programme ***menu_pull.c***

Voici à quoi ressemble l'exécution de ce petit programme:

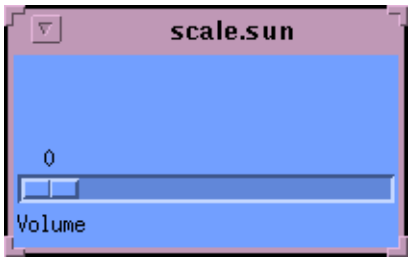

```
#include <Xm/Xm.h>
#include <Xm/List.h>

String colours[] = { "Black",  "Red", "Green",
                "Blue", "Grey"};

Display *display; /* xlib id of display */
Colormap cmap;

/*---------------*/
main(argc, argv)
/*---------------*/
char *argv[];
{
  Widget          top_wid, list;
  XtAppContext    app;
  int             i, nb_items = XtNumber(colours);
  XColor          back, fore, spare;
  XmStringTable   str_list;
  int             n = 0;
  Arg             args[10];
  void            list_cbk();

  top_wid = XtVaAppInitialize(&app, "List_top", NULL, 0,
                        &argc, argv, NULL, NULL);


  str_list =
    (XmStringTable) XtMalloc(nb_items * sizeof (XmString *));
```

```
  for (i = 0; i < nb_items; i++)
    str_list[i] = XmStringCreateSimple(colours[i]);

  n=0;
  XtSetArg(args[n], XmNvisibleItemCount, nb_items); n++;
  XtSetArg(args[n], XmNitemCount, nb_items); n++;
  XtSetArg(args[n], XmNitems, str_list); n++;
  /* The next ressources should be in a ressource file */
  XtSetArg(args[n], XmNwidth, 300); n++;
  XtSetArg(args[n], XmNheight, 300); n++;
  list = XmCreateList(top_wid, "list1", args, n);
  XtManageChild(list);

  for (i = 0; i < nb_items; i++)
    XmStringFree(str_list[i]);
  XtFree((char *) str_list);

  /* background pixel to black foreground to white */
  cmap = DefaultColormapOfScreen(XtScreen(list));
  display = XtDisplay(list);

  XAllocNamedColor(display, cmap, colours[0], &back,
                   &spare);
  XAllocNamedColor(display, cmap, "white", &fore, &spare);

  n = 0;
  XtSetArg(args[n],XmNbackground, back.pixel); n++;
  XtSetArg(args[n],XmNforeground, fore.pixel); n++;
  XtSetValues(list, args, n);

  XtAddCallback(list, XmNdefaultActionCallback, list_cbk,
                NULL);

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*-----------------------------------------*/
void list_cbk(Widget w, XtPointer data,
              XmListCallbackStruct *list_cbs)
/*-----------------------------------------*/
/* called from any of the "Colour" list items.
   Change the color of the list widget.
   Note: we have to use dynamic setting with setargs()*/
{
  int n =0;
  Arg args[1];
  String selection;
  XColor xcolour, spare; /* xlib color struct */

  /* list->cbs holds XmString of selected list item */
  /* map this to "ordinary" string */


  XmStringGetLtoR(list_cbs->item, XmSTRING_DEFAULT_CHARSET,
                  &selection);

  if (XAllocNamedColor(display, cmap, selection,
                       &xcolour, &spare) == 0)
    return;

  XtSetArg(args[n],XmNbackground, xcolour.pixel); n++;
  /* w id of list widget passed in */
  XtSetValues(w, args, n);
}
```

11. Les widgets de type Scale.

11.1. Introduction.

Les widgets de type Scale permettent à l'utilisateur d'entrer des valeurs numériques dans un programme à l'aide d'ascenseurs. Voici un exemple de Scale widget :



11.2. Bases d'utilisation des widgets de type Scale.

Pour créer un Scale widget on utilisera par exemple une de ces fonctions : ***XmCreateScale***() ou encore ***XtVaCreateManagedWidget***() avec comme classe de widget ***xmScaleWidgetClass***. Il existe encore d'autres manières mais à ce stade du cours vous devriez les connaîtres ou mieux les deviner.

Les ressources les plus intéressantes des Scale widgets sont :

- ***XmNmaximum***: La plus grande valeur sélectionnable par l'utilisateur (lorsque l'ascenseur est en butée).
- ***XmNminimum***: La plus petite valeur sélectionnable par l'utilisateur (lorsque l'ascenseur est en butée).
- ***XmNorientation***: Deux valeurs possibles selon que l'on désire un ascenseur horizontal ou vertical : ***XmHORIZONTAL*** et ***XmVERTICAL***.
- ***XmNtitleString***: Le label affiché au-dessus de l'ascenseur (facultatif), de type ***XmString***.
- ***XmNdecimalPoints***: La valeur renvoyée par un Scale widget est toujours un entier (0 par défaut). En

positionnant cette ressource, on peut donner à l'utilisateur l'impression qu'il choisit en déplaçant l'ascenseur une valeur décimale. Par exemple si on choisit comme intervalle [0, 1000] (à l'aide des ressources ***XmNminimum*** et ***XmNmaximum***) et que l'on positionne XmNdecimalPoints avec la valeur 2, l'intervalle affiché sera égal à [0.00, 10.00]. C'est au programmeur de faire attention à effectuer la division par 100 et la conversion ***int*** -> ***float*** ou ***double***.

- ***XmNshowValue***: Deux valeurs possibles : True ou False selon que l'on désire que la valeur correspondant à la position de l'ascenseur soit affichée dans le Scale widget ou pas.
- ***XmNvalue***: La valeur courante correspondant à la position de l'ascenseur. De type ***int***.
- ***XmNprocessingDirection***: Plusieurs valeurs possibles : ***XmMAX_ON_TOP***, ***XmMAX_ON_BOTTOM***, ***XmMAX_ON_LEFT*** ou encore ***XmMAX_ON_RIGHT***. Cette ressource indique oú va se trouver la valeur maximale sélectionnable. Seulement deux valeurs sont possibles selon que le widget est vertical ou horizontal.

11.3. Fonctions de callback des Scale widgets

Il existe deux types de callback pour les Scale widgets :

- 1.***XmNvalueChangedCallback*** : la fonction de callback est appelée lorsque la valeur correspondant à la position de l'ascenseur a changé (l'utilisateur a déplacé le curseur).
- 2.***XmNdragCallback*** : le callback est appelé de manière continue lors du déplacement de l'ascenseur.

Attention, ceci peut affecter les performances de l'application si le traitement effectué dans la fonction de callback est consommateur de Cpu.

La fonction de callback d'un Scale widget est standard :

```
scale_cbk(Widget w, XtPointerData data, XmScaleCallbackStruct *cbk)
```

Le champ value de la structure ***XmScaleCallbackStruct *cbk*** contient la valeur courante (un ***int***) correspondant à la position du curseur.

11.4. Petit programme d'exemple : scale.

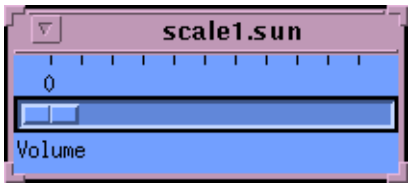
Ce petit programme simule une interface pour règler le volume sonore d'une quelconque application musicale du type xaudio, etc…

```
#include  <Xm/Xm.h>
#include <Xm/Scale.h>

/*-------------------------*/
main(int argc, char **argv)
/*-------------------------*/
{
  Widget        top_wid, scale;
  XmString      title;
  XtAppContext  app;
  void          scale_cbk();


  top_wid = XtVaAppInitialize(&app, "Scale", NULL, 0,
                        &argc, argv, NULL, NULL);

  title = XmStringCreateSimple("Volume sonore");

  scale = XtVaCreateManagedWidget("scale",
                              xmScaleWidgetClass, top_wid,
                              XmNtitleString,   title,
                              XmNorientation,    XmHORIZONTAL,
                              XmNmaximum,       11,
                              XmNdecimalPoints, 0,
                              XmNshowValue,     True,
                              XmNwidth,         200,
                              XmNheight,        100,
                              NULL);


  XtAddCallback(scale,XmNvalueChangedCallback, scale_cbk,
              NULL);

  XtRealizeWidget(top_wid);
  XtAppMainLoop(app);
}

/*-----------------------------------------------------*/
void scale_cbk(Widget widget, int data,
              XmScaleCallbackStruct *scale_struct)
/*-----------------------------------------------------*/
{
  if (scale_struct->value < 4)
    printf("Le volume sonore est trop bas (%d)\n",
          scale_struct->value);
  else if (scale_struct->value < 7)
    printf("le volume sonore est Ok (%d)\n",
          scale_struct->value);


  else
    if (scale_struct->value < 10)
      printf("Le volume sonore est fort (%d)\n",
            scale_struct->value);
    else /* Volume == 11 */
      printf("le volume sonore est vraiment trop fort (%d)\n",
            scale_struct->value);
}
```

11.5. Changer le look d'un Scale widget.

Le document ***The MOTIF Style Guide*** suggère d'utiliser des petits

marqueurs pour décorer l'echelle des valeurs des Scale widgets et ainsi leur donner l'allure de "règles". Voici un exemple :



Chaque marqueur est un widget de type ***Separator*** qui aura pour père le Scale widget. Voici les modifications que l'on pourrait apporter au programme d'exemple précédent pour qúil ressemble à la figure ci-dessus :

```
#include <Xm/SeparatoG.h>

Widget ...,tics[11];
.
.
/* Création des marqueurs le long de l'axe horizontal */
for (i=0; i < 11; ++i ) {
  n=0;
  XtSetArg(args[n], XmNseparatorType, XmSINGLE_LINE); n++;
  XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
  XtSetArg(args[n], XmNwidth, 10); n++;
  XtSetArg(args[n], XmNheight, 5); n++;
  tics[i] = XmCreateSeparatorGadget(scale, "|",
                                    args, n);
}
XtManageChildren(tics, 11);
.
.
```

12. Les chaînes de caractères sous MOTIF.

12.1. Introduction.

Motif reconnaît deux types de chaînes de caractères :

- 1.les chaines de type ***char ****, les chaînes de caractères que nous connaissons déjà (au sens de la programmation en langage C).
- 2.les chaînes de type ***XmString***. Le programmeur n'a pas besoin de connaître les détails concernant ce type d'objet. Après avoir vu la complexité de l'affichage de texte à l`aide des fonctions de la Xlib, vous vous doutez bien que MOTIF doit stocker dans le type String des informations telles que la taille en pixels de la chaîne en fonction de la police de caractères utilisée, le jambage, la baseline, etc…

MOTIF fournit de nombreuses fonctions de manipulation de chaînes de type ***XmString*** (concatenation, recopie, etc...) et de conversion entre les ***char **** et les ***XmString***.

En voici quelques unes :

- ***XmStringCreateSimple***() : création d'une chaîne simple avec la police de caractères par défaut.
- ***XmStringCreateLtoR***() : idem sauf que la chaîne de caractères est créée à l'aide d'une police passée en paramètre.
- ***XmStringGetLtoR***() : conversion de ***XmString*** vers ***char ****.
- ***XmStringCompare***() : comparaison entre deux XmStrings.
- ***XmStringConcat***() : concaténation de deux XmStrings.

- ***XmStringCopy***() : copie d'une XmString dans une autre.

Pour plus d'informations, voir le man de ces différentes fonctions. Leur utilisation est relativement simple.

12.2. Spécifier une police de caractères.

Dans une application MOTIF, une police de caractères est une variable de type ***XmFontList***, créée à partir d'un objet Xlib de type ***XFontStruct***, déjà étudié dans la partie du cours Xlib consacrée à l'affichage de texte. Typiquement, voici les étapes nécessaires pour modifier la police de caractères utilisée par un widget :

- 1.Charger une police de caractères à l'aide de fonctions Xlib. La fonction ***load_font***() (cf le cours Xlib pour des explications détaillées) fait parfaitement l'affaire :

```
/*------------------------------*/
load_font(font_info, font_name)
/*------------------------------*/
XFontStruct **font_info;
char *font_name;
{
   /* on accede a la fonte */
   if((*font_info = XLoadQueryFont(display, font_name))
        == NULL) {
      (void) fprintf(stderr,
            "Police de caractéres %s non trouvée \n",
            font_name);
      exit (-1);
   }
}
```

- 2.Créer un objet de type XmStringCharSet. En général on lui donne la valeur XmSTRING_DEFAULT_CHARSET. Cet objet, une fois initialisé, va contenir toutes les informations concernant une police de caractères : taille des caractères, jambage, etc... il sera utilisé par les fonctions de création de ***XmString*** telles que ***XmStringCreateLtoR***().
- 3.Appeler la fonction ***XmFontListCreate***(***XFontStruct *fnt***, ***XmStringCharSet charset***), qui effectue deux actions :
  - Elle initialise la variable de type ***XmStringCharSet*** avec toutes les informations relatives à la police de caractères ***XFontStruct *fnt***.
  - Elle renvoie une valeur de type XmFontList qui sera utilisée pour positionner les ressources des widgets

possèdant des labels.

12.3. Exemple d’utilisation.

Ce petit exemple montre comment on peut changer la fonte de caractères d’un PushButton figurant dans une barre de menu. Etudier en particulier le rôle de chacun des deux objets: ***charset*** et ***menu_bar_font***.

```
XFontStruct *fnt;
XmFontList menu_bar_font;  /* font pour les boutons du menu */
XmStringCharSet charset =
      (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;
.
.
.
/* C’EST ICI QUE C’EST INTERESSANT !!! */
load_font(&fnt,”vr-20”);
menu_bar_font = XmFontListCreate(fnt, charset);

menu_bar = XmCreateMenuBar (parent, “menu_bar”, NULL, 0);
XtManageChild (menu_bar);

menu_pane = XmCreatePulldownMenu (menu_bar, “menu_pane”, NULL, 0);

/* ET ICI  AUSSI !!! */
label_string = XmStringCreateLtoR(“Exit” , charset);

n = 0;
XtSetArg (args[n], XmNlabelString, label_string);  n++;
XtSetArg (args[n], XmNalignment,XmALIGNMENT_CENTER);  n++;
XtSetArg (args[n], XmNfontList, menu_bar_font);  n++;
button = XmCreatePushButton (menu_pane, “Quit”, args, n);
XtManageChild (button);
```

13. Les widgets de type ScrollBar et ScrolledWindow.

13.1. Introduction.

Motif fournit des widgets permettant de visualiser partiellement d'autres widgets en les faisant "scroller": les ScrollBar et les ScrolledWindow.

Nous avons déjà vu des ScrollBar widgets en action lors de l'étude des Text widgets et des List widgets. Motif propose en effet des méta-widgets tels que les SrolledList ou les ScrolledText widgets qui contiennent une ScrolledWindow.

13.2. Les widgets de type ScrolledWindow.

On crée les ScrolledWindow à l'aide d'une des fonctions ***XtVaCreateManagedWidget***() avec comme classe ***xmScrolledWindowWidgetClass***, ou ***XmCreateScrolledWindow***(), etc... vous devez maintenant être capable de deviner les deux autres manières de créer n'importe quel type de widget connaissant son nom.

Lorsqu'on utilise des ScrolledWindow il faut inclure le fichier <***Xm/ScrolledW.h***>.

Ressources les plus intéressantes

- ***XmNhorizontal***, ***XmNvertical***: IDs des ScrollBars de la ScrolledWindow. Ces dernières doivent donc être créées avant la ScrolledWindow.
- ***XmNScrollingPolicy***: Deux valeurs possibles : ***XmAUTOMATIC*** ou ***XmAPPLICATION_DEFINED***. Dans le premier: cas le scrolling est effectué automatiquement lorsque l'utilisateur clicke sur les flêches des ScrollBars, dans l'autre cas rien n'est effectué et le traitement du scrolling devra être fait à la main dans les fonctions de callback des ScrollBars.

- ***XmNscrollBarDisplayPolicy***: Cette ressource permet de sélectionner l'apparition de la barre de scrolling. Deux valeurs sont possibles :
  - 1.***XmSTATIC*** : une barre de scrolling verticale sera toujours présente sur le bord de la ScrolledWIndow.
  - 2.***XmAS_NEEDED*** : la barre de scrolling ne sera affichée que si le contenu de la Scrolled Window est plus grand que la taille de cette dernière.
- ***XmNvisualPolicy***: Deux valeurs possibles : ***XmCONSTANT*** ou ***XmVARIABLE***. Dans le premier cas la ScrolledWindow ne peut pas se resizer d'elle-même.
- ***XmNworkWindow***: ID du widget qui va se trouver dans la ScrolledWindow et qui va être scrollée. Il doit être crée avant la ScrolledWindow.

13.3. Les widgets de type ScrollBar.

Les fonctions de création sont standards : ***XtVaCreateManagedWidget***() (classe ***xmScrollBarWidgetClass***), etc…

13.3.1. Ressources les plus intéressantes :

- ***XmNsliderSize***: Un slider (l'ascenseur, le petit curseur que l'on déplace entre les deux flêches de la ScrollBar) est divisé en unités de longueur. Cette ressource permet de spécifier le nombre d'unités que l'on désire.
- ***XmNmaximum***: La plus grande valeur sélectionnable par l'utilisateur (lorsque le slider est en butée), en unités de longueur.
- ***XmNmaximum***: La plus petite valeur sélectionnable par l'utilisateur (lorsque le slider est en butée), en unités de longueur.
- ***XmNincrement***: Le nombre d'unités de valeur correspondant au plus petit déplacement souris du slider.
- ***XmNorientation***: Deux valeurs possibles selon que l'on désire un ascenseur horizontal ou vertical : ***XmHORIZONTAL*** et ***XmVERTICAL***.
- ***XmNvalue***: La valeur courante correspondant à la position du slider. De type ***int***.

- ***XmNpageIncrement***: Cette ressource contrôle le déplacement du widget se trouvant à l'intérieur de la ScrolledWindow en fonction du déplacement du slider.

13.3.2. Ressources de callback :

- ***XmNvalueChangedCallback***: La fonction de callback est appelée lorsque la valeur correspondant à la position de l'ascenseur a changé (l'utilisateur a déplacé le curseur).
- ***XmNdragCallback***: Le callback est appelé de manière continue lors du déplacement de l'ascenseur. Attention, ceci peut affecter les performances de l'application si le traitement effectué dans la fonction de callback est consommateur de Cpu.
- ***XmNdecrementCallback***, ***XmNincrementCallback***: Le callbaclk est appelé lorsque la valeur correspondant à la position du slider a diminué/augmenté.
- ***XmNpageDecrementCallback***, ***XmNpageIncrementCallback***: Idem au précédent sauf que le déplacement est suffisamment grand pour que le widget contenu dans la Scrolled Window soit scrollé.
- ***XmNtoTopCallback***, ***XmNtoBottomCallback***: Le callback est appelé lorsque le slider atteint une des extrêmités de la ScrollBar.

14. Les widgets de type Toggle.

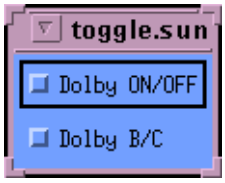
14.1. Introduction.

Un widget de type ***Toggle*** est un bouton (ou une entrée de menu) qui possède deux valeurs: ***On*** ou ***Off***. Il peut avoir la forme d'un carré (square) ou d'un diamant (diamond).

Les Toggle widgets possèdent, comme les boutons poussoirs, un label qui peut être de type texte ou bien de type ***pixmap*** (un dessin).

On peut grouper plusieurs Toggle widgets pour contrôler leur comportement les uns par rapport aux autres. Motif fournit deux types de Toggle widgets qui réagissent différemment lorsqu'ils sont regroupés:

- Le widget de type RadioBox: Un seul de ces widgets peut être sur la position On lorsqu'on en groupe plusieurs. Il y a exclusion mutuelle, lorsqu'un d'entre eux est séléctionné, les autres sont mis sur Off. Ils sont en forme de diamant.
- Le widget de type CheckBox: Plusieurs CheckBox peuvent être séléctionnés en même temps. Ils sont en forme de carré.



14.2. Bases d'utilisation des Toggle widgets.

On crée les Toggle widgets de manière standard : à l'aide des fonctions ***XmCreateToggleButton***() ou encore ***XtVaCreateManagedWidget***() avec comme classe de widget xmToggleButtonWidgetClass, etc…

14.2.1. Ressources les plus importantes.

- ***XmNindicatorType***: On positionnera cette ressource avec la valeur ***XmN_OF_MANY*** pour créer un CheckBox widget ou bien avec la valeur ***XmONE_OF_MANY*** pour créer un RadioBox widget.
- ***XmNindicatorOn***: Mettre à ***True*** pour positionner le widget sur ***On*** (valeur par défaut = ***False***).
- ***XmNindicatorSize***: Taille de l'indicateur (diamant ou carré), en pixels.
- ***XmNlabelType***: Deux valeurs possibles : ***XmSTRING*** (défaut) ou ***XmPIXMAP*** selon que le label est textuel ou graphique.
- ***XmLabelString***: Le texte du label, de type ***XmString***
- ***XmNPixmap***: Pixmap du widget lorsqu'il n'est pas séléctionné (uniquement si la ressource ***XmNlabelType*** vaut ***XmPIXMAP***).
- ***XmNselectedPixmap***: Pixmap du widget lorsqu'il est séléctionné (uniquement si la ressource ***XmNlabelType*** vaut ***XmPIXMAP***).
- ***XmNselectColour***: Couleur de l'indicateur du widget séléctionné. De type ***Pixel.*** Le Pixmap est un type X standard. Pour charger un pixmap on utilisera la fonction XmGetPixmap().

14.2.2. Callbacks des Toggle widgets.

On précise la fonction de callback en positionnant la ressource ***XmNvalueChangedCallback***.

La fonction de Callbacks d'un widget de type Toggle est standard :

```
toggle_cbk(Widget w, XtPointerData data, XmToggleCallbackStruct *cbk)
```

Le champ set de la structure XmToggleCallbackStruct permet de savoir si le widget est séléctionné ou non. Avec l'exemple ci-dessus, si cbk->set vaut True alors le widget w est séléctionné.

14.3. Petit programme d'exemple : *toggle.c.*

```
#include <Xm/Xm.h>
#include <Xm/ToggleB.h>
#include <Xm/RowColumn.h>

/*-------------------------*/
main(int argc, char **argv)
/*-------------------------*/
{
  Widget toplevel, rowcol, toggle1, toggle2;
  XtAppContext app;
  void toggle1_cbk(), toggle2_cbk();

  toplevel = XtVaAppInitialize(&app, "Toggle", NULL, 0,
                        &argc, argv, NULL, NULL);

  rowcol = XtVaCreateWidget("rowcol",
                        xmRowColumnWidgetClass, toplevel,
                        XmNwidth, 300,
                        XmNheight, 200,
                        NULL);


  toggle1 = XtVaCreateManagedWidget("Dolby ON/OFF",
                              xmToggleButtonWidgetClass, rowcol, NULL);

  XtAddCallback(toggle1, XmNvalueChangedCallback,
              toggle1_cbk, NULL);

  toggle2 = XtVaCreateManagedWidget("Dolby B/C",
                              xmToggleButtonWidgetClass, rowcol, NULL);

  XtAddCallback(toggle2, XmNvalueChangedCallback,
              toggle2_cbk, NULL);
  XtManageChild(rowcol);
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app);
}

/*-----------------------------------------------------*/
void toggle1_cbk(Widget widget, XtPointer client_data,
               XmToggleButtonCallbackStruct *state)
/*-----------------------------------------------------*/
{
  printf("%s: %s\n", XtName(widget),
        state->set? "on" : "off");
}

/*-----------------------------------------------------*/
void toggle2_cbk(Widget widget, XtPointer client_data,
XmToggleButtonCallbackStruct *state)
/*-----------------------------------------------------*/
{
  printf("%s: %s\n", XtName(widget), state->set? "B" : "C");
}
```

14.4. Grouper des Toggle widgets automatiquement.

On peut regrouper "à la main" des RadioBox ou des CheckBox widgets dans une RowCOlumn ou une Form, comme dans le programme d'exemple ***toggle.c***.

Motif propose cependant des fonctions de convénience effectuant ce travail : ***XmCreateSimpleRadioBox***() et ***XmCreateSimpleCheckBox***(). Il en

existe d'autres…

Ces fonctions permettent de créer plusieurs Toggle widgets d'un seul coup. Il sont automatiquement rassemblés dans une RowColumn, et la plupart de leurs ressources, comme ***XmNindicatorType***, sont positionnées par la fonction de convénience. Ces fonctions sont trés pratiques mais ne permettent pas de tout faire.