Structure d'un programme X

- Un programme X comporte trois parties :
  - Etablissement de la connexion au serveur.
  - Création de la hiérarchie des fenêtres (resp. des widgets).
  - Gestion de la boucle d'événements :
    - Lire un événement;
    - En fonction de la nature de l'événement et de la fenêtre où il a eu lieu, entreprendre l'action appropriée
    - envoyer les requêtes correspondantes au serveur.

```
#include <X11/Xlib.h>
Display *dpy;
int ecran;
Window fen;
GC ctx;
main(int argc, char **argv)
{
OuvrirConnexion();
CreerFenetre();
PoserFenetre();
ContexteGraphique();
BoucleEvenements();
}
```

1ère étape : Connexion au serveur X

- Le *display* définit la connexion de l'application à un serveur X. Une fois initialisée, la valeur du *display* sera utilisée dans tous les appels aux fonctions X.

- Dans le cas ou le paramètre est égal au pointeur NULL, la fonction lit la valeur de la variable d'environnement *DISPLAY*,

```
nom_du_display:0.0 pour un terminal X
(exemple: Xpierre:0)

:0.0 pour un serveur local
```

```
Display *XOpenDisplay (nom_du_display)
char *nom_du_display
```

2ème étape : Création de fenêtre

- Tout objet affiché à l'écran est décomposé en fenêtres. Les caractéristiques principales d'une fenêtres sont:
  - la fenêtre *parent*
  - la position par rapport au parent (x, y en pixels)
  - les dimensions (largeur/hauteur en pixels)
  - la couleur du fond
  - la couleur du bord
  - l'épaisseur du bord
- Le type X correspondant à la fenêtre est le type *Window*. Pour créer simplement une fenêtre, on pourra utiliser la fonction:

```
Window XCreateSimpleWindow (display, parent, x, y, width, height,
                            border_width, border_color, background)
                  Display *display;
                  Window parent;
                  int x, y;
                  unsigned int width, height, border_width;
                  unsigned long border_color, background;
```

3ème étape : Affichage des fenêtres

- Pour afficher la fenêtre, on utilisera la fonction :

```
XMapWindow (display, w)
            Display *display;
            Window w;
```

- Et pour l´effacer (de l'affichage), on utilisera :

```
XUnmapWindow (display, w)
            Display *display;
            Window w;
```

- Le système des fenêtres a une structure arborescente avec une fenêtre particulière qui est la "mère" de toutes les autres fenêtres : la *Root Window* (fond de l´écran).

Les évènements

- Le serveur filtre les évènements
- Chaque fenêtre est sensibilisée par le client et le serveur n'envoie que les évènements du type demandé.
- 33 évènements différents
- Chaque évènement comporte des informations générales :
  Type, fenêtre ou il a lieu, date/heure (en ms)
- Les évènements comportent des informations spécifiques selon leur type :
  Quelle bouton de souris enfoncé, position du clic, …

Groupes d'évènements

Evènement souris :

```
ButtonPress
ButtonRelease
MotionNotify
EnterNotify
LeaveNotify
```

Evènements clavier:

```
KeyPress
KeyRelease
FocusIn
FocusOut
```

Exposition :

```
Expose
GraphicsExpose
NoExpose
```

Notification (information d'un changement) :

```
ConfigureNotify
CirculateNotify
CreateNotify
DestroyNotify
GravityNotify
MapNotify
MappingNotify
ReparentNotify
UnmapNotify
VisibilityNotify
KeymapNotify
ColormapNotify
```

Contrôle de structure (gestionnaire de fenêtres):

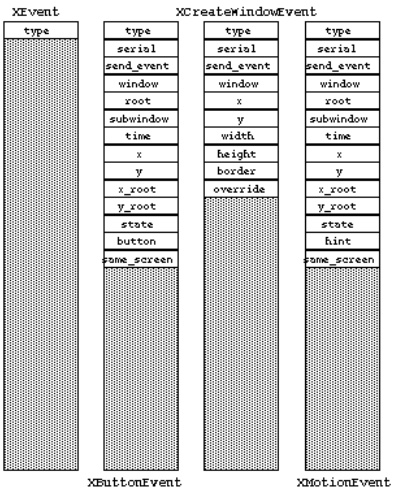
```
CirculateRequest
ConfigureRequest
MapRequest
ResizeRequest
```

Communication entre clients :

```
ClientMessage
PropertyNotify
SelectionClear
SelectionNotify
SelectionRequest
```

XEvent

- Sur le plan de la programmation, le type XEvent est une union de toutes les évènements possibles : XButtonEvent, XMotionEvent, etc…
- C'est le moment de réviser le type « union » du langage C !

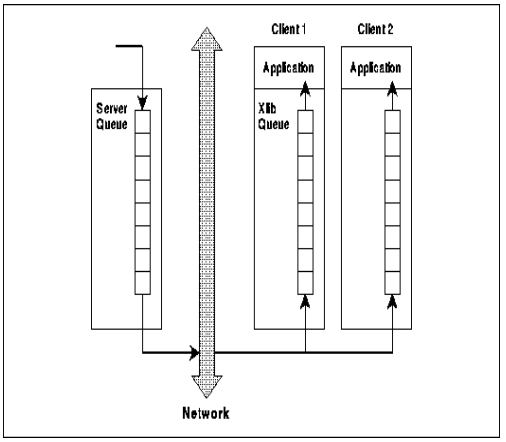


Unions

- Une union permet à des variables de taille et de type différent d'occuper le même espace mémoire (correspondant à la taille max)
- union { int a; float b; char c;} union_var;
- struct {int ID; char a, char b;, char c; char d;} first;
- struct {int ID; int pipo ;} second;
- union {struct first x; structu second y;} union_var;
  - Le champ ID est commun, toujours en 1er position dans l'espace mémoire.

Gestion des évènements

- Les évènements sont mis en file par le serveur pour chaque client intéressé, puis envoyés sur le réseau.
- Ils sont mis en file par le client qui les lit avec XNextEvent
- Les requêtes des clients sont mise en file par le serveur, exécutés dans l'ordre mais sans garantie de délai.
- Les clients peuvent forcer l'envoi des requêtes (XFlush()), mais pas leur exécution



Ce que le serveur ne fait pas

- Il n'interprète pas les évènements
- Il ne redessine pas le contenu d'une fenêtre redécouverte. En revanche, envoie un évènement expose ;
- ne fait pas de zoom
- ne fait pas de gestion logique des fenêtres (comme agrandissement, élastique, etc.) : c'est le rôle du gestionnaire de fenêtres.

4ème étape : autoriser les événements

- Les événements sont identifiés grâce aux *types* d´événement :

```
ButtonPress             appui sur le bouton de la souris
ButtonRelease           relâchement du bouton souris
KeyPress                touche appuyée
KeyRelease              touche relâchée
Expose                  dégagement d´une fenêtre obscurcie (ou
                        bien premier affichage de la fenêtre)
```

- Les événements sont reçus par les *fenêtre* à condition que l´on ait indiqué l´intérêt de la fenêtre pour cet événement :

```
XSelectInput(display, w, event_mask)
             Display *display;
             Window w;
             long event_mask;
```

5ème étape : Gestion des événements

- Boucle infinie de traitement des événements reçus.
- La lecture des événements :

```
XNextEvent (display, event_return)
            Display *display;
            XEvent *event_return;
```

- Généralement, le corps de la boucle est une grande structure de contrôle de type

```
switch(type d'evenement)
{
       case KeyPress :
              faire quelquechose
       case ButtonPress :
              faire autrechose
}
```

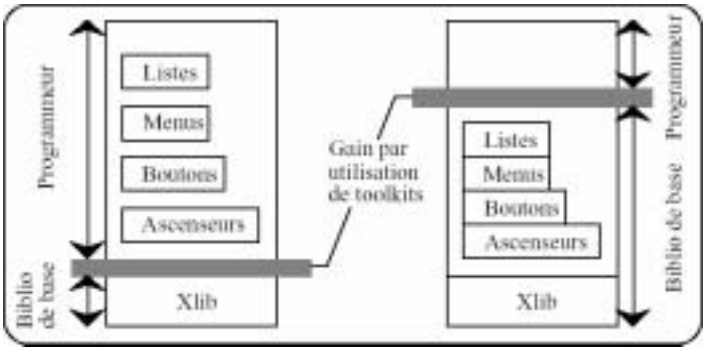
Conclusions sur la programmation Xlib

- La programmation en *Xlib* est facile pour des programmes très simples
- La gestion explicite de la boucle des évènements est :
  - Pédagogique
  - Mais très lourde lorsque l'interface devient complexe.
- La compréhension des mécanismes de base la la Xlib reste cependant une assurance pour la fiabilité des applications développées.
- Il est peu concevable de réaliser une application complète sans utiliser de toolkit (Motif, Widgets, …)

Abstraction de haut niveau

Utilisation de toolkits

- L'utilisation de Toolkits permet d'augmenter la productiovité du programmeur.
- On conservera la Xlib pour les routines graphiques.



Fonctions de Xt

- Xt : Boite à outils (toolkit) proposant des widgets et une gestion de la boucle d'événements
  - widget : objet graphique (presque autonome)
    Effet d 'enfoncement de bouton, clic de confirmation, etc...
  - fenêtre = hiérarchie d'objets
  - objet composé (conteneur) ou primaire :
    Gestion automatique de la position spatiale des fenêtres filles (geometry management)
  - toolkit ➔ homogénéité de style et navigation entre widgets
  - Gestion « cachée » de la boucle d'événements, utilisation de *callbacks*
- Les couches supérieures (MOTIF, GTK, OpenLook, etc.) exploitent Xt

Programmation MOTIF

- Bibliothèque de haut niveau qui s'est imposée face à la concurrence
- Gèré par l 'Open Software Foundation (OSF MOTIF) puis par l 'Open Group (www.opengroup.com) pour Open Motif
- Bibliothèque compatible libre : Lesstif
- Abstraction très grande des mécanismes
  - Grande collection de widgets, avec un *look & feel* uniforme
  - La boucle d'événements n'est pas explicitement accessible
  - Réaction au actions de l 'utilisateur par l 'intermédiaire de fonctions *callbacks* (fonctions réflexes)
  - Utilisation maximale des fichiers de ressources
- Surcouche « bureau » : CDE (Common Desktop Env.) et son clone KDE

Structure d'un programme MOTIF

```
fichiers de définition .h
main ( )
{
   Initialisation et création de la première widget
   Création des autres widgets avec leurs ressources
   Association de code pour ces widgets (callbacks)
   Gestion des widgets par leur mère
   Association widget-window
   Boucle d'événements
}

Fonction (....)
{
/* code lancé et associé à une widget pour
une certaine action de l'utilisateur (i.e. valeur
de ressource de callback)
*/
}
```

Pointeurs sur fonctions : rappel

- Au niveau du code machine (assembleur), le point d 'entrée d'une fonction est une adresse
- Attention : si la fonction peut changer, le nombre de ses paramètres doit rester fixe
- C'est le cas des callback qui ont toujours 3 paramètres

```
/* prototypes */
int somme();
int operateur();

main()
{
int x=10;
int y=15;
res = operateur(x,y,somme);
}

int operateur (int a,int b,int (*op)())
{
return((*op)(a,b);
}

int somme(u,v)
{
return (u+v)
}
```

Les étapes pour un programme MOTIF

- Création de la première widget de classe TopLevelShell par : `Widget XtVaAppInitialize()`
- Création des autres widgets :

```
Widget XtVaCreateWidget(String nomW,
WidgetClass classeWidget, Widget Wmere, ..., NULL)
```

  - Cette fonction retourne l'identificateur de la widget créée.
  - nomW est le nom de la widget créée (utile pour les fichiers de configuration).
  - classeWidget est la classe de la widget créée.
  - Wmere est la widget mère de la widget créée.
  - Suit une liste de paires : ressource - valeur, terminée par NULL

Création d 'un widget (1)

- Exemple d'utilisation :
  - On utilise une fonction de la toolkit, mais la classe de l 'objet est une classe MOTIF
  - Remarquez la liste de paramètres

```
#include <Xm/PushB.h>
Widget Wcree, Wmere;
Wcree = XtVaCreateWidget ("mon_bouton",
xmPushButtonWidgetClass, Wmere,
XmNwidth, 300,
XmNheight, 400,
NULL);
```

Création d 'un widget (2)

- Pour la plupart des widgets, Motif propose aussi des fonctions de la forme :

  `XmCreate<Nom de classe de widget>`

- Par exemple : `XmCreatePushButton()`
- Dès qu'un programme Motif a besoin d'une widget, il doit inclure le fichier .h contenant des définitions relatives à cette widget. Par exemple si on a besoin d'un bouton poussoir Motif il faut écrire :

  `#include <Xm/PushB.h>`

```
#include <Xm/PushB.h>
Widget XmCreatePushButton(Widget widgetmere;
String nom_de_la_widget;
ArgList TabListeArguments;
Cardinal nbEltsTabListeArguments);
```

Paramètres lors de la création d'un widget

- Attention, ces fonctions utilisent un tableau de paires ressource-valeur, manipulé par des fonctions spécifiques.

```
exemple d'utilisation :
#include <Xm/PushB.h>
int n;
Widget Wcree, Wmere;
Arg args[10];
n = 0;
XtSetArg(args[n], XmNwidth, 300); n++;
XtSetArg(args[n], XmNheight, 400); n++;
Wcree = XmCreatePushButton (Wmere,
"mon_bouton", args, n);
```

Widgets

- Boutons

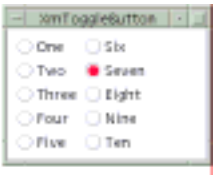
PushButton

DrawnButton

CascadeButton

ToggleButton

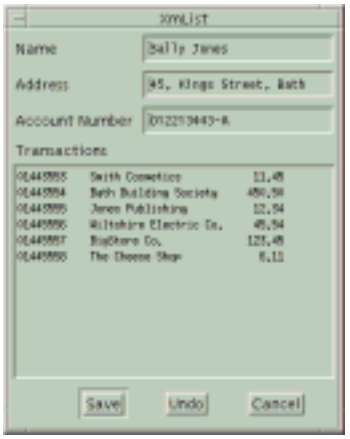
CheckBox

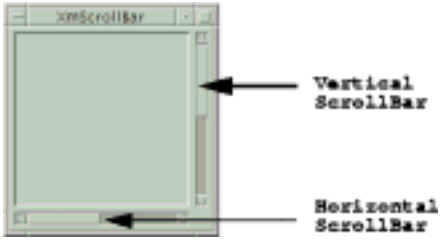


RadioBox

Widgets

- *List Class* et *ScrollBar class*





- *Text, TextField, Separator, …*

Geometry Management

- *Geometry Management* : la façon dont un widget parent contrôle la disposition de widgets fils
  - Si un seul bouton est placé dans une fenêtre, le bouton occupe toute la fenêtre
  - Si plusieurs widgets sont placé dans une fenêtre, il faut un élément intermédiaire dans la hiérarchie : les widgets manager ou container
  - **RowColumn**
    - Widget gèrant automatiquement le positionnement de plusieurs Widgets en lignes et colonnes. C'est le Manager le plus utile et le plus simple à utiliser.
    - Ce Widget est utilisé par Motif de manière sous-entendue : MenuBar, PulldownMenu, CheckBox, … sont des RowColumns!!!

Widgets de la classe Manager

- **BulletinBoard**
  - Un Widget de type BulletinBoard est un panneau sur lequel on peut coller d'autres Widgets en leur donnant des positions (x,y) et une taille absolue. Le BulletinBoard n'a aucune stratégie de placement et ne gère pas la taille de widgets fils. Aucune modification si on redimensionne un BulletinBoard
- **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. Des callbacks sont disponibles pour gérer les événements de type entrée clavier ou click souris, Expose, Resize.

La "boucle d'événements" Xt

- Le programmeur ne programme pas explicitement la boucle
- La boucle est lancée par :

```
XtAppMainLoop(XtAppContext appctx);
```

où appctx est l'argument initialisé par

```
XtVaAppInitialize()
```

- exemple d'utilisation :

```
XtContext appctx;
... = XtVaAppInitialize(&appctx, ...);
...
XtAppMainLoop(appctx);
```

Mise en place d ’un callback

- Ecrire la fonction réflexe. Elle est sur le modèle :

```
cbfn(Widget w,XtPointer client_data,XtPointer call_data);
```

  - Le type XtPointer est un pointeur non typé (void *)
  - Lorsque la fonction réflexe est appelée, les paramètres suivant lui sont passés :
    - Le pointeur w sur le widget qui a déclenché la fonction réflexe
    - La donnée client *client_data*, affecté lors de la mise en place de la fonction réflexe
    - Une donnée dépendante du contexte, généralement un pointeur sur une structure de type Event ayant généré le *callback*
- Mise en place de la fonction callback :

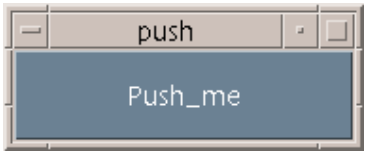
```
void XtAddCallback(Widget w, String callback_name,
XtCallbackProc callback, XtPointer client_data);
```

Mon premier programme MOTIF

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

/* Prototype Callback function */
void pushed_fn(Widget,XtPointer,XmPushButtonCallbackStruct *);

main(int argc, char **argv)

{   Widget top_wid, button;
    XtAppContext  app;

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

   button = XmCreatePushButton(top_wid, "Push_me", NULL, 0);
    /* tell Xt to manage button */
    XtManageChild(button);

    /* attach fn to widget */
    XtAddCallback(button, XmNactivateCallback, pushed_fn, NULL);

    XtRealizeWidget(top_wid); /* display widget hierarchy */
    XtAppMainLoop(app); /* enter processing loop */
}
```

Mon premier programme MOTIF

```
void pushed_fn(Widget w, XtPointer client_data,
               XtPointer call_data)
  {
     XmPushButtonCallbackStruct *ptr;
     prt = (XmPushButtonCallbackStruct)call_data;
     printf("Don't Push Me!!\n");
  }
```



Les ressources

- L'usage de ressources est un des avantages majeurs de la programmation X
- En effet, les paramètres de création des elements de l'IHM peuvent :
  - Etre codés en dur dans le code source
  - Etre extérieurs au code, dans des fichiers de ressources
    - Verrouillés par l'administrateur
    - Éditable par l'utilisateur
- Réglages généraux dans `.Xresources` ou `.Xdefaults`
- Chaque client peut disposer d'un fichier de ressource, généralement localisé dans `/usr/lib/X11/app-defaults`