Le Fox Toolkit

par khayyam90

Date de publication : 05/10/2005

Dernière mise à jour : 05/10/2005

Cet article montre comment utiliser le Fox Toolkit 1.4 pour réaliser des interfaces graphiques en C++.

I - Introduction

Vous en avez assez des applications en ligne de commande, vous voulez manipuler des fenêtres, des boutons graphiques, des barres de menus ... ? Voici un outil Open Source pour réaliser des interfaces graphiques en C++ de manière parfaitement portable. Le Fox Toolkit fonctionnera sur les systèmes Windows et Unix/Linux. Il pourra fonctionner sur un environnement Mac OS-X si un serveur X est installé.

Fox Toolkit n'est pas le seul outil pour réaliser de telles interfaces graphiques, il existe également WxWidgets, QT ...

Le Fox Toolkit se révèle proche de la WxWidgets, au niveau de la syntaxe comme au niveau de la philosophie d'utilisation. Il est donc très facile en connaissant l'un de passer à l'autre. De plus c'est une bibliothèque très légère, elle est moins volumineuse que ses principales rivales, WxWidgets et QT. La bibliothèque et les binaires sont donc d'autant plus petits et rapides à compiler et à utiliser.

Contrairement à QT, la compilation d'applications Fox ne nécessite pas l'utilisation d'un compilateur méta-objets particulier. Donc, vous l'aurez compris, la simplicité est l'un des points forts du Fox Toolkit.

Fox Toolkit est une bibliothèque relativement jeune, elle ne possède pas encore d'éditeur officiel de ressources pour construire rapidement des interfaces. Par contre, il existe un éditeur de ressources en cours de développement, dont la première version est disponible : le Fox Constructor.

Je présente ici un aperçu du Fox Toolkit. Je ne vais pas en détailler toutes les fonctionalités ni toutes les options, je vais en expliquer le principe de fonctionnement.

A l'heure où j'écris cet article, la dernière version stable du Fox Toolkit est la 1.4.17, c'est donc sur celle là que je m'appuie.

Le Fox Toolkit est distribué selon les termes de la LGPL (GNU LESSER GENERAL PUBLIC LICENSE). Le texte de la licence est disponible ici.

I-A - Installation du Fox Toolkit

Tout d'abord, téléchargez le package correspondant à votre système

- Pour Linux/Unix (3.8 Mo archivé en .tar.gz)
- Pour Windows 95 et ultérieurs (4.4 Mo archivé en .zip)

Saisissez-vous de votre compilateur C++ favori pour compiler la bibliothèque. La procédure de compilation dépend de votre compilateur, notez toutefois la procédure simplissime si vous utilisez le GNU/make et GNU/g++ :

```
./configure ; make
su
make install
```

Et pour les adeptes de DevC++, la Fox Toolkit est disponible en version déjà compilée sous forme d'un devpack.

I-B - Hello world

Pour ne pas perdre les bonnes habitudes, commençons par présenter la création d'une simple fenêtre avec un texte "hello world". La démarche est la suivante :

- Création d'une application FoxToolkit
- Création d'une instance d'une classe définissant notre fenêtre
- Association de cette instance à notre application
- Démarrage de l'application

Les noms de toutes les classes du Fox Toolkit commencent par 'FX'.

L'utilisation de cette bibliothèque en est d'autant plus confortable

Voici le code des fichiers main.cpp ainsi que NotreWindow.h et NotreWindow.cpp décrivant la classe NotreWindow :

main.cpp

```
#include <fox/fx.h>
#include "NotreWindow.h"

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

  FXApp notre_appli_fox("Hello World","FoxApp");
  notre_appli_fox.init(argc,argv);
  new NotreWindow(&notre_appli_fox);
  notre_appli_fox.create();

  return notre_appli_fox.run();
}
```

NotreWindow.h

```
#ifndef __NotreWindow_h__
#define __NotreWindow_h__

class NotreWindow : public FXMainWindow{
        FXDECLARE(NotreWindow);
        private:
                NotreWindow(){}
        public:
                NotreWindow(FXApp *);
                void create();
};

#endif
```

NotreWindow.cpp

```
#include <fox/fx.h>
#include "NotreWindow.h"

FXDEFMAP(NotreWindow) NotreWindowMap[]={};
FXIMPLEMENT(NotreWindow,FXMainWindow,NotreWindowMap,ARRAYNUMBER(NotreWindowMap))

NotreWindow::NotreWindow(FXApp *a):FXMainWindow(a,"Ma fenêtre
FOX",NULL,NULL,DECOR_ALL,100,100,250,20){
     FXLabel *lab = new FXLabel(this, "hello world",NULL,LAYOUT_FILL_X);
```

NotreWindow.cpp

```
}

void NotreWindow::create(){
  FXMainWindow::create();

  show(PLACEMENT_VISIBLE);
}
```

I-B-1 - Compilation

Encore une fois, la compilation dépend de votre compilateur. Voici un Makefile utilisable avec g++ :

```
CC     = g++
CFLAGS = -Wall -ansi
LIBS   = -lFOX-1.4
EXEC   = hello

$(EXEC) : main.cpp NotreWindow.o
       $(CC) $(CFLAGS) -o $@ $^ $(LIBS)

NotreWindow.o: NotreWindow.cpp NotreWindow.h
       $(CC) $(CFLAGS) -o NotreWindow.o -c NotreWindow.cpp
```

Au besoin, rajoutez le répertoire contenant les headers du Fox Toolkit s'ils ne figurent pas dans la variable d'environnement $PATH

I-B-2 - Explications du code de la classe NotreWindow

```
...
class NotreWindow : public FXMainWindow{
...
```

Notre classe va implémener une fenêtre graphique, elle hérite donc de la classe FXMainWindow correspondant aux fenêtres principales des applications Fox.

```
...
private:
 NotreWindow(){}
public:
 NotreWindow(FXApp *);
...
```

NotreWindow possède deux constructeurs NotreWindow() et NotreWindow(FXApp*). De par l'architecture fortement orientée objet du FoxToolkit, il n'est pas nécessaire d'implémenter de destructeur pour notre classe NotreWindow. Tous les objets seront détruits à la fin de l'application par le destructeur par défaut. Le premier constructeur déclaré privé est requis pour l'utilisation de la macro FXIMPLEMENT, permettant d'associer une liste d'évènements et de méthodes à une classe. La macro FXDEFMAP permet de définir les associations entre les évènements et des méthodes. Nous reparlerons de ces macros lors de l'utilisation des évènements. Le second constructeur déclaré public permet d'associer une application Fox à une fenêtre. Il fait appel au constructeur de la classe mère FXMainWindow.

Le constructeur de la classe FXMainWindow prend 9 arguments :

- un pointeur vers l'application Fox
- l'intitulé de la fenêtre

- 2 pointeurs vers des icônes
- une liste de drapeaux décrivant l'allure de la fenêtre
- la position (x,y) du coin supérieur gauche de la fenêtre par rapport au coin supérieur gauche de l'écran
- la taille (x,y) de la fenêtre, en pixels

```
FXLabel *lab = new FXLabel(this, "hello world",NULL,LAYOUT_FILL_X);
```

Dans le constructeur de notre fenêtre, nous insérons les objets graphiques dont nous avons besoin, ici, un simple label avec "Hello World". Le constructeur du FXLabel demande les arguments suivants :

- Un pointeur sur l'objet contenant le label
- Le texte à afficher
- Un pointeur sur une icône
- Une liste de drapeaux définissant l'allure du label

```
void NotreWindow::create(){
  FXMainWindow::create();

  show(PLACEMENT_VISIBLE);
}
```

La méthode NotreWindow::create nous permet de créer notre FXMainWindow (non plus au sens des classes C++ mais au sens de Fox). Cette méthode affiche ensuite notre fenêtre d'après un drapeau de positionnement.

la fenêtre créée

I-B-3 - Les drapeaux

Les drapeaux décrivant l'allure d'une fenêtre :

- DECOR_NONE : aucun attribut graphique
- DECOR_TITLE : la barre de titre, mais pas de bouton de maximisation / minimisation / fermeture
- DECOR_MINIMIZE : le bouton de minimisation
- DECOR_MAXIMIZE : le bouton de maximisation
- DECOR_CLOSE : le bouton de fermeture (généralement une croix)
- DECOR_BORDER : la bordure
- DECOR_RESIZE : possibilité de redimensionner la fenêtre à la souris
- DECOR_MENU : le menu déroulant au clic sur l'icône dans la barre de titre
- DECOR_ALL : tous les attributs graphiques

Notez que ce sont des drapeaux, ils peuvent donc être combinés entre eux via l'opérateur |. Ainsi pour obtenir les effets combinés de 3 drapeaux, il faut écrire drapeau1|drapeau2|drapeau3

Quelques uns des drapeaux pour le positionnement d'une fenêtre :

- PLACEMENT_DEFAULT : la fenêtre sera positionnée d'après les arguments passés au constructeur de FXMainWindow.
- PLACEMENT_VISIBLE : idem que pour PLACEMENT_DEFAULT, mais la fenêtre sera active.

- PLACEMENT_CURSOR : la fenêtre sera affichée à la position du curseur.
- PLACEMENT_SCREEN : la fenêtre sera au centre de l'écran.
- PLACEMENT_MAXIMIZED : la fenêtre sera maximisée.

II - La gestion des évènements

Pour rendre notre application dynamique, il est possible (et heureusement) d'associer des méthodes à des évènements.

II-A - Principe de communication entre les objets

Chaque objet interactif possède un pointeur vers l'objet auquel il doit envoyer ses messages. L'objet qui reçoit les messages connait les associations messages / méthodes, il peut donc appeler les fonctions correspondantes lorsqu'il reçoit des messages. Chaque type de message possède un identifiant ainsi qu'une correspondance avec un évènement. Donc, quand un objet reçoit un message, il en connait l'identifiant ainsi que l'évènement qui a généré le message.

La liste des associations entre messages et méthodes est définie par la macro FXDEFMAP. Mais c'est au programmeur de renseigner cette macro de la manière suivante :

```
FXDEFMAP(NotreWindow) NotreWindowMap[]={
                      //___Type du message_______Identifiant_______Méthode_________
                      FXMAPFUNC(SEL_COMMAND , NotreWindow::ID_NEW , NotreWindow::fct1),
                      FXMAPFUNC(SEL_MINIMIZE, NotreWindow::ID_END , NotreWindow::fct2),
                      FXMAPFUNC(SEL_COMMAND , NotreWindow::ID_FOX , NotreWindow::fct3),
                      FXMAPFUNC(SEL_MAXIMIZE, NotreWindow::ID_TOTO, NotreWindow::fct2),
                      [...]
                      FXMAPFUNC(SEL_MOTION  , NotreWindow::ID_TATA, NotreWindow::fct4)
                      };
```

Vous pouvez trouver la liste des types des messages ici.

Une fois la liste établie, il faut appeler la seconde macro qui permet d'implémenter notre liste d'associations

```
FXIMPLEMENT(NotreWindow,FXMainWindow,NotreWindowMap,ARRAYNUMBER(NotreWindowMap))
```

Reste maintenant à informer la classe NotreWindow de l'existence de ces types de messages, en rajoutant un type enuméré en attribut de classe ainsi que les prototypes des fonctions associées.

```
class NotreWindow: public FXMainWindow{
FXDECLARE(NotreWindow);
private:
   ...
public:
   ...
   enum{
     ID_ZERO,
     ID_NEW,
     ID_END,
     ID_FOX,
     ID_TOTO,
     ID_TATA
     };

   long fct1(FXObject*,FXSelector,void*);
   long fct2(FXObject*,FXSelector,void*);
   long fct3(FXObject*,FXSelector,void*);
   long fct4(FXObject*,FXSelector,void*);
};
```

Les fonctions à appeler lors d'un évènement ont toujours le prototype suivant :

```
long nom_fct(FXObject*,FXSelector,void*);
```

où FXObject* est un pointeur sur l'objet qui a émis le message, FXSelector est le type du message et void* est un pointeur sur une structure représentant l'évènement qui a amené l'émission du message.

Vous en mourez d'envie, voici un petit exemple pour illustrer l'utilisation des évènements. Implémentons un label dont le texte change au clic sur un bouton.

NotreWindow.h

```
#ifndef __NotreWindow_h__
#define __NotreWindow_h__

class NotreWindow : public FXMainWindow {
        FXDECLARE(NotreWindow);
        private:
                NotreWindow(){}
                FXLabel *lab;
                FXButton *b1,*b2;
        public:
                NotreWindow(FXApp *);
                void create();
                enum{
                        ID_ZERO,
                        ID_CHANG,
                        ID_DEVEL
                        };

                long onChangez(FXObject*,FXSelector,void*);
                long onDeveloppez(FXObject*,FXSelector,void*);
};

#endif
```

NotreWindow.cpp

```
#include <fox/fx.h>
#include "NotreWindow.h"

FXDEFMAP(NotreWindow) NotreWindowMap[]={
                  FXMAPFUNC(SEL_COMMAND, NotreWindow::ID_CHANG   , NotreWindow::onChangez),
                  FXMAPFUNC(SEL_COMMAND, NotreWindow::ID_DEVEL   , NotreWindow::onDeveloppez)
                  };
FXIMPLEMENT(NotreWindow,FXMainWindow,NotreWindowMap,ARRAYNUMBER(NotreWindowMap))

NotreWindow::NotreWindow(FXApp *a):FXMainWindow(a,"Ma fenêtre
FOX",NULL,NULL,DECOR_ALL,50,100,250,80){
     int buttonStyle = FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT;

     lab=new FXLabel(  this, "hello world",NULL,JUSTIFY_CENTER_X|LAYOUT_FILL_X );
     b1 =new FXButton( this, "Changez"    ,NULL,this,ID_CHANG,buttonStyle,0,0,0,0,10,10,5,5);
     b2 =new FXButton( this, "Developpez" ,NULL,this,ID_DEVEL,buttonStyle,0,0,0,0,10,10,5,5);
}

void NotreWindow::create(){
  FXMainWindow::create();

  show(PLACEMENT_VISIBLE);
}

long NotreWindow::onDeveloppez(FXObject*,FXSelector,void*){
     lab->setText("developpez");
     return 0;
}

long NotreWindow::onChangez(FXObject*,FXSelector,void*){
     lab->setText("changez");
     return 0;
}
```

II-B - Explications du code

J'ai déclaré deux pointeurs sur des boutons (FXButton), créés dans le constructeur public de notre classe. On remarquera la présence du premier argument **this** qui définit l'objet graphique contenant notre bouton. Le second argument positionné à **this** correspond à un pointeur sur l'objet à qui envoyer les messages, et dans notre cas, nous envoyons les messages à la fenêtre. On remarquera aussi les arguments ID_CHANG et ID_DEVEL décrivant le type des messages que les boutons émettront.

Comme la fenêtre sera amenée à recevoir des messages, elle doit connaître les associations entre les messages et les fonctions à appeler, ceci est fait via la macro FXDEFMAP

```
FXMAPFUNC(SEL_COMMAND, NotreWindow::ID_CHANG   , NotreWindow::onChangez),
FXMAPFUNC(SEL_COMMAND, NotreWindow::ID_DEVEL   , NotreWindow::onDeveloppez)
```

Pour éviter les erreurs de résolution de symbole et les références indéfinies, il ne faut pas oublier de créer les méthodes onChangez et onDeveloppez :

```
long NotreWindow::onDeveloppez(FXObject*,FXSelector,void*){
    lab->setText("developpez");
    return 0;
}

long NotreWindow::onChangez(FXObject*,FXSelector,void*){
    lab->setText("changez");
    return 0;
}
```

ainsi que les types de messages ID_CHANG et ID_DEVEL

```
class NotreWindow : public FXMainWindow {
    ...
 public:
    ...
 enum{
    ID_ZERO,
    ID_CHANG,
    ID_DEVEL
   };
    ...
};
```

Attention à ne pas oublier de déclarer un premier item non utilisé pour le type énuméré. En effet, le premier item est géré comme étant 0, valeur ne convenant pas pour les message Fox. En ne déclarant pas le premier item, vous prenez le risque d'obtenir des comportements indéfinis.

III - Le positionnement des objets

Un soucis majeur de notre fenêtre précédente est le positionnement des objets dans notre fenêtre, ils se mettent simplement les uns en dessous des autres. Le Fox Toolkit met à notre disposition les concepts de conteneur vertical et conteneur horizontal.

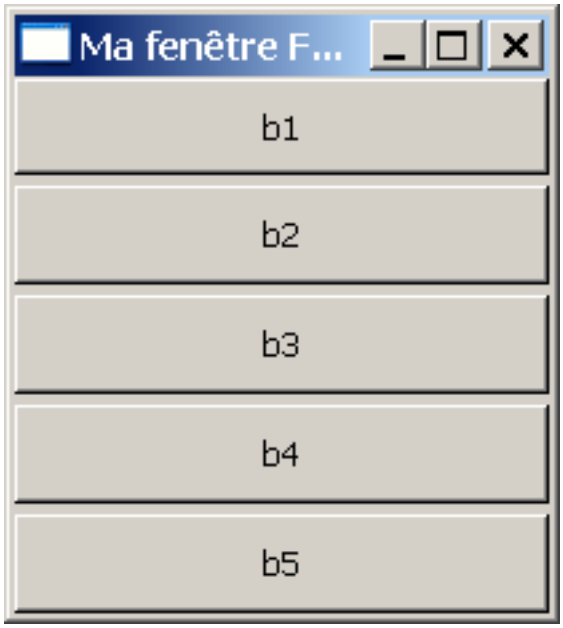
Dans un conteneur vertical, les objets seront placés les uns en dessous des autres tandis que dans un conteneur horizontal, ils seront placés les uns à côté des autres.

III-A - Le conteneur vertical

```
    int styleFrame = LAYOUT_SIDE_TOP|LAYOUT_FILL_X|LAYOUT_FILL_Y;
    int styleButton = FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X|LAYOUT_FILL_Y;

    FXVerticalFrame *vf1 = new FXVerticalFrame(this,styleFrame,0,0,0,0,0,0,0,0);
    FXButton *b1 = new FXButton( vf1, "b1" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b2 = new FXButton( vf1, "b2" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b3 = new FXButton( vf1, "b3" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b4 = new FXButton( vf1, "b4" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b5 = new FXButton( vf1, "b5" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
```

J'ai créé un conteneur vertical nommé vf1, contenu lui-même par **this**, ici, la fenêtre. Et tous les autres objets, b1, b2, b3, b4 et b5 sont contenus par vf1.



le conteneur vertical

III-B - Le conteneur horizontal

```
    int styleFrame = LAYOUT_SIDE_TOP|LAYOUT_FILL_X|LAYOUT_FILL_Y;
    int styleButton = FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X|LAYOUT_FILL_Y;

    FXHorizontalFrame *hf1 = new FXHorizontalFrame(this,styleFrame,0,0,0,0,0,0,0,0);
    FXButton *b1 = new FXButton( hf1, "b1" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b2 = new FXButton( hf1, "b2" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b3 = new FXButton( hf1, "b3" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
    FXButton *b4 = new FXButton( hf1, "b4" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
```

```
    FXButton *b5 = new FXButton( hf1, "b5" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
```

Le principe est le même que pour le conteneur vertical, tous les objets sont contenus par hf1, qui est un conteneur horizontal.

le conteneur horizontal

III-C - Là où ça devient vraiment intéressant

De par le modèle objet du Fox Toolkit, les conteneurs peuvent contenir des objets graphiques, mais ils sont eux aussi des objets graphiques. Un conteneur peut donc contenir .... des conteneurs. Démonstration :

```
    int styleFrame = LAYOUT_SIDE_TOP|LAYOUT_FILL_X|LAYOUT_FILL_Y;
    int styleButton = FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X|LAYOUT_FILL_Y|LAYOUT_TOP|LAYOUT_LEFT;

    FXVerticalFrame *vf1 = new FXVerticalFrame(this,styleFrame,0,0,0,0,0,0,0,0);

    FXButton *b1 = new FXButton( vf1, "b1" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);

    FXHorizontalFrame *hf1 = new FXHorizontalFrame(vf1,styleFrame,0,0,0,0,0,0,0,0);
      FXButton *b2 = new FXButton( hf1, "b2" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
      FXButton *b3 = new FXButton( hf1, "b3" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);

    FXHorizontalFrame *hf2 = new FXHorizontalFrame(vf1,styleFrame,0,0,0,0,0,0,0,0);
      FXButton *b4 = new FXButton( hf2, "b4" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
      FXButton *b5 = new FXButton( hf2, "b5" ,NULL,NULL,0,styleButton,0,0,0,0,10,10,5,5);
```

Je créé un conteneur vertical noté vf1 dans lequel j'insère un bouton (b1) puis deux conteneurs horizontaux (hf1 et hf2) qui contiennent les 4 autres boutons.

On remarquera que le premier argument de chaque bouton et de chaque conteneur correspond à son propre conteneur, vf1, hf1, hf2 ou bien **this**.

des conteneurs qui contiennent des conteneurs

IV - Quelques contrôles

Nous avons déjà vu l'utilisation du label et du bouton, voyons maintenant d'autres contrôles. Le C++ autorise la surcharge des constructeurs / méthodes, c'est pourquoi il existe plusieurs moyens de créer les contrôles suivants.

IV-A - La checkbox

La checkbox permet de cocher ou décocher des options

```
FXCheckButton *chk = new FXCheckButton(this, "Checkbox", this, ID_CLIC);
```

Le premier argument du constructeur est un pointeur sur l'objet contenant la checkbox. Le second correspond au texte à afficher, le troisième est un pointeur sur l'objet auquel les messages doivent être envoyés et le dernier argument correspond au type des messages à envoyer.

La méthode getCheck() renvoie l'état de la checkbox sous forme d'un booléen.

IV-B - Le textfield

Le textfield permet de saisir du texte

```
FXTextField *txt= new FXTextField(this, 5, this, ID_CLIC);
```

Le premier argument du constructeur est un pointeur sur l'objet contenant le textfield. Le second correspond à la largeur du textfield, le troisième est un pointeur sur l'objet auquel les messages doivent être envoyés et le dernier argument correspond au type des messages à envoyer.

La méthode getText() renvoie le contenu du textfield sous forme d'une chaîne de caractères .

IV-C - La liste déroulante

La liste déroulante permet de sélectionner un élément dans une liste.

```
FXListBox *lstb= new FXListBox (this, this, ID_CLIC, FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X);
lstb->insertItem (0,"zero");
```

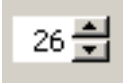
```
    lstb->insertItem (1,"un");
    lstb->insertItem (2,"deux");
    lstb->insertItem (3,"trois");
    lstb->insertItem (4,"quatre");
    lstb->insertItem (5,"cinq");
```

Le premier argument du constructeur est un pointeur sur l'objet contenant la liste déroulante. Le second est un pointeur sur l'objet auquel les messages doivent être envoyés. Le troisième est le type du message à envoyer et le dernier argument est une liste de drapeaux représentant le style de la liste.

La méthode insertItem permet d'ajouter des items dans la liste. Chaque item possède un indice et un intitulé.

La méthode getCurrentItem() renvoie l'indice de l'item sélectionné.

IV-D - Le Spin bouton



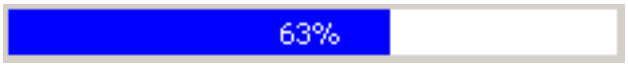
Il permet de saisir des nombres.

```
    FXSpinner *spin= new FXSpinner(this, 3, this, ID_CLIC);
    spin->setRange(10,200);
```

Le premier argument du constructeur est un pointeur sur l'objet contenant le spin bouton. Le second correspond à la largeur du spin bouton, le troisième est un pointeur sur l'objet auquel les messages doivent être envoyés et le dernier argument correspond au type des messages à envoyer. Un des intérêts majeurs des spins boutons est de pouvoir définir des valeurs de saisie autorisées. Ceci se fait via la méthode setRange.

La méthode getValue() renvoie la valeur courante du spin bouton.

IV-E - La barre de progression



Elle permet de représenter l'état d'avancement d'un traitement.

```
    FXProgressBar *p= new FXProgressBar (this, this, ID_CLIC,LAYOUT_FILL_X);
    p->showNumber ();
    p->setBarSize(10);
    p->setTotal (200);
```

Le premier argument du constructeur est un pointeur sur l'objet contenant la barre. Le second est un pointeur sur l'objet auquel les messages doivent être envoyés. Le troisième correspond au type des messages à envoyer. Le dernier argument est une liste de drapeaux pour définir le style de la barre de progression.

La méthode showNumber permet d'afficher le pourcentage d'avancement de la barre.

V - La création d'une barre de menus

Plusieurs objets graphiques sont nécessaires à la création d'une barre de menus : la barre elle-même, les menus, le panel de chaque menu et les items des menus. Un exemple sera plus parlant :

V-A - Code source

NotreWindow.h

```
class NotreWindow ...
private:
  ....
  FXMenuBar *menu_barre;
  FXMenuTitle *menu_file, *menu_edition;
  FXMenuPane *file_pan, *edition_pan;
  FXMenuCommand *item_nouveau, *item_open, *item_close, *item_undo, *item_copy;
   ...
};
```

NotreWindow.cpp

```
        ...
menu_barre=new
FXMenuBar(this,LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X,0,0,0,0,3,3,2,2,DEFAULT_SPACING,DEFAULT_SPACING);

file_pan   =new FXMenuPane(this,0);
edition_pan=new FXMenuPane(this,0);

menu_file=new FXMenuTitle(menu_barre,"Fichier",NULL,file_pan,0);
  item_nouveau=new FXMenuCommand(file_pan,"Nouveau",NULL,this,ID_NEW,0);
  item_open   =new FXMenuCommand(file_pan,"Ouvrir" ,NULL,this,ID_OPEN,0);
  item_close  =new FXMenuCommand(file_pan,"Fermer" ,NULL,this,ID_CLOSE,0);

menu_edition=new FXMenuTitle(menu_barre,"Edition",NULL,edition_pan,0);
  item_undo   =new FXMenuCommand(edition_pan,"Annuler",NULL,this,ID_UNDO,0);
  item_copy   =new FXMenuCommand(edition_pan,"Copier" ,NULL,this,ID_COPY,0);
     ...
```

Le code source n'est pas complet, il ne contient que les morceaux de code qu'il faut rajouter dans les codes précédents. La structure de la fenêtre étant sensiblement la même.

V-B - Explications

```
menu_barre = new
FXMenuBar(this,LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X,0,0,0,0,3,3,2,2,DEFAULT_SPACING,DEFAULT_SPACING);
```

Cette instruction créé une barre de menus. Le premier argument est un pointeur sur l'objet dans lequel la barre devra se trouver, ici, la fenêtre. L'autre argument contient les drapeaux définissant l'aspect de la barre.

```
file_pan = new FXMenuPane(this,0);
```

Cette instruction créé un panel pour un menu déroulant. Cela va donc créer un espace graphique à dérouler. On remarquera la présence de l'argument **this**, qui pointe sur l'objet contenant le panel, ici, la fenêtre.

```
menu_file = new FXMenuTitle(menu_barre,"Fichier",NULL,file_pan,0);
```

Cette instruction crée un menu déroulant dans la barre de menus, avec un intitulé "Fichier", sans icône, associé au panel file_pan, sans option particulière.

Il ne reste plus qu'à remplir notre menu avec les items souhaités :

```
    item_nouveau = new FXMenuCommand(file_pan,"Nouveau",NULL,this,ID_NEW,0);
```

Nous créons ici un item associé au panel file_pan, intitulé "Nouveau" . Cet item n'a pas d'icône (NULL). Il enverra ses messages à **this**, ici, la fenêtre. Les messages qu'il enverra seront du type ID_NEW. Le dernier argument correspond aux drapeaux de positionnement et d'allure de l'item.

Il ne reste plus maintenant qu'à déclarer les message ID_NEW, ID_CLOSE... dans le type énuméré de la classe NotreWindow, à les déclarer dans la macro FXDEFMAP et à implémenter les méthodes que vous souhaitez leur associer.

D'un point de vue esthétique, il est possible de rajouter des séparateurs entre les items d'un menu :

```
    FXMenuSeparator *sep1 = new FXMenuSeparator(file_pan,0);
```

Une barre de menus

VI - Conclusion

VI-A - Les possibilités du Fox Toolkit

De par sa conception fortement orientée objet, le Fox Toolkit se révèle capable de créer des applications très complètes. Le Fox Toolkit possède d'autres talents parmis lesquels la gestion de contexte OpenGL via les classes FXGLContext, FXGLViewer ... ; la gestion des formats d'images via les classes FXPNGImage, FXBMPmage, FXGIFImage, FXJPGImage ... ; ou encore la gestion des projets MDI (Multiple Document Interface) via les classes FXMDIChild, FXMDIClient ... Les screenshots des projets en cours parlent encore le mieux des possibilités offertes par le Fox Toolkit.

VI-B - Liens

- Le site officiel du Fox Toolkit
- La FAQ officielle
- Le site du Fox Constructor
- Quelques projets utilisant le Fox Toolkit

VI-C - Remerciements

Merci Aurelien.Regat-Barrel, bigboomshakala et farscape pour le temps qu'ils ont bien voulu me consacrer pour la relecture de cet article.