# Le Toolkit Helium

## Tutorial

Edouard Thiel

4 février 2002

# Table des matières

# Chapitre 1

# Introduction

Ce document est à la fois un tutorial sur Helium, sur Xlib et sur OpenGL. Xlib est la librairie du système XWindow, qui permet l'affichage distant, et est le standard dans le monde Unix. OpenGL est le standard de fait de l'affichage 3D.

Helium est une librairie pour créer des interfaces graphiques utilisateur (GUI). La license est la LGPL (GNU Library General Public License), ce qui signifie que vous pouvez développer des logiciels libres ou commerciaux utilisant Helium sans payer de license ou verser de royalties. Le texte de cette licence est dans le fichier LICENSE.

Le toolkit est appelé Helium parce que l'helium est léger et stable ! Helium est écrit en C ANSI, sur Xlib, la librairie de base de X11. Le but est l'efficacité, et la portabilité en Unix/X11. La librairie libHelium est écrite sur un modèle objet minimal, avec les idées de classes et de callbacks (fonctions appelées automatiquement).

L'API (Application Program Interface) est en C ANSI ; elle est conçue pour être très simple d'emploi, et résistant autant que possible aux erreurs dans la manipulation des « widgets » (les objets graphiques). On peut par exemple accéder à un widget qui vient d'être détruit, sa destruction réelle étant différée à la fin de l'évènement en cours. D'autre part, une suite d'opérations sur un même widget ne produit qu'un seul affichage final.

Helium est spécialement conçu pour le graphisme. Le widget Canvas permet de faire un zone de dessin Xlib, qui reçoit les évènements X11. Des primitives fournies permettent de précalculer une XImage (morceau d'image prête à être plaquée instantanément à l'écran) sans se soucier du nombre de plans ou du byte-order. Des fonctions permettent de faire de la conversion de couleurs. Le Canvas est optimisé pour limiter au maximum le nombre de réaffichage, par exemple lorsqu'on fait sortir et rentrer la fenêtre à l'écran.

Le toolkit est livré avec une seconde librairie, libHeliumGL, qui définit l'objet GLArea. Ce widget permet de faire du OpenGL, en appelant Mesa ou en adressant directement la carte graphique (sur SGI par exemple). L'avantage par rapport à GLUT est de pouvoir placer dans une même fenêtre des boutons, zones de saisie etc, *et* un widget GLArea.

Helium est en développement ; les mécanismes internes sont maintenant très au point, et les widgets de base sont achevés. La prochaine étape est l'écriture des widgets « menu » et « sélecteur de fichiers ».





La plus récente version de libHelium et de ce tutorial peuvent être téléchargés ici : `http://www.lif-sud.univ-mrs.fr/~thiel/helium`

# Chapitre 2

# Premiers pas

Dans ce chapitre on apprend les notions essentielles de « widget », de « callback » et de boucle d'évènement.

Tous les exemples du tutorial sont sous forme de sources et d'exécutables dans le repertoire `examples/`. Pour télécharger un fichier source depuis votre navigateur il suffit de maintenir enfoncé le bouton [Shift] et de cliquer sur le lien.

## 2.1 Ouvrir une fenêtre

Voici un programme qui ouvre une fenêtre "Hello World".

```
/* examples/hello/hello.c */

#include <helium.h>

He_node *princ;

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Hello World");

    return HeMainLoop (princ);
}
```

La ligne `#include <helium.h>` inclut tous les fichiers `.h` standards (`stdio.h`, `stdlib.h`, etc, mais pas `math.h`) et les fichiers `.h` de X11, puis définit tous les types et prototypes de fonctions de Helium.

Un « widget » est un objet tel qu'une fenêtre, un bouton, un champ de saisie, etc. Tous les widgets d'Helium ont le type `He_node*`. Ici on déclare le widget `princ`, qui sera la fenêtre principale.

La fonction `main` reçoit les arguments de la ligne de commande du système, et renvoie le code de sortie entier (prototype ANSI).





On commence par appeler `HeInit`, qui initialise Helium, ouvre le display X11 (ou échoue). `HeInit` reçoit les arguments de la ligne de commande, les analyse et retire de `argv` toutes les options reconnues (telles que `-display`, `-geom`, `-color`, etc). C'est pourquoi on passe `argc` et `argv` avec des `&`.

On crée ensuite la fenêtre principale, le nom de la classe fenêtre étant « Frame » dans Helium. La fonction `HeCreateFrame()` renvoie l'adresse du widget créé. On ne manipule jamais directement les champs d'un `He_node`, qui est un type interne à Helium, mais on utilise les nombreuses fonctions fournies par Helium. Par exemple ici, on donne un titre à la fenêtre avec `HeSetFrameLabel()`.

Une fois que tous les widgets on été créés, on appelle `HeMainLoop` en lui donnant la fenêtre principale. `HeMainLoop` fait les choses suivantes : elle affiche la fenêtre principale, applique éventuellement l'option `-geom` sur la taille et la position de la fenêtre, puis lance la boucle d'évènements infinie, qui attend puis répartit chaque évènement (souris, clavier, etc).

La fonction `HeMainLoop` sort lorsqu'on ferme la fenêtre par le Window Manager, et donne un code de sortie que l'on renvoie à `main()` ; c'est le code de sortie du programme.

Remarque : il ne doit y avoir qu'un seul appel à `HeInit` et à `HeMainLoop` dans tout le programme.

## 2.2 Compilation

Dans cette section, on montre 4 façons différentes de compiler un programme utilisant Helium : en ligne, avec un script shell, avec un Makefile simple, enfin avec un Makefile sophistiqué.

Pour compiler l'exemple précédent, il suffit de taper :

```
gcc hello.c -o hello `/chemin-helium/helium-cfg --cflags --libs`
```

la commande entre backquotes ` ` permet de retrouver les options dépendant de l'installation d'Helium ; il suffit de remplacer `/chemin-helium` par le bon chemin absolu.

Lancer le programme avec les options suivantes et observer :

```
hello &
hello -help
hello -res
hello -geom 600x500+200-100
```

Pour faciliter la compilation des exemples de ce tutorial, voici un petit script en sh (remplacer `/chemin-helium` par le chemin absolu) :

```
#! /bin/sh
p=/chemin-helium
f=`basename $1 .c`
gcc $f.c -o $f `$p/helium-cfg --cflags --libs`
```

Appeler ce script "hcomp", sauver, taper "`chmod +x hcomp`". Pour compiler un fichier "`ex.c`", on tape "`./hcomp ex.c`" ou "`./hcomp ex`". Pour compiler un exemple et lancer automatiquement l'exécution si la compilation a réussie, on tape "`./hcomp ex.c && ex`".

Voici un Makefile simple ; on inclut le fichier de configuration de Helium `/chemin-helium/.config`, où les variables nécessaires sont déclarées, en particulier `$(CC)`, `$(HE_CFLAGS)` et `$(HE_LIBS)`.
Attention, vérifiez que les lignes décalées commencent bien par un [TAB].

```
include /chemin-helium/.config

.c.o :
	$(CC) -c $(HE_CFLAGS) $*.c

bouton : bouton.o
	$(CC) -o $@ $@.o $(HE_LIBS)
```

Enfin, voici un Makefile sophistiqué, qui affiche un Message d'aide, permet de compiler plusieurs programmes et de nettoyer le répertoire.

```
include /chemin-helium/.config

.c.o :
	$(CC) -c $(HE_CFLAGS) $*.c

# Rajoutez ici le nom de votre_prog
EXECS = bouton bouton2 bouton3

help ::
	@echo "Options du make : help all clean distclean $(EXECS)"

# Rajoutez ici votre_prog : votre_prog.o
bouton : bouton.o
bouton2 : bouton2.o
bouton3 : bouton3.o

all :: $(EXECS)

$(EXECS) :
	$(CC) -o $@ $@.o $(HE_LIBS)

clean ::
	\rm -f *.o core

distclean :: clean
	\rm -f $(EXECS)
```



## 2.3 Premier bouton

Voici un programme qui ouvre une fenêtre avec un bouton "Press me".

```
/* examples/button/pressme.c */

#include <helium.h>

He_node *princ, *panel, *butt;

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Premier bouton");

    panel = HeCreatePanel (princ);

    butt = HeCreateButton (panel);
    HeSetButtonLabel (butt, "Press me");

    return HeMainLoop (princ);
}
```

On ne peut pas attacher directement un bouton à une fenêtre ; il faut créer un objet intermédiaire, le widget Panel : c'est un conteneur de boutons, chargé de leur transmettre les évènements X11.

On crée donc la fenêtre `princ` ; ensuite on crée le Panel avec `HeCreatePanel(propriétaire)`, le propriétaire étant `princ`. Enfin on crée le bouton `butt`, dont le propriétaire est le Panel.

## 2.4 Principe général

D'une façon générale, on crée un widget de la catégorie Class par

```
hn = HeCreateClass(owner);
```

(remplacer Class par Frame, Panel, Button, Text, etc). Un Frame n'a pas de propriétaire `owner` ; le propriétaire d'un Panel est en général un Frame ; le propriétaire d'un bouton, d'un message, d'un champ de saisie, etc est toujours un Panel.

Tous les widgets ont des propriétés communes, telles que taille, position, visible, actif, etc ; on peut changer une propriété Prop par

```
HeSetProp (hn, value);
```

(remplacer Prop par X, Y, Width, Height, Show, Active, etc). On peut récupérer la valeur de cette propriété par

```
value = HeGetProp (hn);
```

Les widgets d'une classe Class ont aussi des propriétés spécifiques à leur classe, par exemple le nom d'un bouton, la valeur d'un champ de saisie, etc ; on peut changer une telle propriété Prop par

```
HeSetClassProp (hn, value);
```

(remplacer ClassProp par ButtonLabel, TextValue, MessageBold, etc). On peut récupérer la valeur de cette propriété par

```
value = HeGetClassProp (hn);
```

Pour détruire un widget, on appelle

```
HeDestroy (hn);
```

Cet appel détruit le widget et tous ses fils. Tout widget détruit est masqué, c'est-à-dire effacé de l'écran.

L'affichage est entièrement automatique ; lorsqu'on crée un widget, modifie ses propriété, ou détruit le widget, Helium reporte les changements nécessaires à l'écran (en une seule fois).

## 2.5 Action liée au bouton

Le programme `pressme.c` ne déclenche aucune action lorsqu'on clique dessus. On va modifier le programme pour qu'il affiche le nom du bouton.

```
/* examples/button/callback.c */

#include <helium.h>

He_node *princ, *panel, *butt;

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel (hn);

    printf ("butt_proc: %s\n", nom);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Action liée au bouton");

    panel = HeCreatePanel (princ);

    butt = HeCreateButton (panel);
    HeSetButtonLabel (butt, "Press me");
    HeSetButtonNotifyProc (butt, butt_proc);

    return HeMainLoop (princ);
}
```

On appelle « callback » une fonction de votre programme, qui est appelée automatiquement par Helium lorsqu'un certain évènement se produit.

Il faut dire à Helium quelle callback doit être appelée dans quelle circonstance. Cela s'appelle « attacher » une callback.



Par exemple ici, on attache la callback `butt_proc` au bouton `butt`. Cette fonction sera appelée automatiquement par Helium chaque fois que le bouton sera pressé.

Une callback a toujours un prototype précis, imposé par Helium (puisque c'est Helium qui appelle votre fonction).

Ici, la fonction reçoit un seul paramètre, qui est le bouton pressé. Ainsi, plusieurs boutons peuvent avoir la même callback NotifyProc et déclencher une action dépendant par exemple du nom du bouton.

## 2.6 Raccourci pour le bouton

On peut remplacer ces lignes

```
He_node *butt;
butt = HeCreateButton (panel);
HeSetButtonLabel (butt, "Press me");
HeSetButtonNotifyProc (butt, butt_proc);
```

par

```
He_node *butt;
butt = HeCreateButtonP (panel, "Press me", butt_proc, NULL);
```

Si on n'a pas besoin de mémoriser butt, on peut écrire

```
HeCreateButtonP (panel, "Press me", butt_proc, NULL);
```

L'exemple `callback.c` devient donc

```
/* examples/button/raccourci.c */

#include <helium.h>

He_node *princ, *panel;

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel (hn);
    printf ("butt_proc: %s\n", nom);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Raccourci du bouton");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Press me", butt_proc, NULL);

    return HeMainLoop (princ);
}
```

Le quatrième paramètre de HeCreateButtonP permet d'attacher un ClientData au bouton (voir la section « 9.1 Une donnée ClientData »).

## 2.7 Quitter un programme

Dans l'exemple suivant on crée deux boutons : le premier est appelé "Press me", et le second "Quit". Les deux boutons ont chacun leur propre callback.

```
/* examples/button/quit.c */

#include <helium.h>

He_node *princ, *panel;

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel (hn);
    printf ("butt_proc: %s\n", nom);
}

void quit_proc (He_node *hn)
{
    HeQuit (0);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Quitter un programme");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Press me", butt_proc, NULL);
    HeCreateButtonP (panel, "Quit", quit_proc, NULL);

    return HeMainLoop (princ);
}
```

Pour quitter le programme, l'utilisateur presse le bouton "Quit". À ce moment, Helium appelle la callback du bouton, qui est `quit_proc`. Dans `quit_proc`, on appelle `HeQuit(0)`, ce qui demande à Helium de quitter proprement le programme.

Le paramètre de HeQuit est le code de sortie de programme ; typiquement 0 signifie pas d'erreur, et 1 signifie erreur.

On peut appeler HeQuit dans n'importe quelle partie du programme, que ce soit avant, pendant ou après HeInit ou HeMainLoop : HeQuit réagit différemment en s'adaptant à la situation.

Lorsque HeQuit est appelé dans une callback, c'est à dire pendant la durée de vie de HeMainLoop, Hequit détruit récursivement tous les widgets puis ferme le display X11.

# Chapitre 3

# Placement des widgets

Lorsqu'on crée un certain nombre de widgets, ils se placent toujours à un endroit par défaut, que vous pouvez ensuite modifier. Vous pouvez aussi cacher un widget (il n'est plus à l'écran et ne réagit plus) ou l'inactiver (il est visible mais ne réagit plus).

## 3.1 Placement manuel

Tout les widgets possèdent des coordonnées x,y par rapport au propriétaire, l'origine étant le coin intérieur en haut à gauche du propriétaire.

À la création, la plupart des widgets sont placés en 0,0 par défaut.

On récupère les coordonnées d'un widget hn par

```
int x1 = HeGetX(hn), y1 = HegetY(hn);
```

et on modifie ces coordonnées par

```
HeSetX(hn, x2); HeSetY(hn, y2);
```

Chaque widget possède également une largeur width et une hauteur height

```
int w1 = HeGetWidth(hn), h1 = HeGetHeight(hn);
```

que l'on modifie par

```
HeSetWidth(hn, w2); HeSetHeight(hn, h2);
```

Les changements au niveau de l'affichage sont automatiques. On peut déplacer un widget de dx,dy par

```
HeSetX (hn, HeGetX(hn) + dx); HeSetY (hn, HeGetY(hn) + dy);
```

ou par le raccourci

```
HeMoveX (hn, dx); HeMoveY (hn, dy);
```

On peut de même changer la taille d'un widget avec HeMoveWidth et HeMoveHeight.





## 3.2 Placement dans un Panel

Lorsqu'un widget est créé dans un Panel, le Panel fixe les coordonnées initiales du widget. Les coordonnées du widget peuvent ensuite être modifiée avec les fonctions `HeSetX`, `HeSetY`.

Par défaut, les widgets sont placés à la suite les uns des autres, sur une même ligne horizontale. On peut changer cela avec la fonction

```
HeSetPanelLayout (panel, layout);
```

où layout vaut `HE_HORIZONTAL`, `HE_VERTICAL` ou `HE_LINE_FEED`.

Avec `HE_VERTICAL` les widgets seront alignés les uns en dessous des autres ; avec `HE_LINE_FEED` les widgets sont placés horizontalement, mais on change de ligne au moment où on fait cet appel. Ainsi

```
HeCreateButtonP (panel, "La", NULL, NULL);
HeCreateButtonP (panel, "première", NULL, NULL);
HeCreateButtonP (panel, "ligne", NULL, NULL);

HeSetPanelLayout(panel, HE_LINE_FEED);

HeCreateButtonP (panel, "puis", NULL, NULL);
HeCreateButtonP (panel, "la", NULL, NULL);
HeCreateButtonP (panel, "seconde", NULL, NULL);
```

crée deux lignes horizontales de boutons. On peut modifier l'espacement entre les widgets avec

```
HeSetPanelXGap (panel, xgap);
HeSetPanelYGap (panel, ygap);
```

qu'il faut appeler AVANT de créer les widgets. On peut de même modifier l'espacement entre les widgets et le bord du Panel avec

```
HeSetPanelXAlign (panel, xalign);
HeSetPanelYAlign (panel, yalign);
```

On peut récupérer les valeurs par défaut avec la fonction `HeGetPanel...(panel)` correspondante.

## 3.3 Ajuster la taille

À un moment donné, on peut demander d'ajuster la taille d'un widget par rapport à son contenu ; il suffit d'appeler

```
HeFit (hn);
```

Pour certains widgets l'ajustement est automatique. Par exemple, les widgets Button et Message recalculent leur taille chaque fois que l'on change leur Label (on peut désactiver le mécanisme en faisant `HeSetAutoFit(hn,FALSE);` ).

Pour d'autres widgets l'opération est manuelle. C'est le cas pour les widgets Frame et Panel : une fois que tous les boutons sont créés dans un Panel, on ajuste le Panel par rapport aux boutons, et ensuite on ajuste le Frame par rapport au Panel. On fait donc dans l'ordre :

```
HeFit (panel);
HeFit (frame);
```

Pour certains widgets enfin, l'ajustement n'a pas de sens. C'est le cas pour les widgets de dessin Canvas et GLArea. La fonction `HeFit` ne fait tout simplement rien.

Dans l'exemple suivant on crée deux lignes de boutons et on ajuste la taille du Panel et de la fenêtre :

```
/* examples/button/layout.c */

#include <helium.h>

He_node *princ, *panel;

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel (hn);
    printf ("butt_proc: %s\n", nom);
}

void quit_proc (He_node *hn)
{
    HeQuit (0);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Placement et ajustement");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Trouvez", butt_proc, NULL);
    HeCreateButtonP (panel, "le", butt_proc, NULL);
    HeCreateButtonP (panel, "bouton", butt_proc, NULL);

    HeSetPanelLayout(panel, HE_LINE_FEED);

    HeCreateButtonP (panel, "pour", butt_proc, NULL);
    HeCreateButtonP (panel, "me", butt_proc, NULL);
    HeCreateButtonP (panel, "quitter", quit_proc, NULL);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```



## 3.4 Alignement et élargissement

Les fonctions `HeGetWidth` et `HeGetHeight` renvoient la largeur et hauteur *intérieure* d'un widget.

Pour certains widgets tels que les boutons, la taille intérieure correspond à la taille extérieure (c'est-à-dire totale) du widget. Ce n'est pas le cas pour le Panel ou le Canvas, qui peuvent avoir un bord (en général de 1 pixel, d'où largeur extérieure = largeur intérieure + 2). Ce n'est pas non plus le cas pour le Frame : il faut compter la taille de la barre de titre et de la décoration, qui sont rajoutés par le Window Manager.

On a donc introduit les fonctions suivantes, qui traitent tous les cas de figure : la largeur et la hauteur *extérieure* d'un widget s'obtient par

```
int w1 = HeGetExtWidth(hn), h1 = HeGetExtHeight(hn);
```

les coordonnées du point à l'extérieur en bas à droite du widget sont obtenues par

```
int x1 = HeGetExtX(hn), y1 = HeGetExtY(hn);
```

Attention, les fonctions `HeSetExt...()` n'existent pas pour le moment.

Pour placer un widget `h2` à droite d'un widget `h1`, on peut faire :

```
HeSetX (h2, HeGetExtX(h1));
```

Si ces widgets sont dans un Panel, il faut tenir compte de l'espacement horizontal du Panel, et faire à la place

```
HeSetX (h2, HeGetExtX(h1) + HeGetPanelXGap(panel));
```

Pour simplifier l'alignement des widgets, on introduit la fonction `HeJustify`, qui traite tous les cas de figure.

```
HeJustify (hn, ref, pos);
```

justifie (c'est-à-dire aligne sur un bord) le widget `hn` par rapport au widget `ref`. Si `ref` est NULL, le widget `hn` est justifié par rapport au bord de son père (très pratique). Si `ref` est non NULL, `ref` et `hn` doivent avoir le même père (pour qu'ils aient le même système de coordonnées relatives) sinon il y a un message d'erreur. `pos` peut être l'une des constantes `HE_LEFT`, `HE_RIGHT`, `HE_TOP`, `HE_BOTTOM` ou `HE_TOP_LEFT`, `HE_BOTTOM_LEFT`, `HE_TOP_RIGHT`, `HE_BOTTOM_RIGHT` (qui font deux justifications à la fois).

Dans notre exemple, pour placer `h2` *à droite* de `h1` il suffit de justifier `h2` *à gauche* par rapport à `h1` :

```
HeJustify (h2, h1, HE_LEFT);
```

Le même problème se pose lorsqu'on veut élargir un widget pour qu'il occupe toute la place disponible jusqu'à un autre widget ou jusqu'au bord du propriétaire. On introduit la fonction `HeExpand`, qui traite tous les cas de figure (maintien de l'espacement dans un Panel, pas d'espacement dans un Frame, taille de la décoration des Frame, etc) :

```
HeExpand (hn, ref, pos);
```

élargit (ou rétrécit) le widget `hn` jusqu'au widget `ref`. Si `ref` est NULL, le widget `hn` est élargi jusqu'au bord de son père. Si `ref` est non NULL, `ref` et `hn` doivent avoir le même père sinon il y a un message d'erreur. `pos` peut être l'une des constantes `HE_LEFT`, `HE_RIGHT`, `HE_TOP`, `HE_BOTTOM` ou `HE_TOP_LEFT`, `HE_BOTTOM_LEFT`, `HE_TOP_RIGHT`, `HE_BOTTOM_RIGHT`.

Par exemple, pour que `h1` occupe tout l'espace à gauche de `h2`, on élargit `h1` à droite vers `h2` :

```
HeExpand (h1, h2, HE_RIGHT);
```

On verra plus loin l'utilité de ces fonctions pour adapter la position et la taille des widgets lorsqu'une fenêtre est redimensionnée.

## 3.5 Afficher/cacher

Tout widget peut être visible ou caché. Si un widget est caché, on ne le voit pas à l'écran et il ne réagit pas. On utilise les fonctions :

```
void HeSetShow (He_node *hn, int show);
int HeGetShow (He_node *hn);
```

pour afficher ou cacher un widget hn. Par défaut, à la création tous les widgets sont visibles, sauf les Frame qui sont tous cachés. (voir la section « Afficher une fenêtre »).

L'exemple suivant montre comment réaliser un bouton qui fait apparaître ou disparaître un autre bouton.

```
/* examples/button/show.c */

#include <helium.h>

He_node *princ, *panel, *butt1;

void show_proc (He_node *hn)
{
    HeSetShow (butt1, !HeGetShow(butt1));
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Afficher un bouton");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Afficher/cacher", show_proc, NULL);
    butt1 = HeCreateButtonP (panel, "Cible", NULL, NULL);

    return HeMainLoop (princ);
}
```



## 3.6 Activer/inactiver

La plupart des widget peuvent être inactivés. Un widget inactivé est un widget visible, mais qui ne réagit pas. Les widgets inactifs sont en général affichés dans une couleur plus claire. On utilise les fonctions :

```
void HeSetActive (He_node *hn, int active);
int HeGetActive (He_node *hn);
```

Par défaut, à la création tous les widgets sont actifs. L'exemple suivant montre comment réaliser un bouton qui active ou inactive un autre bouton.

```
/* examples/button/active.c */

#include <helium.h>

He_node *princ, *panel, *butt1;

void acti_proc (He_node *hn)
{
    HeSetActive (butt1, !HeGetActive(butt1));
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Activer un bouton");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Activer/inactiver", acti_proc, NULL);
    butt1 = HeCreateButtonP (panel, "Cible", NULL, NULL);

    return HeMainLoop (princ);
}
```

# Chapitre 4

# Fenêtres : le widget Frame

Dans ce chapitre on explique comment créer plusieurs fenêtres dans le même programme, les afficher, changer leur taille, les positionner à l'écran, et intercepter leur fermeture.

## 4.1 Afficher une fenêtre

Dans l'exemple suivant, on crée plusieurs fenêtres :

```
/* examples/frame/show1.c */

#include <helium.h>

He_node *princ, *fen1, *fen2;

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Fenêtre principale");

    fen1 = HeCreateFrame ();
    HeSetFrameLabel (fen1, "Fenêtre 1");

    fen2 = HeCreateFrame ();
    HeSetFrameLabel (fen2, "Fenêtre 2");

    return HeMainLoop (princ);
}
```

Si on exécute cet exemple, on ne verra que la fenêtre principale. En effet, on a vu dans la section « 3.5 Afficher/cacher » la règle suivante : par défaut, à la création tous les widgets sont visibles, sauf les Frame qui sont tous cachés.

Pour qu'une fenêtre `fen` apparaisse à l'écran il faut donc faire :

```
HeSetShow (fen, TRUE);
```

Alors pourquoi la fenêtre principale `princ` apparaît-elle ? Parce que `HeMainLoop(princ)` se charge de l'afficher en appelant en interne `HeSetShow(princ, TRUE)`.





Le but est le suivant : on crée les fenêtres d'une application dans le `main()`, et celles-ci restent cachées. On affiche ensuite ces fenêtres à la demande (par exemple dans la callback d'un bouton).

On réécrit l'exemple `show1.c` en affichant `fen1` tout de suite, et avec un bouton qui fait apparaître ou disparaître `fen2` :

```
/* examples/frame/show2.c */

#include <helium.h>

He_node *princ, *fen1, *fen2, *panel;

void show_proc (He_node *hn)
{
    HeSetShow (fen2, !HeGetShow(fen2));
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Fenêtre principale");

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "Afficher/cacher", show_proc, NULL);

    fen1 = HeCreateFrame ();
    HeSetFrameLabel (fen1, "Fenêtre 1");
    HeSetShow (fen1, TRUE);

    fen2 = HeCreateFrame ();
    HeSetFrameLabel (fen2, "Fenêtre 2");

    return HeMainLoop (princ);
}
```

Remarque : lorsque les fenêtres sont affichées, le fait d'iconifier la fenêtre principale fait disparaître toutes les autres fenêtres ; la désiconification réalise l'opération inverse.

## 4.2 Géométrie de la fenêtre

On appelle "géométrie de la fenêtre" la taille et de la position de la fenêtre.

Dans la philosophie X11, c'est le Window Manager qui est responsable en dernier ressort de la géométrie de la fenêtre, et on considère en général que se battre contre un Window Manager est une très mauvaise idée. A fortiori quand on sait qu'il existe au moins une 20aine de Window Managers différents, qui ont tous des stratégies différentes.

Nonobstant, Helium fait tout ce qu'il peut pour vous aider à placer les fenêtres où vous voulez qu'elles soient.

Le Window Manager décore une fenêtre en y rajoutant une barre de titre, des coins de redimensionnements, etc. On distingue la taille intérieure de la fenêtre, de la taille extérieure de la fenêtre (taille extérieure = intérieure + décoration).

La taille de la décoration est calculée par Helium et stockée dans les variables globales `he_wmdeco_width` et `he_wmdeco_height`. Ainsi, la largeur et la hauteur totale de la fenêtre `fen` sont

```
HeGetWidth (fen) + he_wmdeco_width
HeGetHeight(fen) + he_wmdeco_height
```

ou plus simplement (cf section 3.4 )

```
HeGetExtWidth (fen)
HeGetExtHeight(fen)
```

Les coordonnées x,y que l'on emploie avec HeSetX/Y ou HeGetX/Y sont les coordonnées du coin extérieur en haut à gauche de la fenêtre, par rapport au coin en haut à gauche de l'écran. Pour placer une fenêtre en haut à gauche on fait donc

```
HeSetX (fen, 0); HeSetY (fen, 0);
```

Pour placer une fenêtre en bas à droite il faut se servir de la taille de la décoration, et aussi de la taille de l'écran, qui est stockée dans les variables globales `he_root_width` et `he_root_height` :

```
HeSetX(fen, he_root_width - HeGetWidth(fen) - he_wmdeco_width);
HeSetY(fen, he_root_height- HeGetHeight(fen)- he_wmdeco_height);
```

on obtient le même résultat en justifiant la fenêtre en bas à droite (cf section 3.4 ) :

```
HeJustify(fen, NULL, HE_BOTTOM_RIGHT);
```

Helium fournit une fonction qui permet un placement simple des fenêtres :

```
HeSetFrameGeom (fen, char *geom);
```

où geom est un string désignant la position et/ou la taille de la fenêtre, dans la plus pure syntaxe de X11 (wxh+-x+-y) ; C'est la même syntaxe que pour l'argument "`-geom`" de la ligne de commande.

Par exemple, pour placer la fenêtre en haut à gauche on fait :

```
HeSetFrameGeom (fen, "+0+0");
```

Pour la placer en bas à droite on fait :

```
HeSetFrameGeom (fen, "-0-0");
```

Pour faire une fenêtre de taille intérieure 500,400 située à 100 du bord gauche et à 200 du bord bas on fait :

```
HeSetFrameGeom (fen, "500x400+100-200");
```

Voici un exemple avec des boutons qui repositionnent la fenêtre ; le positionnement reste correct si on agrandit la fenêtre :



```
/* examples/frame/hautbas.c */

#include <helium.h>

He_node *princ, *panel;

void butt1_proc (He_node *hn)
{
    HeSetFrameGeom (princ, "+0+0");
}

void butt2_proc (He_node *hn)
{
    HeSetFrameGeom (princ, "-0-0");
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    printf ("he_wmdeco_width  = %d\n", he_wmdeco_width);
    printf ("he_wmdeco_height = %d\n", he_wmdeco_height);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Positionnement de la fenêtre");

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "En haut à gauche", butt1_proc, NULL);
    HeCreateButtonP (panel, "En bas à droite", butt2_proc, NULL);

    HeFit (panel);
    HeFit (princ);

    return HeMainLoop (princ);
}
```

## 4.3 Fenêtre redimensionnée

Le programme peut être averti chaque fois que la taille de la fenêtre est modifiée ; il suffit d'attacher une callback ResizeProc à la fenêtre avec

```
HeSetFrameResizeProc (fen, fen_resize_proc);
```

Le prototype de la callback est

```
void fen_resize_proc (He_node *hn, int width, int height);
```

où `hn` est le Frame, `width` et `height` sont les nouvelles dimensions de l'intérieur de la fenêtre.

Dans l'exemple suivant on fait afficher la nouvelle taille de la fenêtre chaque fois que l'utilisateur redimensionne la fenêtre.

```
/* examples/frame/resize1.c */

#include <helium.h>

He_node *princ;
```

```
void fen_resize_proc (He_node *hn, int width, int height)
{
    printf ("fen_resize_proc %d %d\n", width, height);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Change ma taille");
    HeSetFrameResizeProc (princ, fen_resize_proc);

    return HeMainLoop (princ);
}
```

Comme on le voit, la ResizeProc est appelée exactement une fois à l'ouverture de la fenêtre (juste avant que son contenu ne soit affiché) puis une fois à chaque redimensionnement.

Un HeSetWidth ou un HeSetHeight sur la fenêtre provoque aussi un appel à la ResizeProc. Plusieurs HeSetWidth/Height à la suite ne provoquent qu'un seul appel.

En fait la ResizeProc est l'endroit où l'on repositionne le contenu de la fenêtre en fonction de la nouvelle taille, à coups de HeSetX/Y/Width/Height, HeJustify, HeExpand, etc, sur les widgets contenus.

Dans l'exemple suivant, on crée deux boutons, et on force le deuxième bouton à se placer toujours à la droite de la fenêtre :

```
/* examples/frame/resize2.c */

#include <helium.h>

He_node *princ, *panel, *butt1, *butt2;
int largeur_min;

void fen_resize_proc (He_node *hn, int width, int height)
{
    /* largeur minimale ; provoque un nouveau resize */
    if (width < largeur_min)
        { HeSetWidth (hn, largeur_min); return; }

    /* ajuste largeur du panel pour que son contenu soit visible */
    HeSetWidth (panel, width);

    /* Justifie butt2 à droite */
    HeJustify (butt2, NULL, HE_RIGHT);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Bouton à droite");
    HeSetFrameResizeProc (princ, fen_resize_proc);

    panel = HeCreatePanel (princ);
    butt1 = HeCreateButtonP (panel, "Gauche", NULL, NULL);
    butt2 = HeCreateButtonP (panel, "Droite", NULL, NULL);

    HeFit (panel);
```



```
    largeur_min = HeGetWidth(panel);
    HeFit (princ);

    return HeMainLoop (princ);
}
```

## 4.4 Fenêtre déplacée

Le programme peut être averti chaque fois que l'utilisateur déplace la fenêtre ; il suffit d'attacher une callback ReplaceProc à la fenêtre avec

```
HeSetFrameReplaceProc (fen, fen_replace_proc);
```

Le prototype de la callback est

```
void fen_replace_proc (He_node *fen, int xb, int yb);
```

où `fen` est le Frame, `xb` et `yb` sont les nouvelles coordonnées du coin extérieur en haut à gauche de la fenêtre.

Dans l'exemple suivant, on crée deux fenêtres `princ` et `fen1` ; chaque fois que `princ` est déplacée par l'utilisateur, on place `fen1` à sa droite :

```
/* examples/frame/replace.c */

#include <helium.h>

He_node *princ, *fen1;

void fen_replace_proc (He_node *hn, int xb, int yb)
{
    printf ("fen_replace_proc : %d %d\n", xb, yb);
    HeJustify (fen1, princ, HE_LEFT);
    HeSetY (fen1, HeGetY(princ));
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Deplace-moi");
    HeSetFrameReplaceProc (princ, fen_replace_proc);

    fen1 = HeCreateFrame ();
    HeSetFrameLabel (fen1, "Je me mets à droite");
    HeSetShow (fen1, TRUE);

    return HeMainLoop (princ);
}
```

Exercice : rajouter une ResizeProc qui met `fen1` à droite de `princ` chaque fois que l'utilisateur redimensionne `princ`.

Remarques :

▷ la ReplaceProc n'est pas appelée lorsque le programme change la position de la fenêtre, alors que la ResizeProc est appelée lorsque le programme change la taille de la fenêtre.

▷ Si le programme ne précise pas la position de la fenêtre, le Window Manager est laissé libre de placer la fenêtre où bon lui semble (vivement conseillé par la philosophie X11) ; dès qu'une fenêtre a une position, celle-ci est mémorisée de telle sorte qu'après désiconification par exemple, elle réapparaisse au même endroit que celui où elle était.

## 4.5 Intercepter la fermeture

Lorsque l'utilisateur ferme la fenêtre principale par le Window Manager, l'application est quittée. On peut intercepter la fermeture d'une fenêtre et dire à ce moment ce qu'on veut faire.

On commence par attacher une callback CloseProc à la fenêtre `fen` :

```
HeSetFrameCloseProc (fen, fen_close_proc);
```

Le prototype de la callback est :

```
void fen_close_proc (He_node *hn);
```

où hn est le Frame.

Lorsqu'une fenêtre est fermée par le biais du Window Manager, le comportement de Helium est le suivant :

▷ si la fenêtre n'a pas de callback CloseProc
  - si c'est la fenêtre principale, le programme est quitté avec un code de sortie 0 par `HeQuit(0)` ;
  - sinon la fenêtre est simplement masquée par `HeSetShow (hn, FALSE)` ;

▷ sinon la fenêtre n'est pas masquée et le programme n'est pas quitté, mais la callback est appelée, et c'est dans la callback qu'on décide ce que l'on veut faire.

Dans l'exemple suivant on crée une fenêtre principale qui refuse de se fermer :

```
/* examples/frame/close.c */

#include <helium.h>

He_node *princ;

void fen_close_proc (He_node *hn)
{
    /* Décommenter pour que le programme accepte de quitter */
    /* HeQuit(0); */
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Essaie de me quitter");
    HeSetFrameCloseProc (princ, fen_close_proc);

    return HeMainLoop (princ);
}
```



Pour que la fenêtre se ferme et que l'application soit quittée, il suffit de rajouter `HeQuit(0)` ; dans la callback.

Remarque : cette callback est l'emplacement idéal pour appeler une boîte de dialogue "Voulez-vous quitter ? Oui/Non". Voir la section « 6.1 Boîte de dialogue ».

# Chapitre 5

# Widgets du Panel

On a déjà vu que les boutons doivent obligatoirement être créés dans un Panel, dont le rôle est de placer les boutons et de leur distribuer les évènements.

Dans ce chapitre, on présente d'autres widgets, qui doivent aussi être créés dans un Panel : le message sur une ligne, le champ de saisie, et le bouton à bascule.

## 5.1 Message sur une ligne

Le widget Message sert à afficher une légende sur une seule ligne, en fonte normale (par défaut) ou en gras.

```
He_node *mess;
mess = HeCreateMessage (panel);
HeSetMessageLabel (mess, "Le texte");
HeSetMessageBold (mess, TRUE);
```

Un raccourcis équivalent est

```
mess = HeCreateMessageP (panel, "Le texte", TRUE);
```

On peut récupérer le texte et l'attribut gras par

```
char *texte = HeGetMessageLabel (mess);
int   gras  = HeGetMessageBold (mess);
```

Dans l'exemple suivant, on affiche 4 boutons sur une ligne et un Message sur la ligne du dessous ; la callback attachée aux boutons récupère le nom du bouton pressé, et change le texte du Message.

```
/* examples/panel/message.c */

#include <helium.h>

He_node *princ, *panel, *mess;
```





```
void butt_proc (He_node *hn)
{
    char buf[200],
         *nom = HeGetButtonLabel (hn);

    sprintf (buf, "Vous avez cliqué sur \"%s\"", nom);
    HeSetMessageLabel (mess, buf);

    if (!strcmp (nom, "Quit")) HeQuit(0);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Message sur une ligne");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Ne pressez pas", butt_proc, NULL);
    HeCreateButtonP (panel, "sur le", butt_proc, NULL);
    HeCreateButtonP (panel, "bouton", butt_proc, NULL);
    HeCreateButtonP (panel, "Quit", butt_proc, NULL);

    HeSetPanelLayout(panel, HE_LINE_FEED);

    mess = HeCreateMessageP (panel, "Cliquez sur un bouton", FALSE);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

Chaque fois que l'on change le texte du Message, la largeur du widget est automatiquement ajustée. Par défaut, la longueur maximale du texte est limitée à 80 caractères (la suite est tronquée). Pour modifier cette limite, utiliser

```
void HeSetMessageMaxLen (He_node *hn, int max_len);
int HeGetMessageMaxLen (He_node *hn);
```

Pour supprimer la limite, appeler `HeSetMessageMaxLen(hn, 0)` ;

## 5.2 Champ de Saisie

### 5.2.1 Principe

Le widget Text permet d'afficher un champ de saisie ; c'est un véritable éditeur de texte sur une ligne, avec mode insertion, affichage des caractères de contrôle, sélection, double et triple clic, déplacement de sélection, copier/coller avec les boutons gauche et milieu de la souris, couper/copier/coller dans le clipboard avec ALT-x,c,v, la complétion de noms de fichiers, etc.

Il y a de très nombreuses fonctions dans l'API pour manipuler ce widget ; nous y reviendrons plus loin dans le tutorial.

```
He_node *text;
text = HeCreateText (panel);
HeSetTextVisibleLen (text, 20);
HeSetTextValue (text, "Texte par défaut");
```

Ceci crée un champ de saisie de 20 caractères de larges (mais sans limite de taille pour la chaîne de caractère saisie par l'utilisateur), avec un "Texte par défaut".

Pour récupérer la chaîne saisie, il faut faire

```
char *blabla;
blabla = HeGetTextValue (text);
```

La longueur de cette chaîne est

```
int len = HeGetTextLen (text);
```

ATTENTION, la chaîne renvoyée n'est pas une copie, c'est l'adresse de la chaîne interne au widget. Cette adresse est donc très VOLATILE, car la moindre opération de l'utilisateur ou par le programme sur le widget Text peut libérer le string pointé par blabla. L'utilisation normale de blabla est donc la comparaison immédiate avec `!strcmp()`, ou la copie avec `strdup()` ou `sprintf()`.

On peut déclencher une callback NotifyProc lorsque l'utilisateur valide avec la touche [Entrée] :

```
HeSetTextNotifyProc (text, entree_proc);
```

Le prototype de la callback est

```
void entree_proc (He_node *hn);
```

Dans l'exemple suivant on crée un Message, un champ de saisie, un bouton "lit valeur" et un bouton "Quit". Le texte saisi est affiché lorsque l'utilisateur clique sur le bouton ou lorsque l'utilisateur appuie sur la touche [Entrée].

```
/* examples/panel/text.c */

#include <helium.h>

He_node *princ, *panel, *text;

void entree_proc (He_node *hn)
{
    printf ("[enter] : \"%s\"\n", HeGetTextValue (hn));
}

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel (hn);

    if (!strcmp (nom, "Quit"))
         HeQuit(0);
    else printf ("lit valeur : \"%s\"\n", HeGetTextValue (text));
}
```



```
int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Champ de saisie");

    panel = HeCreatePanel (princ);

    HeCreateMessageP (panel, "Texte:", TRUE);

    text = HeCreateText (panel);
    HeSetTextVisibleLen (text, 20);
    HeSetTextNotifyProc (text, entree_proc);

    HeCreateButtonP (panel, "lit valeur", butt_proc, NULL);
    HeCreateButtonP (panel, "Quit", butt_proc, NULL);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

### 5.2.2 Saisie d'un entier

On peut se servir d'un widget Text pour saisir un entier. Dans l'exemple suivant on a rajouté 4 petits boutons "–", "-", "+" et "++" qui incrémentent la valeur de -10, -1, 1 et 10 respectivement :

```
/* examples/panel/entier.c */

#include <helium.h>

He_node *princ, *panel, *text;

void entree_proc (He_node *hn)
{
    int val = atoi(HeGetTextValue(hn));

    printf ("L'entier lu est %d\n", val);
}

void butt_proc (He_node *hn)
{
    char *nom = HeGetButtonLabel(hn), bla[100];
    int  val  = atoi(HeGetTextValue(text));

    if (!strcmp (nom, "--")) val -= 10;
    else if (!strcmp (nom, "-")) val --;
    else if (!strcmp (nom, "+")) val ++;
    else if (!strcmp (nom, "++")) val += 10;

    sprintf (bla, "%d", val);
    HeSetTextValue (text, bla);

    entree_proc(text);
}

int main (int argc, char *argv[])
{
```

```
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Saisie d'un entier");

    panel = HeCreatePanel (princ);

    HeCreateMessageP (panel, "Entier:", TRUE);

    text = HeCreateText (panel);
    HeSetTextVisibleLen (text, 8);
    HeSetTextNotifyProc (text, entree_proc);
    HeSetTextValue (text, "100");

    HeCreateButtonP (panel, "--", butt_proc, NULL);
    HeCreateButtonP (panel, "-",  butt_proc, NULL);
    HeCreateButtonP (panel, "+",  butt_proc, NULL);
    HeCreateButtonP (panel, "++", butt_proc, NULL);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

### 5.2.3 Completion d'un nom de fichier

Lorsqu'un champ de saisie sert à donner un nom de fichier, on peut demander à Helium d'essayer de compléter tout seul le nom du fichier en faisant

```
HeSetTextCompletion (text, TRUE);
```

Chaque fois que l'on tape un caractère à la fin de la ligne, Helium parcourt le système de fichier, regarde si le chemin est bon, et tente de compléter le nom du fichier ou du répertoire. Si la solution est unique, Helium rajoute tout le reste du mot. Si la solution est multiple, Helium rajoute la partie commune. La partie rajoutée est sélectionnée, si bien que lorsque vous décidez d'écrire autre chose, ce que vous écrivez écrase immédiatement la sélection. L'exemple suivant permet de tester la completion :

```
/* examples/panel/completion.c */

#include <helium.h>

He_node *princ, *panel, *text;

void entree_proc (He_node *hn)
{
    printf ("Nom de fichier lu : \"%s\"\n", HeGetTextValue (hn));
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Complétion");

    panel = HeCreatePanel (princ);
```



```
    HeCreateMessageP (panel, "Entrez un chemin de fichier :", TRUE);
    HeSetPanelLayout (panel, HE_LINE_FEED);

    text = HeCreateText (panel);
    HeSetTextVisibleLen (text, 50);
    HeSetTextNotifyProc (text, entree_proc);
    HeSetTextCompletion (text, TRUE);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

### 5.2.4 Filtrage du clavier

On peut attacher une callback KbdProc qui est appelée chaque fois qu'une touche est enfoncée ou relachée, avec

```
HeSetTextKbdProc (text, clavier_proc);
```

Le prototype de la callback est

```
int clavier_proc (He_node *hn, He_event *hev);
```

On reçoit chaque fois un string (et non un caractère, car une touche peut être programmée pour générer un string), et le code symbolique de la touche pressée ou relachée ; le tout est stocké dans la variable `hev`. La callback doit renvoyer `TRUE` ou `FALSE` ; si elle renvoie `FALSE`, l'évènement est intercepté : cela veut dire que lorsque la touche est enfoncée, le string reçu n'est pas inséré dans le texte du widget (cela n'a aucun effet lorsque la touche est relachée).

Dans l'exemple suivant, on affiche le string reçu ainsi que le code symbolique des touches pressées, et on filtre la lettre 'a' :

```
/* examples/panel/filtre.c */

#include <helium.h>

He_node *princ, *panel, *text;

int clavier_proc (He_node *hn, He_event *hev)
{
    switch (hev->type) {
        case KeyPress   : printf ("Touche enfoncée"); break;
        case KeyRelease : printf ("Touche relachée"); break;
    }
    printf ("   symbole XK_%s   string \"%s\"   longueur %d\n",
        XKeysymToString(hev->sym), hev->str, hev->len);

    if (strcmp(hev->str, "a") == 0) return FALSE;
    return TRUE;
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);
```

```
    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Filtre du clavier");

    panel = HeCreatePanel (princ);

    HeCreateMessageP (panel, "Tapez un mot avec un 'a' :", TRUE);
    HeSetPanelLayout (panel, HE_LINE_FEED);

    text = HeCreateText (panel);
    HeSetTextVisibleLen (text, 50);
    HeSetTextKbdProc (text, clavier_proc);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

Les touches [Tab] et [Shift][Tab] permettent de passer le focus du clavier de widget en widget. Lorsque le focus arrive sur un widget Text de cette façon, le contenu du Text est entièrement sélectionné ; de la sorte, vous pouvez remplir une suite de champs sans à chaque fois tout sélectionner avec un triple clic.

Remarque : les évènements [Tab] et [Shift][Tab] sont filtrés au niveau du Panel (c'est lui qui réparti les évènements dans les widgets dont il est propriétaire) et n'arrivent dans la KbdProc que dans le widget Text destination et lorsque la touche est enfoncée. De la sorte on peut aussi filtrer ces évènement dans la KbdProc.

## 5.3 Boutons à cocher

### 5.3.1 Principe

Le widget Toggle permet de créer un bouton à bascule, qui peut être dans un état allumé ou éteint. Le widget Toggle peut prendre 3 aspects différents, aspect que l'on choisit à la création avec

```
He_node *tog;
tog = HeCreateToggle (panel, nature);
HeSetToggleLabel (tog, "Nom du toggle");
```

où `nature` est l'une des trois constantes `HE_LED`, `HE_CHECK` et `HE_RADIO`. On peut être allerté chaque fois que l'utilisateur clique sur un Toggle, en lui attachant une callback NotifyProc :

```
HeSetToggleNotifyProc (tog, etat_proc);
```

Le prototype de la callback est

```
void etat_proc (He_node *hn);
```

où `hn` est le Toggle. Pour connaitre l'état du Toggle, on utilise



```
int HeGetToggleValue (tog);
```

et pour le modifier on utilise

```
HeSetToggleValue (tog, value);
```

où value est TRUE pour allumé, FALSE pour éteint. Voici un exemple qui montre les trois catégories de Toggle et affiche l'état d'un Toggle lorsqu'on clique dessus :

```
/* examples/panel/toggle.c */

#include <helium.h>

He_node *princ, *panel, *tmp;

void etat_proc (He_node *hn)
{
    char *nom = HeGetToggleLabel (hn);
    int  etat = HeGetToggleValue (hn);
    printf ("%s : %s\n", nom, etat ? "allumé" : "éteint");
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Toggle");

    panel = HeCreatePanel (princ);
    HeSetPanelLayout (panel, HE_VERTICAL);

    tmp = HeCreateToggle (panel, HE_LED);
    HeSetToggleLabel (tmp, "Toggle HE_LED");
    HeSetToggleNotifyProc (tmp, etat_proc);

    tmp = HeCreateToggle (panel, HE_CHECK);
    HeSetToggleLabel (tmp, "Toggle HE_CHECK");
    HeSetToggleNotifyProc (tmp, etat_proc);

    tmp = HeCreateToggle (panel, HE_RADIO);
    HeSetToggleLabel (tmp, "Toggle HE_RADIO");
    HeSetToggleNotifyProc (tmp, etat_proc);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

### 5.3.2 Regroupement

Les 3 catégories de Toggle ont été créées dans un but précis :
- ▷ HE_LED est pour un choix isolé ;
- ▷ HE_CHECK est pour un ensemble de choix, où on peut allumer plusieurs Toggle en même temps ;
- ▷ HE_RADIO est pour un ensemble de choix, où un seul des Toggle peut être allumé à la fois.

Pour les Toggle de catégorie HE_RADIO, il faut dire à Helium quels sont les Toggle à regrouper dans un ensemble, pour qu'il puisse automatiquement éteindre un Toggle lorsqu'un autre est allumé. Helium mémorise un tel ensemble sous forme d'une liste circulaire, chaque Toggle du groupe pointant sur le suivant. Pour racrocher un Toggle ou un groupe à autre Toggle ou à un autre groupe, on utilise

```
HeGroupToggle (h1, h2);
```

où l'un des Toggle `h1` ou `h2` (ça n'a pas d'importance) désigne un membre de chaque groupe ; la seule contrainte est que les deux groupes soient distincts avant l'appel. On peut isoler un Toggle de groupe avec

```
HeUngroupToggle (hn);
```

(c'est fait automatiquement lorsqu'on détruit un Toggle). Le fait de regrouper des Toggle n'a aucun effet sur leur état : il faut donc veiller à ce qu'ils soient tous à `FALSE` (sauf éventuellement l'un deux) quand on les groupe (Ils sont à `FALSE` par défaut à la création). On peut ensuite utiliser les deux fonctions de haut niveau suivantes :

```
He_node *HeGetToggleLighted (He_node *hn);
void HeSetToggleLighted (He_node *hn);
```

La première renvoie le toggle du groupe qui est allumé, ou `NULL` si aucun n'est allumé ; le second allume le Toggle `hn`, éteint celui qui était allumé et appelle la callback NotifyProc de chacun des deux Toggle.

Remarque : on peut utiliser ces fonctions sur les 3 catégories de Toggle (et même grouper des HE_LED et HE_CHECK) ; on peut en particulier utiliser `HeSetToggleLighted` à la place de `HeSetToggleValue` si on veut que la callback NotifyProc soit appelée.

Dans l'exemple suivant, on crée un groupe de HE_RADIO et on allume le premier, et dans la callback on affiche l'état des Toggle :

```
/* examples/panel/radio.c */

#include <helium.h>

He_node *princ, *panel, *tog1, *tog2, *tog3;

void etat_proc (He_node *hn)
{
    char *nom = HeGetToggleLabel (hn);
    int  etat = HeGetToggleValue (hn);
    printf ("%s : %s\n", nom, etat ? "allumé" : "éteint");
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Toggle radio");

    panel = HeCreatePanel (princ);
    HeCreateMessageP (panel, "Radio :", TRUE);
```



```
    tog1 = HeCreateToggle (panel, HE_RADIO);
    HeSetToggleLabel (tog1, "Choix 1");
    HeSetToggleNotifyProc (tog1, etat_proc);

    tog2 = HeCreateToggle (panel, HE_RADIO);
    HeSetToggleLabel (tog2, "Choix 2");
    HeSetToggleNotifyProc (tog2, etat_proc);
    HeGroupToggle (tog1, tog2);

    tog3 = HeCreateToggle (panel, HE_RADIO);
    HeSetToggleLabel (tog3, "Choix 3");
    HeSetToggleNotifyProc (tog3, etat_proc);
    HeGroupToggle (tog2, tog3);

    HeSetToggleLighted (tog1);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

### 5.3.3 Raccourcis

La fonction suivante crée un Toggle de nature HE_LED, lui affecte un label, une callback NotifyProc et une valeur :

```
He_node *HeCreateToggleLedP (He_node *owner, char *label,
          He_notify_proc proc, int value)
```

Il en va de même pour cette fonction qui crée un Toggle de nature HE_CHECK :

```
He_node *HeCreateToggleCheckP (He_node *owner, char *label,
          He_notify_proc proc, int value)
```

Enfin, cette fonction crée un ensemble de Toggles de nature HE_RADIO (à choix unique). Lors de la création d'un groupe, il faut passer `NULL` au paramètre `group` pour le premier élément, puis aux éléments suivant il faut passer le premier élément au paramètre `group`. Le premier élément du groupe est automatiquement mis à vrai, les autres à faux. Ont peut alors utiliser `HeSetToggleLighted` pour changer le Toggle qui est allumé.

```
He_node *HeCreateToggleRadioP (He_node *owner, char *label,
          He_notify_proc proc, He_node *group)
```

On utilise souvent des Toggle pour griser certaines parties de l'application ; voici un exemple récapitulatif :

```
/* examples/panel/griser.c */

#include <helium.h>

He_node *princ, *panel, *tog1, *tog2, *tog3, *tog4, *tog5,
    *mess1, *mess2, *mess3, *mess4, *mess5, *text1, *text2;

void tog1_proc (He_node *hn)
```

```
{
    int etat = HeGetToggleValue (hn);
    HeSetActive (mess2, etat);
    HeSetActive (tog2, etat);
    HeSetActive (tog3, etat);
}

void tog2_proc (He_node *hn)
{
    int etat = HeGetToggleValue (hn);
    HeSetActive (mess1, etat);
    HeSetActive (tog1, etat);
}

void tog3_proc (He_node *hn)
{
    int etat = HeGetToggleValue (hn);
    HeSetActive (mess3, etat);
    HeSetActive (tog4, etat);
    HeSetActive (tog5, etat);
}

void tog4_proc (He_node *hn)
{
    int etat = HeGetToggleValue (hn);
    HeSetActive (mess4, etat);
    HeSetActive (text1, etat);
}

void tog5_proc (He_node *hn)
{
    int etat = HeGetToggleValue (hn);
    HeSetActive (mess5, etat);
    HeSetActive (text2, etat);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Toggle qui grise");

    panel = HeCreatePanel (princ);

    mess1 = HeCreateMessageP (panel, "Bouton Led   :", TRUE);
    tog1 = HeCreateToggleLedP (panel, "Ligne dessous", tog1_proc, TRUE);
    HeSetPanelLayout (panel, HE_LINE_FEED);

    mess2 = HeCreateMessageP (panel, "Bouton Check :", TRUE);
    tog2 = HeCreateToggleCheckP (panel, "Dessus" , tog2_proc, TRUE);
    tog3 = HeCreateToggleCheckP (panel, "Dessous", tog3_proc, TRUE);
    HeSetPanelLayout (panel, HE_LINE_FEED);

    mess3 = HeCreateMessageP (panel, "Bouton Radio :", TRUE);
    tog4 = HeCreateToggleRadioP (panel, "Gauche", tog4_proc, NULL);
    tog5 = HeCreateToggleRadioP (panel, "Droite", tog5_proc, tog4);
    HeSetPanelLayout (panel, HE_LINE_FEED);

    mess4 = HeCreateMessageP (panel, "Texte 1 :", TRUE);
    text1 = HeCreateText (panel);
    HeSetTextVisibleLen (text1, 10);
    HeSetTextValue (text1, "Bonjour");
    mess5 = HeCreateMessageP (panel, "Texte 2 :", TRUE);
    text2 = HeCreateText (panel);
    HeSetTextVisibleLen (text2, 10);
```



```
    HeSetTextValue (text2, "Bonsoir");

    /* On force l'appel des callback pour griser */
    HeSetToggleLighted (tog4);
    HeSetToggleLighted (tog5);

    HeFit(panel);
    HeFit(princ);

    return HeMainLoop (princ);
}
```

# Chapitre 6

# Macro widgets

Ce sont des fenêtres prêtes à l'emploi que Helium génère à la demande, telles que boîte de dialogue ou sélecteur de fichier.

## 6.1 Boîte de dialogue

Une fenêtre de dialogue est une fenêtre temporaire avec un titre, quelques lignes de texte, une ligne horizontale pour faire joli et un ou plusieurs boutons, tels que Ok, Cancel, etc.

Le widget SimpleDialog permet de créer et de gérer une fenêtre de dialogue de la façon la plus simple possible.

Un appel à `HeSimpleDialog()` crée la fenêtre avec son texte et ses boutons, l'affiche et attend qu'un bouton soit pressé ; dès qu'un bouton est pressé, la fenêtre est automatiquement cachée et détruite et la callback du bouton est appelée.

Une seule fenêtre de dialogue peut être affichée à la fois ; un nouvel appel à HeSimpleDialog remplace la fenêtre précédente.

La syntaxe est la suivante : on passe une suite d'arguments `HE_DIALOG_` à la fonction, et cette suite est terminée par 0.

```
HeSimpleDialog (
    HE_DIALOG_BELL,
    HE_DIALOG_TITLE,   "Le titre",
    HE_DIALOG_MESSAGE, "Attention, le fichier",
    HE_DIALOG_QUOTED,  file_name,
    HE_DIALOG_MESSAGE, "ne peut etre ouvert",
    HE_DIALOG_BUTTON,  "Reessayer"
    HE_DIALOG_BUTTOK,  "Annuler"
    HE_DIALOG_PROC,    my_dialog_proc
    HE_DIALOG_DATA,    my_data,
    0);
```

La présence de `HE_DIALOG_BELL` fait retentir un bip à l'affichage de la fenêtre. Par `HE_DIALOG_TITLE` on donne un titre à la fenêtre. Chaque `HE_DIALOG_MESSAGE` rajoute une





ligne de texte. `HE_DIALOG_QUOTED` joue le même rôle, mais en rajoutant des "" autour du texte (très utile pour le nom d'un fichier). Chaque `HE_DIALOG_BUTTON` rajoute un bouton. On peut préciser quel est le bouton qui est validé par défaut lorsqu'on appuie sur [Entrée] : il suffit de le créer avec `HE_DIALOG_BUTTOK`. La callback de `HE_DIALOG_PROC` est attachée à tous les boutons. On peut enfin définir avec `HE_DIALOG_DATA` une variable `my_data` qui sera transmise en paramètre dans la callback.

Le prototype de la callback my_dialog_proc est

```
void my_dialog_proc(char *name, void *my_data);
```

Tous les arguments passés à HeSimpleDialog sont optionnels ; s'il n'y a pas de `HE_DIALOG_PROC`, le clic sur un bouton fait tout simplement disparaître la boîte de dialogue.

Attention, l'appel à HeSimpleDialog est non bloquant ; c'est-à-dire que la fenêtre est immédiatement créée et affichée, puis le programme continue sans pouvoir attendre la réponse ; celle-ci sera connue lors de l'appel de la callback.

Dans l'exemple suivant, on crée une boîte de dialogue dans la fonction `ask_quit()`. Cette fonction est la callback du bouton Quit. On détourne aussi la fermeture de la fenêtre via le Window Manager en attachant une CloseProc au Frame.

```
/* examples/macro/dialog1.c */

#include <helium.h>

He_node *princ, *panel;


void dialog_proc (char *name, void *my_data)
{
    if (!strcmp (name, "Oui")) HeQuit(0);
}

void ask_quit (He_node *hn)
{
    HeSimpleDialog (
        HE_DIALOG_BELL,
        HE_DIALOG_TITLE,   "Attention",
        HE_DIALOG_MESSAGE, "Etes-vous certain",
        HE_DIALOG_MESSAGE, "de vouloir quitter ?",
        HE_DIALOG_BUTTON,  "Oui",
        HE_DIALOG_BUTTOK,  "Annuler",
        HE_DIALOG_PROC,    dialog_proc,
        0);
 }

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Boite de dialogue");
    HeSetFrameCloseProc (princ, ask_quit);

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Quit", ask_quit, NULL);

    return HeMainLoop (princ);
}
```

# Chapitre 7

# Dessins en Xlib

Dans ce chapitre, nous présentons le widget Canvas, qui permet de faire des dessins en Xlib et de réagir au clavier et à la souris.

Vous n'avez pas besoin de connaître déjà Xlib pour aborder ce chapitre : les notions nécessaires sont présentées au fur et à mesure.

## 7.1 Courte présentation de Xlib

Le système X-Window (ou X11) est le standard d'affichage graphique du monde Unix ; c'est lui qui permet entre autre l'affichage distant, avec un modèle client-serveur et un protocole réseau.

X11 repose sur un modèle hiérarchique de zones rectangulaires, appelées "Window" :

1. Chaque Window peut être inclus dans un autre Window (son propriétaire) et peut inclure d'autre Window (ses fils) ; des Window qui ont le même propriétaire sont des frères (*sibling*).
2. L'écran lui-même est un Window (le Root Window) qui contient tous les Window.
3. Un Window peut être devant ou derrière un Window frère ; le Window qui est devant masque tout ou partie de celui qui est derrière.
4. Tout dessin fait dans un Window est "coupé" automatiquement, c'est à dire que seule la partie du dessin à l'intérieur du Window est tracé.
5. Un Window peut être apparent ("mappé") ou caché ; les ordres de dessins faits dans un Window non mappé sont ignorés.
6. Chaque évènement (clavier, souris, etc) est adressé à un Window en particulier.
7. Un Window ne mémorise en principe pas son contenu : chaque fois qu'il doit être réaffiché, il reçoit un évènement "Expose", lui demandant de redessiner tout son contenu.

On voit donc que le système X11 est relativement "bas niveau" mais qu'il est spécialement pensé pour écrire par dessus des Window-Manager et des Toolkits (tel que Helium).

La librairie standard de X11 est Xlib ; elle comporte des centaines de fonctions et des dizaines de types, pour gérer les Window, les évènements, le clavier et la souris, l'affichage





de dessins, de texte, d'images et de couleurs, etc. Helium en exploite un certain nombre pour le dessin automatique des widgets.

## 7.2 Le widget Canvas

Le widget Canvas que nous présentons dans cette section est un widget dans lequel vous pouvez faire vos propres dessins en Xlib, et où vous pouvez réagir à un certain nombre d'évènements souris et clavier.

En fait, le widget Canvas est tout simplement un Window de Xlib, avec des évènements filtrés par Helium : toutes les fonctions de dessin de Xlib sont utilisables (avec naturellement le coupage automatique), et la gestion des évènements est très simplifiée (mais on a accès aux "vrais" évènements X11 si on le désire).

Pour créer un Canvas on fait

```
He_node *canvas;
canvas = HeCreateCanvas (frame);
```

où `frame` est le Frame qui hébergera le Canvas. Par défaut, le Canvas est placé en 0,0 du Frame, et son aspect est un rectangle blanc, bordé d'un fin trait noir. Pour dessiner dedans, on attache une callback RepaintProc :

```
HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
```

Le prototype de la callback est

```
void canvas_repaint_proc (He_node *hn, Window win);
```

où `hn` est le Canvas et `win` est son Window X11. Chaque fois que le Canvas doit être réaffiché, Helium appelle la RepaintProc en lui passant le Canvas `hn` et le Window `win` dans lequel dessiner. N'importe quelle fonction de dessin de Xlib peut être appelée dans la RepaintProc, comme dans l'exemple suivant :

```
/* examples/canvas/dessins.c */

#include <helium.h>

He_node *princ, *canvas;

void canvas_repaint_proc (He_node *hn, Window win)
{
    printf("Appel de canvas_repaint_proc\n");
    XSetForeground (he_display, he_gc, he_black);
    XDrawLine (he_display, win, he_gc,
        10, 10, 290, 290);
    XDrawRectangle (he_display, win, he_gc,
        10, 10, 280, 280);
    XDrawArc (he_display, win, he_gc,
        10, 10, 280, 280, 0, 360*64);
}

int main (int argc, char *argv[])
{
```

```
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Dessins");

    canvas = HeCreateCanvas (princ);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);

    HeSetWidth (canvas, 300); HeSetHeight (canvas, 300);
    HeFit (princ);

    return HeMainLoop (princ);
}
```

Toutes les fonctions de dessin de Xlib (XDraw*, XFill*, XSet*) sont documentées avec la commande Unix **man** ; les plus courantes sont expliquées dans la section « 7.4 Fonctions de dessins de Xlib ».

Les variables `he_display`, `he_gc`, `he_black` et `he_white` sont des variables globales de Helium :

- ▷ `he_display` mémorise les paramètres de l'écran sur lequel on affiche (local ou distant, nombre de plans, définition, etc) : on doit le donner à presque toutes les fonctions de Xlib.
- ▷ `he_gc` est un GC (Graphical Context), qui mémorise tous les attributs de dessin (couleur, mais aussi épaisseur, pointillé, etc ; en tout il y en a 23). Il évite d'avoir à passer tous ces paramètres aux fonctions de dessin de Xlib.
- ▷ `he_black` est l'indice de la couleur noire, `he_white` est l'indice de la couleur blanche (cf section « 7.3 Dessins en couleur »). Le noir et le blanc sont les seules couleurs qui existent au départ.

`win` est l'identificateur du Window dans lequel on va dessiner (ici, le Window du Canvas). On le passe donc aux fonctions de dessin.

Les coordonnées dans le Canvas varient entre `0,0` (coin en haut à gauche) et `width-1,height-1`. Le repère est orienté vers le bas, en "coordonnées souris". Le Canvas fait le coupage des dessins qui sont faits ; autrement dit, aucun dessin ne peut dépasser le Canvas, et on n'a pas besoin de faire ce genre de test.

## 7.3 Dessins en couleur

Dans cette section, on explique comment faire des dessins en couleur dans un Canvas avec Xlib. Dans l'exemple précédent `dessins.c`, la RepaintProc commence par ces lignes :

```
XSetForeground (he_display, he_gc, he_black);
XDrawLine (he_display, win, he_gc, 10, 10, 290, 290);
```

On demande à Xlib de fixer la couleur de dessin à `he_black` ; tous les dessins qui suivent sont faits dans cette couleur, jusqu'au prochain appel à `XSetForeground`.

La variable de couleur transmise à `XSetForeground` est un index de couleur, codée sur un `int`. Pour obtenir un index pour une couleur `R,G,B` ou une couleur `H,S,V` on appelle



```
int HeAllocRgb (int R, int G, int B, int defaut);
int HeAllocHsv (int H, int S, int V, int defaut);
```

où `defaut` est l'index de la couleur par défaut si l'appel échoue. L'action de ces fonctions dépend du mode d'affichage du serveur X11.

En mode PseudoColor (jusqu'à 8 plans), le nombre de couleurs simultanées est limité (256 pour 8 plans), et chaque allocation "consomme" une case de couleur ; lorsque plus aucune case n'est disponible, les appels à `HeAlloc...` échouent. (Remarque : les couleurs utilisées sont rendues au système à la terminaison du programme).

En mode TrueColor (15, 16, 24 ou 32 plans), l'index est une simple combinaison calculée entre les bits r, g et b ; l'appel ne peut échouer.

En général on alloue donc les couleur nécessaires au début du programme et on les mémorise dans des variables globales ou un tableau.

Dans l'exemple suivant, on alloue 4 couleurs et on dessine un carré.

```
/* examples/canvas/couleur.c */

#include <helium.h>

He_node *princ, *canvas;
int rouge, vert, bleu, gris;

void init_couleurs ()
{
    rouge = HeAllocRgb (255, 0, 0, he_black);
    vert  = HeAllocRgb (0, 255, 0, he_black);
    bleu  = HeAllocRgb (0, 0, 255, he_black);
    gris  = HeAllocRgb (150, 150, 150, he_black);
}

void dessin_ligne (Window win, int x1, int y1, int x2, int y2, int coul)
{
    XSetForeground (he_display, he_gc, coul);
    XDrawLine (he_display, win, he_gc, x1, y1, x2, y2);
}

void canvas_repaint_proc (He_node *hn, Window win)
{
    dessin_ligne (win, 50, 50, 250, 50, rouge);
    dessin_ligne (win, 250, 50, 250, 250, vert);
    dessin_ligne (win, 250, 250, 50, 250, bleu);
    dessin_ligne (win, 50, 250, 50, 50, gris);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "carré en couleur");

    canvas = HeCreateCanvas (princ);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
    init_couleurs ();

    HeSetWidth (canvas, 300); HeSetHeight (canvas, 300);
    HeFit (princ);
```

```
        return HeMainLoop (princ);
    }
```

Remarque : l'appel à `HeAllocHsv` est plus coûteux que `HeAllocRgb`, car il passe par une conversion de HSV en RGB avant d'appeler `HeAllocRgb`. Dans la suite on parlera des fonctions fournies par Helium pour convertir des couleurs.

## 7.4 Fonctions de dessins de Xlib

Toutes les fonctions de dessin de Xlib reçoivent toujours les mêmes trois premiers paramètres, qui sont : `he_display`, `win`, `he_gc` (le Display, qui mémorise les caractéristiques de l'écran sur lequel afficher ; le Window dans lequel dessiner ; les attributs de dessin courants, qui évitent d'avoir à passer un trop grand nombre de paramètres aux fonctions de dessin de Xlib).

Les coordonnées que l'on donnent sont toujours par rapport au coin en haut à gauche du Canvas (0,0), avec le repère vers le bas (c'est à dire en coordonnées souris). Enfin on rappelle que le coupage des dessins est automatique dans le Window.

Pour dessiner un point `x1,y1` on fait

```
XDrawPoint (he_display, win, he_gc, x1, y1);
```

Pour dessiner une ligne d'un point `x1,y1` à un point `x2,y2` on fait

```
XDrawLine (he_display, win, he_gc, x1, y1, x2, y2);
```

Pour dessiner un rectangle de coin en haut à gauche `x1,y1` et de coin en bas à droite `x2,y2` on fait

```
XDrawRectangle (he_display, win, he_gc, x1, y1, x2-x1, y2-y1);
```

En effet, les deux derniers paramètres sont la largeur et la hauteur du rectangle. On peut aussi tracer un rectangle plein avec XFillRectangle. Attention, les deux derniers paramètres ne sont pas tout à fait les mêmes, il faut faire :

```
XFillRectangle (he_display, win, he_gc,
                x1, y1, x2-x1+1, y2-y1+1);
```

Pour tracer un cercle ou une ellipse, on donne les coordonnées des coins du rectangle qui contient le cercle ou l'ellipse (la boîte englobante), soit `x1,y1` le coin en haut à gauche et `x2,y2` le coin en bas à droite :

```
XDrawArc (he_display, win, he_gc,
          x1, y1, x2-x1, y2-y1, 0, 360*64);
```

L'avant dernier paramètre correspond à l'angle de départ du tracé, en 64èmes de degrés, par rapport à l'horizontale de droite, dans le sens inverse des aiguilles d'une montre. Le dernier paramètre est l'angle de tracé par rapport à l'angle de départ (et pas par rapport à l'horizontale), toujours en 64èmes de degrés. On peut donc tracer un morceau d'arc au lieu de tracer une ellipse pleine. On peut aussi tracer un cercle plein, une ellipse pleine, ou un camembert plein, avec :



```
XFillArc (he_display, win, he_gc,
          x1, y1, x2-x1+1, y2-y1+1, 0, 360*64);
```

Attention, comme pour XFillRectangle, il faut rajouter 1 à la hauteur et la largeur de la boîte englobante.

On peut aussi afficher du texte dans un Canvas : les fonctions sont décrites dans la section « 7.11 Afficher du texte ».

Les autres fonctions de dessin de Xlib sont documentées dans les pages de man ; si j'en oublie, prévenez-moi ! Les voici :

`XDrawArcs`, `XDrawLines`, `XDrawPoints`, `XDrawRectangles`, `XDrawSegments`, `XFillArcs`, `XFillPolygon`, `XFillRectangles`, `XSetArcMode`, `XSetLineAttributes`, `XSetDashes`, `XSetFillRule`, `XSetFillStyle`, `XSetClipMask`, `XSetClipRectangles`, `XSetRegion`, `XUnionRectWithRegion`, `XSetClipOrigin`, `XSetTile`, `XSetStipple`, `XQueryBestSize`, `XQueryBestTile`, `XQueryBestStipple`, `XSetFunction`, `XSetPlaneMask`, `XCopyPlane`, `XCopyArea`, `XCreateGC`, `XChangeGC`, `XGetGCValues`.

## 7.5 Évènements dans un Canvas

Pour recevoir des évènements dans un Canvas, il suffit d'attacher une callback EventProc avec

```
HeSetCanvasEventProc (canvas, canvas_event_proc);
```

le prototype de la callback est

```
void canvas_event_proc (He_node* hn, He_event *hev);
```

où `hn` est le Canvas et `hev` contient les caractéristiques de l'évènement. Cette callback est appelée par Helium chaque fois que l'un des évènements X11 suivants arrive au Canvas :

- ▷ `EnterNotify` : la souris rentre dans le Canvas
- ▷ `LeaveNotify` : la souris sort du Canvas
- ▷ `KeyPress` : une touche est enfoncée
- ▷ `KeyRelease` : une touche est relachée
- ▷ `ButtonPress` : un bouton de la souris est enfoncé
- ▷ `ButtonRelease` : un bouton de la souris est relaché
- ▷ `MotionNotify` : la souris a bougé

Le type `He_event` est défini dans include/types.h :

```
typedef struct He_event_st {
    int     type,           /* Type d'évènement = xev->type */
            sx, sy,         /* Coords souris / window */
            sb;             /* Bouton souris filtré : 0,1,2,3 */
    Time    time;           /* Temps en milli secondes */
    Window  win;            /* Le Window X11 du widget */
    XEvent *xev;            /* Evènement X11 complet */
```

```
    char    str[256];       /* Buffer lu au clavier */
    int     len;            /* Nombre de char dans str */
    KeySym  sym;            /* Symbole de la touche pressée */
} He_event;
```

Dans l'exemple suivant on affiche tous les évènements :

```
/* examples/canvas/event.c */

#include <helium.h>

He_node *princ, *canvas;

void canvas_repaint_proc (He_node *hn, Window win)
{
    printf ("canvas_repaint_proc\n");
}

void canvas_event_proc (He_node *hn, He_event *hev)
{
    printf ("canvas_event_proc ");

    switch (hev->type) {
        case EnterNotify :
            printf ("EnterNotify   %d,%d %d\n", hev->sx, hev->sy, hev->sb);
            break;
        case LeaveNotify :
            printf ("LeaveNotify   %d,%d %d\n", hev->sx, hev->sy, hev->sb);
            break;
        case ButtonPress :
            printf ("ButtonPress   %d,%d %d\n", hev->sx, hev->sy, hev->sb);
            break;
        case ButtonRelease :
            printf ("ButtonRelease %d,%d %d\n", hev->sx, hev->sy, hev->sb);
            break;
        case MotionNotify :
            printf ("MotionNotify  %d,%d %d\n", hev->sx, hev->sy, hev->sb);
            break;
        case KeyPress :
            printf ("KeyPress: \"%s\" keysym = XK_%s len = %d\n",
                hev->str, XKeysymToString(hev->sym), hev->len);
            break;
        case KeyRelease :
            printf ("KeyRelease\n");
            break;
    }
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Évènements du canvas");

    canvas = HeCreateCanvas (princ);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
    HeSetCanvasEventProc (canvas, canvas_event_proc);

    HeSetWidth (canvas, 300); HeSetHeight (canvas, 300);
    HeFit (princ);

    return HeMainLoop (princ);
}
```



Remarques :

- ▷ On peut aussi dessiner dans l'EventProc (le Window est `hev->win`) si c'est pour rajouter des dessins. Il faut prévoir de mémoriser ces dessins pour que la RepaintProc puisse éventuellement tout redessiner.
- ▷ On peut provoquer un réaffichage du Canvas dans l'EventProc : voir la section « 7.8 Provoquer un réaffichage ».

## 7.6 Canvas redimensionné

A la création du Canvas, et chaque fois que la taille du Canvas change, la callback ResizeProc du Canvas est automatiquement appelée (si définie), juste avant que la RepaintProc ne soit appelée à son tour. Pour attacher une callback ResizeProc au Canvas on fait

```
HeSetCanvasResizeProc (canvas, canvas_resize_proc);
```

Le prototype de la callback est

```
void canvas_resize_proc (He_node *hn, int width, int height);
```

où `hn` est le Canvas, `width` et `height` sont les nouvelles dimensions.

Un bon endroit où changer la taille d'un Canvas est la ResizeProc d'un Frame. On a alors le schéma suivant : l'utilisateur redimensionne une fenêtre à la souris, ce qui déclenche l'appel de la ResizeProc du Frame. Dans cette callback on change la taille du Canvas, ce qui déclenche l'appel de la ResizeProc puis de la RepaintProc du Canvas.

Dans l'exemple suivant on crée une fenêtre avec deux Panels et un Canvas. Dans le Panel 1 on place un bouton Quit, et dans le Panel 2 on place un Message ; entre les deux Panel on place le Canvas, qui doit occuper toute la place disponible. Lorsqu'on redimensionne la fenêtre, les Panels et le Canvas sont mis à la nouvelle largeur ; le Panel 2 est abaissé, et la hauteur du Canvas est ajustée pour occuper tout l'espace disponible. Depuis la ResizeProc du Canvas on affiche chaque fois la nouvelle taille du Canvas dans le Message.

```
/* examples/canvas/taille.c */

#include <helium.h>

He_node *princ, *panel1, *panel2, *canvas, *mess1;

void princ_resize_proc (He_node *hn, int width, int height)
{
    /* ajuste largeurs panels */
    HeSetWidth (panel1, width);
    HeSetWidth (panel2, width);

    /* met le panel2 en bas */
    HeJustify (panel2, NULL, HE_BOTTOM);

    /* le canvas prend toute la place disponible */
    HeExpand (canvas, panel2, HE_BOTTOM);
    HeExpand (canvas, NULL, HE_RIGHT);
}
```

```
void canvas_resize_proc (He_node *hn, int width, int height)
{
    char bla[100];
    sprintf (bla, "Canvas %d x %d", width, height);
    HeSetMessageLabel (mess1, bla);
}

void butt_proc (He_node *hn)
{
    HeQuit(0);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Canvas redimensionnée");
    HeSetFrameResizeProc (princ, princ_resize_proc);

    panel1 = HeCreatePanel (princ);
    HeCreateButtonP (panel1, "Quit", butt_proc, NULL);
    HeFit(panel1);

    canvas = HeCreateCanvas (princ);
    HeJustify (canvas, panel1, HE_TOP);
    HeSetWidth (canvas, 300);
    HeSetHeight (canvas, 300);
    HeSetCanvasResizeProc (canvas, canvas_resize_proc);

    panel2 = HeCreatePanel (princ);
    HeJustify (panel2, canvas, HE_TOP);

    mess1 = HeCreateMessageP (panel2, NULL, FALSE);
    HeFit(panel2);

    HeFit (princ);

    return HeMainLoop (princ);
}
```

Remarques :

▷ Le Canvas est placé par défaut en 0,0 à la création, il faut donc le placer explicitement sous le Panel 1, sinon il va le recouvrir.

▷ Par défaut, le canvas a un bord extérieur de 1 pixel (de couleur noire) ; donc la largeur totale est `1+width+1` et la hauteur totale est `1+height+1`. Voir HeSetBorder() plus loin dans le tutorial pour changer la largeur du bord extérieur.

La ResizeProc du Canvas est un bon endroit pour recalculer par exemple les coordonnées de points de contrôle, stockées dans un vecteur, pour mettre un dessin à la nouvelle échelle du Canvas. Dans l'exemple suivant, on fait un dessin à la souris et on le mémorise ; on le remet à l'échelle chaque fois qu'on redimensionne la fenêtre ; lorsqu'un bouton est pressé on réinitialise le dessin, puis on dessine en tirant la souris avec un bouton enfoncé.

```
/* examples/canvas/echelle.c */

#include <helium.h>

He_node *princ, *canvas;
int old_width = 1, old_height = 1;

#define SMAX 1000
```



```
int Sx[SMAX], Sy[SMAX], Sn = 0;

void dessin_segment (Window win, int i)
{
    if (i == 0)
         XDrawPoint (he_display, win, he_gc,
            Sx[0], Sy[0]);
    else XDrawLine (he_display, win, he_gc,
            Sx[i-1], Sy[i-1], Sx[i], Sy[i]);
}

void princ_resize_proc (He_node *hn, int width, int height)
{
    HeExpand (canvas, NULL, HE_BOTTOM_RIGHT);
}

void canvas_resize_proc (He_node *hn, int width, int height)
{
    int i;
    for (i = 0; i < Sn; i++) {
        Sx[i] = Sx[i] * width / old_width;
        Sy[i] = Sy[i] * height / old_height;
    }
    old_width = width; old_height = height;
}

void canvas_repaint_proc (He_node *hn, Window win)
{
    int i;
    XSetForeground (he_display, he_gc, he_black);
    for (i = 0; i < Sn; i++)
        dessin_segment (win, i);
}

void canvas_event_proc (He_node *hn, He_event *hev)
{
    switch (hev->type) {
        case ButtonPress :
            HeDrawBg (hn, he_white);
            Sn = 0;
            Sx[Sn] = hev->sx; Sy[Sn] = hev->sy;
            dessin_segment (hev->win, Sn++);
            break;
        case MotionNotify :
            if (hev->sb > 0 && Sn < SMAX) {
                Sx[Sn] = hev->sx; Sy[Sn] = hev->sy;
                dessin_segment (hev->win, Sn++);
            }
            break;
    }
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Mise à l'échelle");
    HeSetFrameResizeProc (princ, princ_resize_proc);

    canvas = HeCreateCanvas (princ);
    HeSetWidth (canvas, 300);
    HeSetHeight (canvas, 300);
    HeSetCanvasResizeProc  (canvas, canvas_resize_proc);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
    HeSetCanvasEventProc   (canvas, canvas_event_proc);
```

```
    HeFit (princ);

    return HeMainLoop (princ);
}
```

## 7.7 Double buffer d'affichage

Le Canvas possède un double buffer d'affichage intégré, qui permet d'éviter des clignotements lors de l'affichage. Par défaut il est désactivé. Pour l'activer, utiliser

```
void HeSetCanvasDBuf (He_node *hn, int val);
```

avec `val = TRUE` (ou `FALSE` pour le désactiver). Pour savoir si un Canvas est actuellement en mode double buffer, consulter

```
int HeGetCanvasDBuf (He_node *hn);
```

Il n'y a strictement rien d'autre à faire, Helium se charge complètement de la gestion des doubles buffers. Une petite contrainte cependant : il ne faut faire des dessins que dans la `RepaintProc`, et il faut utiliser le second paramètre `win` de la `RepaintProc` pour dessiner (c'est soit le `Window` natif en mode non bufférisé, soit le *back buffer* en mode bufférisé).

Comme exemple, voir `demo/grille.c`

## 7.8 Provoquer un réaffichage

On peut appeler directement la fonction qui fait office de RepaintProc pour tout redessiner à un moment donné ; il suffit de lui fournir le Canvas et son Window.

Soit `canvas` un Canvas et `canvas_repaint_proc` sa RepaintProc. Si on veut faire l'appel depuis l'EventProc du Canvas, on écrira

```
canvas_repaint_proc (canvas, hev->win);
```

si on veut faire l'appel depuis la callback d'un bouton, on écrira

```
canvas_repaint_proc (canvas, HeGetWindow(canvas));
```

Attention : il faut être conscient que l'exécution de la RepaintProc peut durer un certain temps, pendant lequel l'affichage peut être bloqué, et certains boutons peuvent êtres "gelés" en position enfoncée par exemple ; cela peut aussi faire clignoter l'affichage en cas d'appels multiples. La solution est simple : il suffit de remplacer l'appel direct de la RepaintProc par :

```
HePostRepaint (canvas);
```



qui provoque un appel (à peine) différé de la RepaintProc par Helium. (En fait, HePostRepaint envoie simplement un évènement Expose au Window du Canvas ; cet évènement est traité une fois qu'on est sorti de la callback d'où on a fait l'appel).

Remarque 1 : lorsque le Canvas reçoit plusieurs Expose très rapprochés, Helium le détecte et ne commande l'appel de la RepaintProc que sur le dernier Expose, pour gagner en fluidité ; donc si vous appelez plusieurs fois à la suite HePostRepaint, un seul RepaintProc sera effectué (lire les commentaires au dessus de "`case Expose :`" dans gui/event.c).

Remarque 2 : lorsque la RepaintProc est appelée suite à un Expose normal, le fond est initialisé à blanc, et donc vous pouvez dessiner de suite sans effacer le fond. Ce n'est pas le cas lorsque le Expose est dû à un HePostRepaint : le fond n'étant pas initialisé à blanc, il peut s'avérer nécessaire de l'effacer avant de faire vos dessins. Pour cela il suffit d'appeler

```
HeDrawBg (canvas, he_white);
```

qui vide et met en blanc le Canvas. On peut appeler HeDrawBg au début de la RepaintProc ou au moment de l'appel de HePostRepaint.

Remarque 3 : si le double buffer d'affichage est activé, il ne faut jamais appeler directement la RepaintProc car on ne connaît pas le *back buffer* ; il faut obligatoirement utiliser HePostRepaint.

L'exemple suivant récapitule ce qui est dit dans cette section. Agrandissez la fenêtre pour ralentir l'affichage, et observez le redessin des boutons ; cliquez plusieurs fois d'affilée très vite sur un bouton ou dans le Canvas, et comptez le nombre de réaffichages effectifs.

```
/* examples/canvas/reaffi.c */

#include <helium.h>

He_node *princ, *panel, *canvas;

void dessin_point (Window win, int x, int y, int coul)
{
    XSetForeground (he_display, he_gc, coul);
    XDrawPoint (he_display, win, he_gc, x, y);
}

void canvas_repaint_proc (He_node *hn, Window win)
{
    int x, y, w = HeGetWidth(hn), h = HeGetHeight(hn);

    printf ("Début RepaintProc\n");

    /* Dessin du fond */
    HeDrawBg (hn, he_white);

    /* Dessin volontairement lent */
    for (y = 0; y < h; y++)
    for (x = 0; x < w; x++)
        if ((w/2-x)*(w/2-x)+(h/2-y)*(h/2-y) < w*h/4)
            dessin_point (win, x, y, he_black);

    printf ("Fin RepaintProc\n");
}

void canvas_event_proc (He_node *hn, He_event *hev)
{
```

```
    switch (hev->type) {
        case ButtonPress :
            if (hev->sb == 1) {
                printf ("Debut appel direct\n");
                canvas_repaint_proc (canvas, hev->win);
                printf ("Fin appel direct\n");
            } else if (hev->sb == 2) {
                printf ("Debut appel différé\n");
                HePostRepaint (canvas);
                printf ("Fin appel différé\n");
            }
            break;
    }
}

void butt1_proc (He_node *hn)
{
    printf ("Debut appel direct\n");
    canvas_repaint_proc (canvas, HeGetWindow(canvas));
    printf ("Fin appel direct\n");
}

void butt2_proc (He_node *hn)
{
    printf ("Debut appel différé\n");
    HePostRepaint (canvas);
    printf ("Fin appel différé\n");
}

void princ_resize_proc (He_node *hn, int width, int height)
{
    HeExpand (canvas, NULL, HE_BOTTOM_RIGHT);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Provoquer un réaffichage");
    HeSetFrameResizeProc (princ, princ_resize_proc);

    panel = HeCreatePanel (princ);
    HeSetPanelLayout (panel, HE_VERTICAL);

    HeCreateButtonP (panel, "Appel direct", butt1_proc, NULL);
    HeCreateButtonP (panel, "Appel différé", butt2_proc, NULL);
    HeCreateMessageP (panel, "Bouton souris 1 : appel direct", FALSE);
    HeCreateMessageP (panel, "Bouton souris 2 : appel différé", FALSE);
    HeFit(panel);

    canvas = HeCreateCanvas (princ);
    HeSetY (canvas, HeGetHeight(panel) + 2);
    HeSetWidth (canvas, 500);
    HeSetHeight (canvas, 500);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
    HeSetCanvasEventProc (canvas, canvas_event_proc);

    HeFit (princ);

    return HeMainLoop (princ);
}
```



## 7.9 Les XImages

Lorsqu'on veut afficher une image dans un Canvas, on peut faire un affichage point par point ; le problème est que cette méthode est extrêmement lente. On présente dans cette section comment réaliser un affichage instantané.

Une XImage est une "mémoire image", qui peut être plaquée à l'écran. C'est une structure de donnée spéciale de Xlib, comprenant un tableau des couleurs des points, codées sous leur forme finale, c'est-à-dire sous la forme de la mémoire graphique.

En premier lieu on déclare une XImage rectangulaire de taille `width*height` ; les coordonnées dans `xi` varient entre `0,0` (coin en haut à gauche) et `width-1,height-1`.

```
XImage *xi;
xi = HeCreateXi (width, height);
```

Cette création contient une allocation de mémoire qui peut échouer ; il faut donc tester si `xi != NULL` avant de continuer.

On peut initialiser la couleur de fond de `xi` par

```
HeSetXiBg (xi, R, G, B);
```

où `R,G,B` sont des `unsigned char`. On fixe ensuite la couleur de chaque pixel de coordonnées `x,y` dans `xi` avec

```
HeSetXiPixel (xi, offset, R, G, B);
```

où `offset` est `y*width+x`. Attention, `x,y` ne doivent jamais être en dehors de `[0..width-1,0..height-1]` sous peine de plantage.

Si les valeurs des couleurs sont stockées dans des tableaux `tabR, tabG, tabB` de `unsigned char` et de taille exacte `[width*height]`, alors on peut appeler

```
HeRGBtoXi (xi, tabR, tabG, tabB);
```

Cet appel est équivalent à

```
int x, y, t;
for (y = 0; y < height; y++)
for (x = 0; x < width; x++) {
    t = y * width + x;
    HeSetXiPixel (xi, t, tabR[t], tabG[t], tabB[t]);
}
```

mais il est encore 50% plus rapide.

Une fois que `xi` est achevée, on peut l'afficher à l'écran. Cette étape est instantanée. Il suffit d'appeler

```
HePutXi (win, gc, xi, x, y);
```

où `win` est le Window du Canvas, et `x,y` sont les coordonnées dans le Canvas du coin en haut à gauche de `xi`. On peut donc ainsi afficher `xi` plusieurs fois et en plusieurs endroits du Canvas.

On peut aussi afficher une partie rectangulaire de `xi` avec

```
HePutSubXi (win, gc, xi, x, y, sub_x, sub_y, sub_w, sub_h);
```

où `x,y` sont les coordonnées dans le Canvas du coin en haut à gauche de `xi`, et `sub_x, sub_y, sub_w, sub_h` délimitent dans `xi` la partie rectangulaire de `xi` à afficher.

On appelle typiquement `HePutXi` ou `HePutSubXi` dans la RepaintProc d'un Canvas.

Lorsqu'on n'a plus besoin de `xi`, il faut la détruire. De même, si on veut changer la taille de `xi`, il faut détruire `xi` puis la recréer avec la nouvelle taille. On appelle donc

```
HeDestroyXi (xi);
```

Dans l'exemple suivant, on ouvre une fenêtre avec un Canvas, on crée une XImage dans `init_xi` (sans initialiser le fond) puis on l'affiche en damier dans la RepaintProc. On ne détruit pas la XImage dans cet exemple. Pour tester la rapidité de l'affichage, bouger la fenêtre ou faire glisser une autre fenêtre devant.

```
/* examples/canvas/xi.c */

#include <helium.h>

#define XMAX 180
#define YMAX 220

He_node *princ, *canvas;
XImage *xi;

void init_xi ()
{
    int x, y;

    /* Création */
    xi = HeCreateXi (XMAX, YMAX);
    if (xi == NULL) return;

    /* Calcul */
    for (y = 0; y < YMAX; y++)
    for (x = 0; x < XMAX; x++) {
        Uchar R = x, G = 255-x, B = y;
        HeSetXiPixel (xi, y*XMAX+x, R, G, B);
    }
}

void canvas_repaint_proc (He_node *hn, Window win)
{
    int i, j;

    /* Affichage multiple instantané */
    for (i = 0; i <= HeGetWidth (hn)/XMAX; i++)
    for (j = 0; j <= HeGetHeight(hn)/YMAX; j++)
        HePutXi (win, he_gc, xi, i*XMAX, j*YMAX);
}

int main (int argc, char *argv[])
```



```
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "XImage");

    canvas = HeCreateCanvas (princ);
    HeSetCanvasRepaintProc (canvas, canvas_repaint_proc);
    HeSetWidth (canvas, 500); HeSetHeight (canvas, 500);
    init_xi();

    HeFit (princ);

    return HeMainLoop (princ);
}
```

Remarque : dans Xlib, tous les mécanismes bas-niveau sont prévus pour manipuler les XImages ; mais l'étape de calcul d'une XImage est laissée au soin de l'utilisateur, et l'algorithme est relativement compliqué ; il suffit de regarder `gui/xi.c` pour s'en convaincre ...

## 7.10 Conversion de couleurs

Quelques fonctions de conversion de couleurs sont fournies dans Helium.

Dans le système RGB (Red, Green, Blue), chaque valeur est codée entre `0` et `HE_COLOR_MAXRGB = 255`. Dans le système HSV (Hue, Saturation, Value), `H` est entre `0` et `HE_COLOR_MAXH = 360`, `S` et `V` sont entre `0` et `HE_COLOR_MAXSV = 1000` (voir `gui/color.h`). Les fonctions de conversion entre les deux systèmes sont :

```
void HeHsvToRgb (int h, int s, int v, int *r, int *g, int *b);
void HeRgbToHsv (int r, int g, int b, int *h, int *s, int *v);
```

Le type XColor est un type de couleur fourni par Xlib dans `Xlib.h` :

```
typedef struct {
    unsigned long pixel;
    unsigned short red, green, blue;
    char flags;  /* do_red, do_green, do_blue */
    char pad;
} XColor;
```

Les valeurs `red, green, blue` sont codées sur 3*16 bits. On a les fonctions de conversion suivantes :

```
void HeRgbToXColor (int r, int g, int b, XColor *x);
void HeHsvToXColor (int h, int s, int v, XColor *x);
void HeXColorToRgb (XColor *x, int *r, int *g, int *b);
void HeXColorToHsv (XColor *x, int *h, int *s, int *v);
```

Dans la philosophie X11, les couleurs peuvent être décrites par un string, soit par leur nom (`"blue"`, `"yellow2"`, etc) soit par leur code hexadécimal (`"#37a"`, `"#e0b4b0"`, etc). Les fonctions de Helium pour interpréter ces noms sont :

```
int HeParseXColor (char *name, XColor *x);
int HeParseRgb (char *name, int *r, int *g, int *b);
int HeParseHsv (char *name, int *h, int *s, int *v);
```

## 7.11 Afficher du texte

Xlib fournit des fonctions très efficaces pour afficher du texte dans de multiples polices de caractères. On peut se servir de toutes les fonctions de dessin de Xlib dans un Canvas.

Par exemple, XLoadQueryFont charge une fonte (c'est-à-dire une police de caractères) à partir de son nom ; XSetFont fixe la fonte courante ; XDrawString affiche un string sans effacer le fond ; XDrawImageString affiche un string en effaçant le fond ; XDrawText affiche plusieurs strings avec plusieurs fontes.

Helium fournit quelques fonctions qui simplifient l'usage des fonctions de Xlib, spécialement au niveau du positionnement du string par rapport à un point de coordonnées x,y.

Les variables globales `he_normal_font` et `he_bold_font` sont les fontes normale et grasse utilisées pour le dessin des widgets. On rappelle que la variable globale `he_gc` sert à mémoriser les attributs du tracé. La fonction

```
HeDrawString (win, he_gc, he_normal_font, x, y, ligne)
```

affiche le `char *ligne` dans le Window `win` du Canvas, dans la fonte normale d'Helium. Le point `x,y` est le coin en haut à gauche du string. On peut aussi afficher un sous-string avec

```
HeDrawSubString (win, he_gc, he_normal_font,
                 x, y, ligne, pos, len)
```

où `pos` est le numéro du premier caractère de ligne à afficher, et `len` est le nombre de caractères à afficher. Si on veut afficher un string centré en hauteur et largeur par rapport à une zone rectangulaire, on dispose de la fonction

```
HeDrawStringCenterRect (win, he_gc, he_normal_font,
    xb, yb, xm, ym, ligne)
```

où `xb,yb` est le coin en haut à gauche et `xm,ym` est la largeur et la hauteur de la zone. La fonction

```
HeDimString (he_normal_font, ligne, &x, &y)
```

demande les dimensions en pixels d'un string dans une fonte, et les stocke dans les entiers `x,y`. Enfin la fonction

```
HeDrawStringPos (win, he_gc, he_normal_font, x, y, pos, ligne)
```

affiche une ligne de texte justifiée en hauteur et en largeur selon `pos`, par rapport au point de coordonnées `x,y`. Le paramètre `pos` peut prendre l'une des 9 valeurs :



```
HE_TOP_LEFT, HE_TOP_MIDDLE, HE_TOP_RIGHT
HE_BASE_LEFT, HE_BASE_MIDDLE, HE_BASE_RIGHT
HE_BOTTOM_LEFT, HE_BOTTOM_MIDDLE, HE_BOTTOM_RIGHT.
```

Dans l'exemple suivant, on illustre les différentes possibilités de justification avec HeDrawStringPos par rapport à un point donné :

```
/* examples/canvas/drawstring.c */

#include <helium.h>

He_node *princ, *canvas;
int rouge;

void init_couleurs ()
{
    rouge = HeAllocRgb (255, 0, 0, he_black);
}

void dessin_ligne (Window win, int x1, int y1, int x2, int y2, int c)
{
    XSetForeground (he_display, he_gc, c);
    XDrawLine (he_display, win, he_gc, x1, y1, x2, y2);
}

void dessin_string (Window win, int x, int y, int pos, char *ligne)
{
    dessin_ligne (win, x-20,y,x+20,y, rouge);
    dessin_ligne (win, x,y-20,x,y+20, rouge);
    XSetForeground (he_display, he_gc, he_black);
    HeDrawStringPos (win, he_gc, he_normal_font,
        x, y, pos, ligne);
}

void canvas_repaint (He_node *hn, Window win)
{
    dessin_string (win, 50,   50, HE_TOP_LEFT,      "TopLeft");
    dessin_string (win, 300,  50, HE_TOP_MIDDLE,    "TopMiddle");
    dessin_string (win, 550,  50, HE_TOP_RIGHT,     "TopRight");
    dessin_string (win, 50,  150, HE_BASE_LEFT,     "BaseLeft");
    dessin_string (win, 300, 150, HE_BASE_MIDDLE,   "BaseMiddle");
    dessin_string (win, 550, 150, HE_BASE_RIGHT,    "BaseRight");
    dessin_string (win, 50,  250, HE_BOTTOM_LEFT,   "BottomLeft");
    dessin_string (win, 300, 250, HE_BOTTOM_MIDDLE, "BottomMiddle");
    dessin_string (win, 550, 250, HE_BOTTOM_RIGHT,  "BottomRight");
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "HeDrawStringPos");

    canvas = HeCreateCanvas (princ);
    HeSetCanvasRepaintProc (canvas, canvas_repaint);
    init_couleurs ();

    HeSetWidth (canvas, 600); HeSetHeight (canvas, 300);
    HeFit (princ);

    return HeMainLoop (princ);
}
```

## 7.12 En savoir plus sur Xlib

Les livres officiels pour apprendre à programmer Xlib sont : "Xlib Programming Manual" (Volume One) et "Xlib Reference Manual" (Volume Two), chez O'Reilly.

Le système XWindow est maintenu par le XConsortium, regroupant les grands groupes informatiques du monde Unix.

Actuellement, les plus gros développements sont faits par le groupe XFree86, auteur d'une implémentation libre de Xlib.

# Chapitre 8

# Dessins en OpenGL

Dans ce chapitre nous présentons le widget GLArea, qui sert à faire des dessins en 2D ou 3D avec OpenGL, et à réagir au clavier et à la souris.

La seconde partie du chapitre est un mini-tutorial pour découvrir OpenGL ; nous indiquons à la fin des pointeurs pour en savoir plus (livres et documentation des fonctions OpenGL sur le web, etc).

## 8.1 Courte présentation de OpenGL

OpenGL est un langage pour le développement d'applications graphiques, produisant des images en couleurs d'objets 3D. Les fonctions de OpenGL permettent de construire des modèles géométriques, de représenter intéractivement des scènes dans l'espace, de contrôler les propriétés de couleurs, de matériaux et d'éclairage, de manipuler les pixels, et donne accès à la transparence, l'antialiasing, le plaquage de textures et à des effets atmosphériques.

OpenGL a été développé pour interfacer les cartes graphiques 3D, mais peut aussi être complètement émulé avec la librairie Mesa (au prix de performances moindres). OpenGL est devenu le standard de fait de l'industrie, et est maintenu par un Consortium de grands groupes informatiques, l'ARB (Architecture Review Board).

## 8.2 Installation et compilation

Le toolkit Helium est réparti en deux librairies : libHelium, qui contient toutes les fonctions ne dépendant que de Xlib, et libHeliumGL, qui contient les fonctions pour interfacer OpenGL. Nous avons fait cette séparation pour que les applications sous Helium qui n'utilisent pas OpenGL puissent être compilées indépendament.

Les librairies d'OpenGL sont : libGL (les fonctions de base) et libGLU (des fonctions ”Utiles”, qui étendent ou simplifient les fonctions de base). Si vous utilisez Mesa, ces librairies risquent de s'appeler libMesaGL et libMesaGLU.





La première chose à faire est de tester si Helium a été configuré avec OpenGL, en exécutant `/chemin-helium/gldemo/gears` . Si les roues dentées apparaissent, tout va bien ; sinon reportez-vous au document INSTALL.

Un programme Helium utilisant OpenGL doit comporter la ligne `#include <heliumGL.h>` à la place de `#include <helium.h>` .

Pour compiler un tel programme, il suffit de remplacer `--cflags` et `--libs` par `--gl-cflags` et `--gl-libs` en ligne de commande ou dans un script. Dans un Makefile, remplacer simplement `HE_CFLAGS` et `HE_LIBS` par `HE_GL_CFLAGS` et `HE_GL_LIBS`. À noter, `HE_GL_LIBS` contient ”`-lm`”.

Pour compiler un exemple ”`ex.c`”, il suffit donc de taper (avec les backquotes ‘ ‘) :

```
gcc ex.c -o ex  `/chemin-helium/helium-cfg --gl-cflags --gl-libs`
```

(remplacer `/chemin-helium` par le chemin absolu). Pour faciliter la compilation des exemples utilisant GLArea, voici un petit script en sh :

```
#! /bin/sh
p=/chemin-helium
f=`basename $1 .c`
gcc $f.c -o $f `$p/helium-cfg --gl-cflags --gl-libs`
```

Appeler ce script ”`hcompgl`”, sauver, taper ”`chmod +x hcompgl`”. Pour compiler un fichier ”`ex.c`”, on tape ”`./hcompgl ex.c`” ou ”`./hcompgl ex`”. Pour compiler un exemple et lancer automatiquement l'exécution si la compilation a réussie, on tape ”`./hcompgl ex.c && ex`”.

Voici un Makefile simple ; on inclut le fichier de configuration de Helium `/chemin-helium/.config`, où les variables nécessaires sont déclarées, en particulier `$(CC)`, `$(HE_GL_CFLAGS)` et `$(HE_GL_LIBS)`.
Attention, vérifiez que les lignes décalées commencent bien par un [TAB].

```
include /chemin-helium/.config

.c.o :
	$(CC) -c $(HE_GL_CFLAGS) $*.c

bouton : bouton.o
	$(CC) -o $@ $@.o $(HE_GL_LIBS)
```

Enfin, voici un Makefile sophistiqué, qui affiche un Message d'aide, permet de compiler plusieurs programmes et de nettoyer le répertoire.

```
include /chemin-helium/.config

.c.o :
	$(CC) -c $(HE_GL_CFLAGS) $*.c

# Rajoutez ici le nom de votre_prog
```

```
EXECS = bouton bouton2 bouton3

help ::
	@echo "Options du make : help all clean distclean $(EXECS)"

# Rajoutez ici votre_prog : votre_prog.o
bouton : bouton.o
bouton2 : bouton2.o
bouton3 : bouton3.o

all :: $(EXECS)

$(EXECS) :
	$(CC) -o $@ $@.o $(HE_GL_LIBS)

clean ::
	\rm -f *.o core

distclean :: clean
	\rm -f $(EXECS)
```

## 8.3 Le widget GLArea

Le widget GLArea est un window X11 dans lequel on peut faire du OpenGL, via Mesa ou le GL natif de la carte graphique (sur SGI par exemple). Helium utilise l'extension GLX pour interfacer X11 et OpenGL.

La philosophie d'un GLArea est la même que celle d'un Canvas ; il y a une RepaintProc, une ResizeProc et une EventProc. Il y a aussi une InitProc, dont nous allons voir l'utilité.

Pour créer un GLArea on fait

```
int attr_list[] = { GLX_RGBA, None };
He_node *glarea;
glarea = HeCreateGLArea (frame, attr_list, NULL);
```

où `frame` est le Frame qui hébergera le GLArea. Par défaut, le GLArea est placé en 0,0 du Frame, et son aspect est un rectangle noir, sans bord.

La variable `attr_list` permet de paramétrer OpenGL, en lui donnant une liste d'attributs terminés par `None` (cf documentation de `glXChooseVisual()`). Ici on demande de coder les couleurs en RGBA (Red Green Blue Alpha, c'est-à-dire rouge vert bleu transparence).

Le troisième paramètre de HeCreateGLArea, que l'on a mis à `NULL`, sert à partager un contexte avec un autre GLArea déjà créé, par exemple pour afficher plusieurs vues d'un même objet.

Pour dessiner dans le GLArea, on attache une callback RepaintProc :

```
HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);
```



Le prototype de la callback est

```
void glarea_repaint_proc (He_node *hn);
```

où `hn` est le GLArea. Chaque fois que le GLArea doit être réaffiché, Helium appelle la RepaintProc en lui passant le widget GLArea dans lequel dessiner en OpenGL. N'importe quelle fonction de GL ou de GLU peut être appelée dans la RepaintProc.

Lorsqu'on fait un dessin en OpenGL, il faut d'abord préciser le type de projection (avec ou sans perspective), poser un repère, donner l'éclairage, etc. Ce genre d'initialisation n'a besoin d'être faite qu'une seule fois au début du programme. L'endroit idéal est la callback InitProc, qui n'est appelée qu'une seule fois, juste avant le premier appel de la RepaintProc. Pour attacher une callback InitProc au GLArea on fait :

```
HeSetGLAreaInitProc (glarea, glarea_init_proc);
```

Le prototype de la callback est

```
void glarea_init_proc (He_node *hn);
```

où `hn` est le GLArea. N'importe quelle fonction de GL ou de GLU peut être appelée dans la InitProc.

L'exemple suivant affiche un triangle blanc sur fond noir ; on revient sur les fonctions OpenGL dans la section « 8.7 Dessins en 2D avec OpenGL » :

```
/* examples/glarea/triangle.c */

#include <heliumGL.h>

He_node *princ, *glarea;

void glarea_init_proc (He_node *hn)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 100.0, 0.0, 100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void glarea_repaint_proc (He_node *hn)
{
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1,1,1);
    glBegin(GL_TRIANGLES);
    glVertex2f(10,10);
    glVertex2f(10,90);
    glVertex2f(90,90);
    glEnd();
}

int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA, None };
```

```
    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Triangle");

    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetWidth (glarea, 500);
    HeSetHeight (glarea, 400);
    HeSetGLAreaInitProc (glarea, glarea_init_proc);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);

    HeFit (princ);

    return HeMainLoop (princ);
}
```

La chose importante à comprendre, est qu'on ne peut pas appeler des fonctions de OpenGL n'importe où dans le programme : il faut le faire dans des endroits prévus pour, ou alors il faut appeler une fonction spéciale, qui demande de "rendre courant un contexte GL". En effet, contrairement aux fonctions de Xlib où on passe de nombreux paramètres, ce qui est un peu lourd à écrire mais plus précis en terme de programmation, les fonctions OpenGL prennent très peu de paramètres. La raison est que OpenGL est une sorte d'automate à états ; on dispose de nombreuses fonctions pour modifier ces états, et le résultat d'une fonction OpenGL dépend de la valeurs des états au moment où on l'appelle. Ces états sont mémorisés dans un contexte GL (un peu comme un GC pour Xlib) mais il n'y a pas de paramètre pour désigner le contexte GL aux fonctions OpenGL ; à la place, il faut rendre courant le contexte GL du GLArea.

Ceci est fait automatiquement par Helium juste avant l'appel de chaque callback InitProc, RepaintProc, ResizeProc et EventProc, si bien que vous pouvez faire directement du OpenGL dans ces callbacks. Dans toute autre fonction qui n'est pas appelée depuis ces callback, comme dans le programme principal, si vous voulez faire du OpenGL, il faut commencer par rendre courant le contexte du GLArea, en faisant

```
if (HeGLAreaMakeCurrent(hn) < 0) return;
```

où `hn` est le GLArea. La fonction HeGLAreaMakeCurrent renvoie 0 en cas de succès, -1 en cas d'erreur. Si le contexte est déjà courant, cet appel n'a aucun effet.

## 8.4 Redimensionner le GLArea

La ResizeProc du GLArea fonctionne exactement comme la ResizeProc du Canvas : chaque fois que la taille du GLArea change, la callback ResizeProc du GLArea est automatiquement appelée (si définie), juste avant que la RepaintProc ne soit appelée à son tour. Au premier affichage du GLArea, les callbacks du GLArea sont appelées dans cet ordre : l'InitProc, puis la ResizeProc, puis la RepaintProc. Pour attacher une callback ResizeProc au GLArea on fait

```
HeSetGLAreaResizeProc (glarea, glarea_resize_proc);
```

Le prototype de la callback est



```
void glarea_resize_proc (He_node *hn, int width, int height);
```

où `hn` est le GLArea, `width` et `height` sont les nouvelles dimensions.

La différence avec le Canvas est que le GLArea a une Resizeproc par défaut. Si vous donnez une ResizeProc à votre GLArea, cette ResizeProc par défaut est annulée. La ResizeProc par défaut est la fonction suivante (cf `glx/glarea.c`) :

```
void HeGLAreaDefaultResizeProc (He_node *hn, int width, int height)
{
    glViewport(0,0, width, height);
}
```

La fonction glViewport sert à dire dans quelle partie rectangulaire du GLArea on doit afficher la scène. Les paramètres de glViewport, exprimés en pixels, sont (x, y, largeur, hauteur) où x,y est la coordonnée du coin en haut à gauche relativement au coin en haut à gauche du GLArea. On voit donc que la ResizeProc par défaut demande que la vue occupe tout le GLArea.

Moralité : le glViewport est automatique par défaut, et vous n'avez pas besoin de vous en soucier ni de mettre de ResizeProc (c'était notre idée de départ). Maintenant, si vous avez besoin de faire d'autres choses au moment du Resize, n'oubliez pas d'appeler glViewport dans votre ResizeProc.

Comme pour le Canvas, un HeSetWidth ou un HeSetHeight sur le GLArea déclenche un appel de la ResizeProc puis de la RepaintProc. Un bon endroit où changer la taille du GLArea est la ResizeProc d'un Frame. Un effet amusant lorsqu'un GLArea change de taille est qu'il change aussi les proportions du dessin : dans l'exemple suivant, les triangles affichés s'applatissent lorsqu'on réduit la hauteur de la fenêtre. Il y a une parade pour conserver les proportions, parade que l'on verra plus loin.

```
/* examples/glarea/resize.c */

#include <heliumGL.h>

He_node *princ, *glarea;

void princ_resize (He_node *hn, int width, int height)
{
    HeExpand(glarea, NULL, HE_BOTTOM_RIGHT);
}

void glarea_init_proc (He_node *hn)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,100, 0,100);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void glarea_repaint_proc (He_node *hn)
{
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glColor3f(1,0,0);
    glBegin(GL_TRIANGLES);
    glVertex2f(10,10);
    glVertex2f(10,90);
    glVertex2f(90,90);
    glEnd();

    glColor3f(0,0.5,0.75);
    glBegin(GL_TRIANGLES);
    glVertex2f(20,10);
    glVertex2f(50,40);
    glVertex2f(90,20);
    glEnd();
}

int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA, None };

    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Resize et proportions");
    HeSetFrameResizeProc (princ, princ_resize);

    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetWidth (glarea, 500);
    HeSetHeight (glarea, 400);
    HeSetGLAreaInitProc (glarea, glarea_init_proc);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);
    /* ici on laisse la ResizeProc par défaut du GLArea */

    HeFit (princ);

    return HeMainLoop (princ);
}
```

## 8.5 Évènements dans un GLArea

La EventProc du GLArea fonctionne exactement comme la EventProc du Canvas : pour recevoir des évènements dans un GLArea, il suffit d'attacher une callback EventProc avec

```
HeSetGLAreaEventProc (glarea, glarea_event_proc);
```

le prototype de la callback est

```
void glarea_event_proc (He_node* hn, He_event *hev);
```

où `hn` est le GLArea et `hev` contient les caractéristiques de l'évènement. Cette callback est appelée par Helium chaque fois que l'un des évènements X11 suivants arrive au GLArea :

- ▷ `EnterNotify` : la souris rentre dans le GLArea
- ▷ `LeaveNotify` : la souris sort du GLArea
- ▷ `KeyPress` : une touche est enfoncée
- ▷ `KeyRelease` : une touche est relachée
- ▷ `ButtonPress` : un bouton de la souris est enfoncé



- ▷ `ButtonRelease` : un bouton de la souris est relaché
- ▷ `MotionNotify` : la souris a bougé

Le type `He_event` est défini dans include/types.h :

```
typedef struct He_event_st {
    int     type,           /* Type d'évènement = xev->type */
            sx, sy,         /* Coords souris / window */
            sb;             /* Bouton souris filtré : 0,1,2,3 */
    Time    time;           /* Temps en milli secondes */
    Window  win;            /* Le Window X11 du widget */
    XEvent *xev;            /* Evènement X11 complet */
    char    str[256];       /* Buffer lu au clavier */
    int     len;            /* Nombre de char dans str */
    KeySym  sym;            /* Symbole de la touche pressée */
} He_event;
```

Dans l'exemple suivant on affiche tous les évènements :

```
/* examples/glarea/event.c */

#include <heliumGL.h>

He_node *princ, *glarea;

void glarea_repaint_proc (He_node *hn)
{
    printf ("glarea_repaint_proc\n");
}

void glarea_event_proc (He_node *hn, He_event *hev)
{
    printf ("glarea_event_proc ");

    switch (hev->type) {
        case EnterNotify :
            printf ("EnterNotify\n");
            break;
        case LeaveNotify :
            printf ("LeaveNotify\n");
            break;
        case KeyPress :
            printf ("KeyPress: \"%s\" keysym = XK_%s len = %d\n",
                hev->str, XKeysymToString(hev->sym), hev->len);
            break;
        case KeyRelease :
            printf ("KeyRelease\n");
            break;
        case ButtonPress :
            printf ("ButtonPress  bouton %d\n", hev->sb);
            break;
        case ButtonRelease :
            printf ("ButtonRelease  bouton %d\n", hev->sb);
            break;
        case MotionNotify :
            printf ("MotionNotify  %d,%d\n", hev->sx, hev->sy);
            break;
    }
}
```

```
int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA, None };

    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Évènements du glarea");

    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);
    HeSetGLAreaEventProc (glarea, glarea_event_proc);

    HeSetWidth (glarea, 300); HeSetHeight (glarea, 300);
    HeFit (princ);

    return HeMainLoop (princ);
}
```

Remarques :

- ▷ On peut aussi dessiner dans l'EventProc : n'importe quelle fonction de GL ou de GLU peut y être appelée. Mais il faut prévoir de mémoriser ces dessins pour que la RepaintProc puisse éventuellement tout redessiner.
- ▷ Les coordonnées `hev->sx` et `hev->sy` sont des coordonnées souris par rapport au coin en haut à gauche du GLArea ; ces coordonnées n'ont rien à voir avec les coordonnées des objets de la scène. Retrouver les coordonnées d'un sommet dans la scène 3D à partir des coordonnées souris est une sorte de sport !
- ▷ On peut provoquer un réaffichage du GLArea dans l'EventProc. On peut aussi utiliser l'EventProc pour simuler un trackball : voir les sections suivantes.

## 8.6 Provoquer un réaffichage

Comme pour le Canvas, on peut provoquer un réaffichage du GLArea avec

```
HePostRepaint (glarea);
```

Ceci provoque un appel (à peine) différé de la RepaintProc par Helium. On peut appeler HePostRepaint depuis n'importe où ; les deux cas typiques sont l'EventProc du GLArea et la NotifyProc d'un bouton.

En fait, HePostRepaint envoie simplement un évènement Expose au Window du GLArea ; cet évènement est traité une fois qu'on est sorti de la callback d'où on a fait l'appel. Lorsque le GLArea reçoit plusieurs Expose très rapprochés, Helium le détecte et ne commande l'appel de la RepaintProc que sur le dernier Expose, pour gagner en fluidité ; donc si vous appelez plusieurs fois de suite HePostRepaint, un seul RepaintProc sera effectué (le dernier du lot).

On peut aussi appeler directement la fonction qui fait office de RepaintProc pour tout redessiner à un moment donné ; on doit lui fournir le GLArea, mais ce n'est pas forcément suffisant : il faut d'abord être sûr que le contexte GL du GLArea est le contexte GL courant. Examinons les cas de figure : soit `my_repaint_proc` la RepaintProc d'un GLArea `gla`. L'appel



```
my_repaint_proc(gla);
```

peut être fait depuis toute callback de `gla` (puisque Helium rend courant le contexte GL d'un GLArea avant d'appeler ses callback), par exemple depuis son EventProc. Mais si vous voulez le faire depuis la callback d'un bouton par exemple, il faudra écrire

```
if (HeGLAreaMakeCurrent(gla) == 0) my_repaint_proc(gla);
```

Moralité : il est fortement conseillé d'adopter la première solution, avec l'appel de HePostRepaint.

## 8.7 Dessins en 2D avec OpenGL

OpenGL représente tous les objets dans l'espace (x,y,z). Pour faire des dessins en 2D, il suffit de donner des coordonnées avec z=0 d'une part, et d'autre part de regarder le plan (x,y) en s'éloignant un peu sur l'axe des z.

On voit donc qu'il y a deux notions, d'un côté les coordonnées des objets et les transformations géométriques tels que rotation, translation, etc ; de l'autre côté, il y a le mode de projection à l'écran, avec perspective (projection cônique) ou sans (projection orthographique).

OpenGL fait cette distinction : il code toutes les transformations et les projections dans deux matrices : `GL_MODELVIEW` et `GL_PROJECTION`. La fonction `glMatrixMode()` permet d'indiquer quelle est la matrice "courante". Toutes les fonctions de transformation, que se soit géométriques ou de projection, modifient des valeurs de la matrice courante.

Ainsi la fonction `glLoadIdentity()` initialise la matrice courante à la matrice identité (diagonale à 1 et le reste à 0). Il est fortement conseillé d'initialiser les deux matrices avant toute chose, et la place idéale est la callback InitProc.

Revenons au dessin en 2D : pour avoir une vue 2D de notre plan (x,y), il suffit donc de demander une projection orthographique 2D, et ceci se fait dans la matrice `GL_PROJECTION`. Voici l'InitProc des exemples précédents "`triangle.c`" et "`resize.c`" avec des commentaires :

```
void glarea_init_proc (He_node *hn)
{
    /* La matrice de projection devient la matrice courante */
    glMatrixMode(GL_PROJECTION);

    /* On initialise la matrice de projection */
    glLoadIdentity();

    /* On code la projection orthographique dans la matrice */
    gluOrtho2D(0.0, 100.0, 0.0, 100.0);

    /* La matrice des transfo. géométriques devient courante */
    glMatrixMode(GL_MODELVIEW);
```

```
    /* On initialise la matrice des transfo. géométriques */
    glLoadIdentity();

    /* Dans la RepaintProc on sera donc en mode GL_MODELVIEW et
       on pourra faire des rotations et translations */
}
```

Les paramètres de `gluOrtho2D()` sont (gauche, droite, bas, haut) : ce sont les coordonnées de la région visible dans l'espace. Ici on se retrouve avec 0.0,0.0 en bas à gauche du GLArea et 100.0,100.0 en haut à droite (quelle que soit la taille en pixels du GLArea).

Passons maintenant aux dessins proprement dits, que l'on a fait dans la Repaintproc des exemples précédents "`triangle.c`" et "`resize.c`". On commence par initialiser la couleur du fond avec

```
glClearColor(0,0,0,1);
glClear(GL_COLOR_BUFFER_BIT);
```

La commande `glClearColor` fixe une couleur RGBA pour le fond, chaque paramètre étant entre 0.0 et 1.0. Ici on choisit la couleur noire (0,0,0) et opaque (1). Le dessin du fond proprement dit est fait par l'ordre `glClear(GL_COLOR_BUFFER_BIT)` .

Pour dessiner un objet dans une couleur `r,g,b`, il suffit d'appeler avant la commande

```
glColor3f(r,g,b);
```

où `r,g,b` sont entre 0.0 et 1.0 . Tous les dessins faits par la suite auront cette couleur, jusqu'au prochain appel de `glColor3f` (c'est le même principe que XSetForeground pour le Canvas).

Pour dessiner un point de coordonnées réelles `x1,y1` on fait

```
glBegin(GL_POINTS);
glVertex2f(x1,y1);
glEnd();
```

Pour dessiner une ligne d'un point `x1,y1` à un point `x2,y2` on fait

```
glBegin(GL_LINES);
glVertex2f(x1,y1);
glVertex2f(x2,y2);
glEnd();
```

En fait de nombreux types de dessins peuvent être faits sur le modèle

```
glBegin(mode);
glVertex2f(x1,y1);
glVertex2f(x2,y2);
....
glVertex2f(xn,yn);
glEnd();
```



en précisant le `mode` et une liste de sommets réels `xi,yi`. Les valeurs possibles de `mode` sont :
- ▷ `GL_POINTS` : trace un point pour chacun des n sommets.
- ▷ `GL_LINES` : trace une série de lignes non connectées. Les segments sont tracés entre le sommet 1 et 2, puis 3 et 4, etc. Si n est impair, le dernier sommet est ignoré.
- ▷ `GL_LINE_STRIP` : trace une ligne reliant les sommets 1 à 2, puis 2 à 3, jusqu'à n-1 à n.
- ▷ `GL_LINE_LOOP` : identique à `GL_LINE_STRIP`, plus une ligne reliant le sommet n au sommet 1, pour fermer la boucle.
- ▷ `GL_TRIANGLES` : remplis une suite de triangle avec les sommets 1,2,3 puis 4,5,6 etc. Si n n'est pas multiple de 3 les sommets restants sont ignorés.
- ▷ `GL_TRIANGLE_STRIP` : remplis une suite de triangle avec les sommets 1,2,3 puis 3,2,4 puis 3,4,5 puis 5,4,6 etc. L'ordre des sommet est pris pour respecter l'orientation des triangles.
- ▷ `GL_TRIANGLE_FAN` : remplis une suite de triangle avec les sommets 1,2,3 puis 1,3,4 puis 1,4,5 etc. Tous les triangles ont le sommet 1 en commun.
- ▷ `GL_QUADS` : remplis une suite de quadrilatères de sommets 1,2,3,4 puis 5,6,7,8 etc. Si n n'est pas multiple de 4 les sommets restants sont ignorés.
- ▷ `GL_QUADS` : remplis une suite de quadrilatères de sommets 1,2,4,3 puis 3,4,6,5 puis 5,6,8,7 etc. L'ordre des sommet est pris pour respecter l'orientation des quadrilatères.
- ▷ `GL_POLYGON` : remplis un polygone avec les n sommets. Si n est inférieur à 3 rien n'est tracé. Le polygone ne doit pas se croiser et il doit être convexe, sinon le résultat est imprévisible.

On peut changer la taille d'un point, l'épaisseur des lignes, et même activer l'antialiasing, c'est-à-dire "lisser" l'affichage en enlevant les "marches d'escalier". Pour changer la taille d'un point on fait

```
glPointSize(size);
```

où `size` est un réel, indiquant la taille du carré qui représentera le point à l'écran, en nombre de pixels. La taille par défaut est 1. Si l'anti-aliasing est actif, l'affichage donnera un disque (c'est là que la valeur réelle de `size` prend un sens). Pour changer l'épaisseur de tracé d'une ligne, on fait

```
glLineWidth(width);
```

où `width` est un réel représentant l'épaisseur en nombre de pixels, en coupe verticale si la pente est inférieure à 1, sinon en coupe horizontale. Pour activer ou désactiver l'anti-aliasing on fait

```
glEnable(GL_LINE_SMOOTH);
glDisable(GL_LINE_SMOOTH);
```

Par défaut l'anti-aliasing est désactivé, ce qui économise du temps de calcul.

## 8.8 Dessins en 3D avec OpenGL

C'est en 3D que OpenGL montre sa vraie puissance. Tout ce qu'on a dit pour les dessins en 2D reste valable pour les dessins en 3D. On doit simplement donner les 3 coordonnées de chaque sommet avec la fonction `glVertex3f()`. Par exemple pour tracer un point de coordonnées réelles `x1,y1,z1` on fait

```
glBegin(GL_POINTS);
glVertex3f(x1,y1,z1);
glEnd();
```

Ce qui change par rapport au 2D, c'est le mode de projection, nécessaire pour visualiser une scène 3D sur un écran 2D. OpenGL propose différents modes de projections, la plus réaliste étant la projection en perspective. À la place de `gluOrtho2D` on appelle

```
gluPerspective(fovy, aspect, near, far);
```

où `fovy` est l'angle de vue, entre 0.0 et 180.0 degrés ; `aspect` est le ratio largeur/hauteur ; `near` et `far` sont les distances des plans de coupe à la camera (toujours positives).

L'angle de vue `fovy` donne le même effet que le zoom d'un appareil photo : un petit angle rapproche les objets comme un téléobjectif, alors qu'un grand angle donne une vue panoramique de la scène.

Le paramètre `aspect` permet de fixer les proportions entre largeur et hauteur de la vue. Si on met toujours 1.0, on aura une déformation si l'on redimensionne le GLArea. Pour au contraire conserver les proportions des objets dans la vue, il suffit de donner comme valeur : largeur du GLArea / hauteur du GLArea.

Un plan de coupe permet d'économiser beaucoup de temps de calcul, en annulant tous les dessins qui sont de l'autre côté du plan de coupe. Dans le mode de projection perspective, la partie visible est un volume limité par 6 plans de coupes : deux plans perpendiculaires à la direction du point de vue, distants de `near` et de `far` du point de vue ; tout ce qui est devant le plan `near` ou derrière le plan `far` est coupé. Les 4 autres plans partent du point de vue (comme le sommet d'une pyramide) et matérialisent l'angle de vue ; ils correspondent en projection aux côtés du GLArea.

Par défaut, la caméra est en 0,0,0 et regarde -Oz. Si les objets de la scène ont des coordonnées dans un "voisinage de l'origine", on peut demander de reculer toute la scène de quelques unités dans l'axe des z, avec

```
glTranslated (0, 0, -dist);
```

où `dist` est une valeur positive, par exemple 10. On obtient ainsi le même effet qu'en reculant l'appareil photo. Attention, lorsqu'on fait une translation, à garder le point de vue dirigé vers la scène, sinon on ne voit plus rien !

Dans l'exemple suivant on affiche une vue en perspective de 2 triangles en couleurs dans l'espace :

```
/* examples/glarea/perspective.c */

#include <heliumGL.h>

He_node *princ, *panel, *glarea;

void princ_resize (He_node *hn, int width, int height)
```



```
{
    HeExpand (panel, NULL, HE_RIGHT);
    HeExpand (glarea, NULL, HE_BOTTOM_RIGHT);
}

void butt_proc (He_node *hn)
{
    HeQuit(0);
}

void glarea_repaint_proc (He_node *hn)
{
    /* Mode perspective :
     * au lieu de le faire une seule fois dans l'InitProc, on le
     * fait à chaque fois : ça permet de conserver les proportions
     * de perspective. On aurait pu le faire dans la ResizeProc ...
    */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60, (GLdouble) HeGetWidth(hn)/HeGetHeight(hn),
                   1, 100);

    /* Coordonnées de la scène :
     * on recule tout ce qui va être dessiné ; de la sorte on peut
     * travailler avec des z autour de 0, et pas uniquement < 0.
    */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslated (0, 0, -4);

    /* Init du fond : noir opaque */
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1,0,0);   /* Rouge */
    glBegin(GL_TRIANGLES);
    glVertex3f(-1, -1, 0.5);
    glVertex3f(0, 1, 0.5);
    glVertex3f(1, -1, 0.5);
    glEnd();

    glColor3f(0,1,0);   /* Vert */
    glBegin(GL_TRIANGLES);
    glVertex3f(1, 1, -0.5);
    glVertex3f(-1, 0.3, 1.5);
    glVertex3f(0.8, -2, -0.5);
    glEnd();

    /* Le triangle vert est censé traverser le triangle rouge ;
     * or ici il apparait au dessus du rouge, car il est dessiné en
     * second. Il faut donc activer le Zbuffer.
    */
}

int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA, None };

    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Perspective");
    HeSetFrameResizeProc (princ, princ_resize);

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "Quit", butt_proc, NULL);
```

```
    HeFit(panel);

    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetY (glarea, HeGetHeight(panel)+2);
    HeSetWidth (glarea, 500);
    HeSetHeight (glarea, 400);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);

    HeFit (princ);

    return HeMainLoop (princ);
}
```

Dans cet exemple on constate que les proportions sont bien conservées lorsqu'on redimensionne la fenêtre. Par contre, le triangle vert apparaît "au dessus" du rouge, alors qu'ils sont sensés se traverser d'après leur coordonnées. Il faut donc activer le Zbuffer d'Opengl, c'est-à-dire le calcul des parties cachées depuis le point de vue. Il y a trois choses à faire dans différents endroits du programme pour activer le Zbuffer :

```
int attr_list[] = { ..., GLX_DEPTH_SIZE, 1, ... };
glEnable(GL_DEPTH_TEST);
glClear(... | GL_DEPTH_BUFFER_BIT);
```

Après corrections, l'exemple "`perspective.c`" devient "`zbuffer.c`" :

```
/* examples/glarea/zbuffer.c */

#include <heliumGL.h>

He_node *princ, *panel, *glarea;

void princ_resize (He_node *hn, int width, int height)
{
    HeExpand (panel, NULL, HE_RIGHT);
    HeExpand (glarea, NULL, HE_BOTTOM_RIGHT);
}

void butt_proc (He_node *hn)
{
    HeQuit(0);
}

void glarea_init_proc (He_node *hn)
{
    glEnable(GL_DEPTH_TEST);
}

void glarea_repaint_proc (He_node *hn)
{
    /* Mode perspective : au lieu de le faire une seule fois dans
     * l'InitProc, on le fait à chaque fois : ça permet de conserver
     * les proportions de perspective.
    */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60, (GLdouble) HeGetWidth(hn)/HeGetHeight(hn),
                   1, 100);

    /* Coordonnées de la scène :
     * on recule tout ce qui va être dessiné ; de la sorte on peut
     * travailler avec des z autour de 0, et pas uniquement < 0.
    */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslated (0, 0, -4);

    /* Init du fond (noir opaque) et du Zbuffer */
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1,0,0);   /* Rouge */
    glBegin(GL_TRIANGLES);
    glVertex3f(-1, -1, 0.5);
    glVertex3f(0, 1, 0.5);
    glVertex3f(1, -1, 0.5);
    glEnd();

    glColor3f(0,1,0);   /* Vert */
    glBegin(GL_TRIANGLES);
    glVertex3f(1, 1, -0.5);
    glVertex3f(-1, 0.3, 1.5);
    glVertex3f(0.8, -2, -0.5);
    glEnd();
}

int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA,           /* Codage couleurs */
                        GLX_DOUBLEBUFFER,   /* Double buffer   */
                        GLX_DEPTH_SIZE, 1,  /* Zbuffer         */
                        None };

    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Perspective et Zbuffer");
    HeSetFrameResizeProc (princ, princ_resize);

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "Quit", butt_proc, NULL);
    HeFit(panel);

    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetY (glarea, HeGetHeight(panel)+2);
    HeSetWidth (glarea, 500);
    HeSetHeight (glarea, 400);
    HeSetGLAreaInitProc (glarea, glarea_init_proc);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);

    HeFit (princ);

    return HeMainLoop (princ);
}
```



On en a profité pour initialiser OpenGL avec le double buffer dans `attr_list`. Le but est d'éviter l'affichage de vues incomplètement dessinées. Pour ce faire, OpenGL conserve deux images en mémoire, celle qui est affichée et celle dans laquelle on dessine la future image ; Helium permute automatiquement les deux images à la fin de chaque RepaintProc (en appelant `HeGLAreaSwapBuffers(glarea);`). Comparez la stabilité de l'affichage en glissant une petite fenêtre ou un icone devant le GLArea de "`perspective.c`" et "`zbuffer.c`".

## 8.9 Simuler un trackball

Nous avons inclus dans Helium des fonctions pour changer intéractivement de vue sur la scène 3D. Ces fonctions sont très faciles d'emploi ; elles permettent de faire des rotations sur les 3 axes (à la façon d'un trackball), de changer l'angle de vue (à la façon d'un téléobjectif) ou encore de déplacer le point de vue (en se rapprochant ou en s'éloignant).

Tout le GLArea sert pour simuler le trackball : les mouvements verticaux et horizontaux passant par le centre du GLArea font les rotations correspondantes, et un mouvement circulaire autour du centre du GLArea fait la rotation sur le troisième axe. En ce qui concerne les zooms, seul le déplacement vertical est considéré : on se rapproche vers le haut et on s'éloigne vers le bas.

Dans l'exemple suivant, le bouton 1 de la souris sert pour le trackball, le bouton 2 sert pour changer l'angle de vue et le bouton 3 sert pour éloigner le point de vue. Les fonctions sont expliquées après l'exemple.

```
/* examples/glarea/demotb.c */

#include <heliumGL.h>

He_node *princ, *panel, *glarea;
He_trackball info_tb;

void princ_resize (He_node *hn, int width, int height)
{
    HeExpand (panel, NULL, HE_RIGHT);
    HeExpand (glarea, NULL, HE_BOTTOM_RIGHT);
}

void butt_proc (He_node *hn)
{
    HeQuit(0);
}

void glarea_init_proc (He_node *hn)
{
    HeTbInitRotations(&info_tb);
    HeTbInitFovy(&info_tb, 60, 5, 120);
    HeTbInitZdist(&info_tb, 4, 1, 10);

    glEnable(GL_DEPTH_TEST);
}

void glarea_repaint_proc (He_node *hn)
{
    /* Mode perspective : au lieu de le faire une seule fois dans
     * l'InitProc, on le fait à chaque fois : ça permet de conserver
     * les proportions de perspective.
    */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective( HeTbGetFovy(&info_tb),
                    (GLdouble) HeGetWidth(hn)/HeGetHeight(hn),
                    1, 100 );

    /* Coordonnées de la scène :
     * on recule tout ce qui va être dessiné ; de la sorte on peut
     * travailler avec des z autour de 0, et pas uniquement < 0.
    */
    glMatrixMode(GL_MODELVIEW);
```



```
    glLoadIdentity();
    glTranslated (0, 0, -HeTbGetZdist(&info_tb));

    /* On tourne le repère d'après le trackball */
    HeTbMultRotations(&info_tb);

    /* Init du fond (noir opaque) et du Zbuffer */
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1,0,0);   /* Rouge */
    glBegin(GL_TRIANGLES);
    glVertex3f(-1, -1, 0.5);
    glVertex3f(0, 1, 0.5);
    glVertex3f(1, -1, 0.5);
    glEnd();

    glColor3f(0,1,0);   /* Vert */
    glBegin(GL_TRIANGLES);
    glVertex3f(1, 1, -0.5);
    glVertex3f(-1, 0.3, 1.5);
    glVertex3f(0.8, -2, -0.5);
    glEnd();
}

void glarea_event_proc (He_node *hn, He_event *hev)
{
    switch (hev->type) {

        case ButtonPress :
            HeTbMemoPointer (&info_tb, hev);
            break;

        case MotionNotify :
            switch (hev->sb) {
                case 1 : HeTbEventRotations(&info_tb, hn, hev);
                         HePostRepaint(hn);
                         break;
                case 2 : HeTbEventFovy (&info_tb, hn, hev);
                         HePostRepaint(hn);
                         break;
                case 3 : HeTbEventZdist (&info_tb, hn, hev);
                         HePostRepaint(hn);
                         break;
            }
            break;
    }
}

int main (int argc, char *argv[])
{
    int attr_list[] = { GLX_RGBA,           /* Codage couleurs */
                        GLX_DOUBLEBUFFER,   /* Double buffer   */
                        GLX_DEPTH_SIZE, 1,  /* Zbuffer         */
                        None };

    HeInit (&argc, &argv);

    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Demo du trackball");
    HeSetFrameResizeProc (princ, princ_resize);

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "Quit", butt_proc, NULL);
    HeFit(panel);
```

```
    glarea = HeCreateGLArea (princ, attr_list, NULL);
    HeSetY (glarea, HeGetHeight(panel)+2);
    HeSetWidth (glarea, 500);
    HeSetHeight (glarea, 400);
    HeSetGLAreaInitProc (glarea, glarea_init_proc);
    HeSetGLAreaRepaintProc (glarea, glarea_repaint_proc);
    HeSetGLAreaEventProc (glarea, glarea_event_proc);

    HeFit (princ);

    return HeMainLoop (princ);
}
```

Voici des explication sur ce qui est fait dans cet exemple. Pour utiliser les fonctions du trackball on déclare une variable `info_tb` de type `He_trackball` que l'on passe par adresse à toutes les fonctions. L'initialisation du trackball est faite dans l'InitProc, avec les lignes

```
HeTbInitRotations(&info_tb);
HeTbInitFovy(&info_tb, 60, 5, 120);
HeTbInitZdist(&info_tb, 4, 1, 10);
```

La première ligne initialise les angles de rotation ; la seconde ligne initialise l'angle de vue à 60 degrés, et fixe l'intervalle de 5 à 120 pour cet angle ; la troisième ligne initialise la distance de l'observateur à la scène à 4, et fixe l'intervalle de distance de 1 à 10 (l'ordre de grandeur de ces distances est relatif à l'ordre de grandeur des coordonnées de la scène dans l'espace).

La EventProc permet de traduire les mouvements de la souris en mouvements du trackball :

```
switch (hev->type) {

    case ButtonPress :
        HeTbMemoPointer (&info_tb, hev);
        break;

    case MotionNotify :
        switch (hev->sb) {
            case 1 : HeTbEventRotations(&info_tb, hn, hev);
                     HePostRepaint(hn);
                     break;
            case 2 : HeTbEventFovy (&info_tb, hn, hev);
                     HePostRepaint(hn);
                     break;
            case 3 : HeTbEventZdist (&info_tb, hn, hev);
                     HePostRepaint(hn);
                     break;
        }
        break;
}
```

Lors du ButtonPress on mémorise le point de départ du clic souris. Dans le MotionNotify on assigne un bouton à chaque fonctionnalité du trackball (on peut donc changer cette as-



signation, ou les utiliser à d'autres fins). Chaque appel à `HeTbEvent*` est suivi par un appel à `HePostRepaint` pour mettre à jour l'affichage avec la nouvelle vue. Les transformations sont toutes faites dans la RepaintProc :

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective( HeTbGetFovy(&info_tb),
                (GLdouble) HeGetWidth(hn)/HeGetHeight(hn),
                1, 100 );
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslated (0, 0, -HeTbGetZdist(&info_tb));
HeTbMultRotations(&info_tb);
```

On utilise `HeTbGetFovy` pour connaître l'angle de vue, `HeTbGetZdist` pour avoir la distance d'éloignement, puis `HeTbMultRotations` pour appliquer les rotations. Attention, ces transformations doivent se faire exactement dans cet ordre.

## 8.10 Exemples

Le programme `gldemo/nurbs.c` permet de tester les différents modes d'échantillonnage des nurbs. Le source est commenté.

## 8.11 En savoir plus sur OpenGL

Le guide officiel pour apprendre OpenGL est en anglais : "OpenGL Programming Guide", chez Addison Wesley.

Cette page web à l'EPFL contient de nombreux pointeurs sur des documents pour approfondir OpenGL ; on y trouve en particulier les ouvrages de référence de Addison-Wesley en ligne : le guide de programmation, et le manuel de référence (en anglais).

On trouve bien entendu une mine d'information chez SGI et sur opengl.org .

# Chapitre 9

# Compléments

Dans ce chapitre on présente les données attachées en ClientData, les DestroyProc, les arguments de la ligne de commande et les resources X11, les signaux Unix, et des compléments sur les widgets.

## 9.1 Une donnée ClientData

Parmi les propriétés communes des widget, il y a une propriété « fourre-tout » qui permet de stocker sur un `void*` une adresse quelconque (string, adresse d'un struct, d'un tableau, etc, ou un `int` en le castant) dont votre programme a besoin. C'est la propriété ClientData.

Pour attacher une donnée `client_data` à un widget `hn` :

```
HeSetClientData (hn, client_data);
```

On peut retrouver cette donnée plus tard, par exemple dans une callback, avec

```
client_data = HeGetClientData (hn);
```

On rappelle qu'en C ANSI, le type `void*` est compatible avec tous les types pointeurs, et qu'il faut éviter chaque fois que possible de faire un cast pour bénéficier pleinement des vérifications du compilateur.

Dans l'exemple `clientdata.c`, on crée deux boutons "Asterix" et "Obelix". On attache en ClientData une légende (un string) à chaque bouton. Lorsque l'un des boutons est pressé, la callback `affi_legende` retrouve la légende dans le ClientData du bouton, puis l'affiche.

```
/* examples/misc/clientdata.c */

#include <helium.h>

He_node *princ, *panel, *butt1, *butt2;

void affi_legende (He_node *hn)
{
    char *nom     = HeGetButtonLabel(hn),
         *legende = HeGetClientData (hn);
```





```
    printf ("%s : %s\n", nom, legende);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Client Data");

    panel = HeCreatePanel (princ);

    butt1 = HeCreateButton (panel);
    HeSetButtonLabel(butt1, "Asterix");
    HeSetClientData (butt1, "petit guerrier malin et rusé");
    HeSetButtonNotifyProc (butt1, affi_legende);

    butt2 = HeCreateButton (panel);
    HeSetButtonLabel(butt2, "Obelix");
    HeSetClientData (butt2, "gros livreur de menhirs");
    HeSetButtonNotifyProc (butt2, affi_legende);

    return HeMainLoop (princ);
}
```

On a vu dans la section « 2.6 Raccourci pour le bouton » la fonction HeCreateButtonP ; le quatrième paramètre de la fonction, que l'on avait laissé à NULL, sert en fait pour attacher un ClientData. Ainsi, le code

```
butt1 = HeCreateButton (panel);
HeSetButtonLabel(butt1, "Asterix");
HeSetClientData (butt1, "petit guerrier malin et rusé");
HeSetButtonNotifyProc (butt1, affi_legende);
```

est équivalent à

```
butt1 = HeCreateButtonP (panel, "Asterix", affi_legende,
                         "petit guerrier malin et rusé");
```

Dans la section suivante, on voit comment libérer automatiquement un ClientData lors de la destruction d'un widget.

## 9.2 Destruction d'un widget

On a vu dans la section « 2.4 Principe général » que pour détruire un widget `hn` il suffit d'appeler `HeDestroy(hn)`. En fait cet appel détruit récursivement `hn` et tous ses fils.

De même, un appel à HeQuit(0) provoque la destruction de tous les widgets du programme.

On peut bien sûr appeler `HeDestroy` ou `HeQuit` depuis n'importe quelle callback, par exemple depuis la callback d'un bouton ; on peut aussi détruire un widget depuis sa propre callback sans problème.

On a la possibilité d'attacher à un widget une callback qui sera appelée à la destruction de ce widget : c'est la callback DestroyProc. Pour attacher une DestroyProc à un widget `hn` on fait :

```
HeSetDestroyProc (hn, destroy_proc);
```

Le prototype de la callback est

```
void destroy_proc (He_node *hn)
{
}
```

où `hn` est le widget allant être détruit (il existe encore au moment de l'appel de la callback).

Dans l'exemple suivant, on crée quelques widgets auxquels on attache à chacun une DestroyProc, chacune affichant un message lors de son appel. Dans la callback de l'un des boutons on détruit le bouton lui-même, dans une autre on détruit le Panel.

```
/* examples/misc/destroy.c */

#include <helium.h>

He_node *princ, *panel, *butt1, *butt2, *butt3;

void butt_destroy_proc (He_node *hn)
{
    printf ("butt_destroy_proc %s\n", HeGetButtonLabel(hn));
}

void panel_destroy_proc (He_node *hn)
{
    printf ("panel_destroy_proc\n");
}

void princ_destroy_proc (He_node *hn)
{
    printf ("princ_destroy_proc\n");
}

void butt1_proc (He_node *hn)
{
    HeDestroy (hn);
}

void butt2_proc (He_node *hn)
{
    HeDestroy (panel);
}

void butt3_proc (He_node *hn)
{
    HeQuit(0);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "DestroyProc");
    HeSetDestroyProc (princ, princ_destroy_proc);

    panel = HeCreatePanel (princ);
    HeSetDestroyProc (panel, panel_destroy_proc);

    butt1 = HeCreateButtonP (panel, "Suicide", butt1_proc, NULL);
```



```
    HeSetDestroyProc (butt1, butt_destroy_proc);
    butt2 = HeCreateButtonP (panel, "Détruit Panel", butt2_proc, NULL);
    HeSetDestroyProc (butt2, butt_destroy_proc);
    butt3 = HeCreateButtonP (panel, "Quit", butt3_proc, NULL);
    HeSetDestroyProc (butt3, butt_destroy_proc);

    return HeMainLoop (princ);
}
```

Helium garantit que l'appel de la DestroyProc est unique. Un usage classique de la DestroyProc est le suivant : lorsqu'une donnée est allouée avec un `malloc`, et que son adresse n'est mémorisée que dans un ClientData, on attache une DestroyProc pour libérer automatiquement la mémoire du ClientData à la destruction du widget.

Voici un exemple de DestroyProc qui fait un `free` sur le ClientData :

```
void destroy_proc (He_node *hn)
{
    void *client_data = HeGetClientData (hn);
    free (client_data);
}
```

## 9.3 Arguments de la ligne de commande

Helium propose des fonctions qui décodent la ligne de commande. Par défaut, Helium reconnait déjà un certain nombre d'options, telles que "`-help`", "`-res`", "`-display`", etc. Pour connaître la liste des options disponibles, il suffit de lancer votre programme avec l'option "`-help`".

Le décodage des options de la ligne de commande se fait au moment du

```
HeInit (&argc, &argv);
```

Helium parcourt la ligne de commande, et supprime de `argc` et `argv` toutes les options reconnues. Autrement dit, après l'appel de HeInit, il ne reste dans `argc` et `argv` que les arguments qui n'ont pas été reconnus par Helium (remarque : les variables globales `he_argc` et `he_argv` conservent la ligne de commande originale ).

Vous pouvez alors décoder les arguments restants avec vos propres fonctions. Vous pouvez également profiter des fonctions de Helium pour décoder ces arguments. L'avantage est que vos nouvelles options seront affichées avec les options d'Helium lorsque vous lancez votre programme avec l'option "`-help`".

On procède en deux temps : avant l'appel de HeInit on déclare les nouvelles options ; après l'appel de HeInit on teste la présence des options dans la ligne de commande.

Pour déclarer une nouvelle option "`-pomme`" on fait

```
HeDeclArg ("-pomme", NULL, NULL, 0, NULL);
```

Pour qu'il y ait une petite légende affichée lors du "`-help`" on fait

```
HeDeclArg ("-pomme", NULL, NULL, 0, "petite légende");
```

Pour tester la présence de l'option on fait

```
if (HeGetOption("-pomme", NULL)) { ... }
```

On peut aussi déclarer une option avec un ou plusieurs paramètres, ainsi qu'une petite légende sur l'option et sur le ou les paramètres ; par exemple

```
HeDeclArg ("-poire", NULL, "<variété>", 1, "petite légende");
HeDeclArg ("-scoubidou", NULL, "<couleur longueur>", 2, "légende");
```

Lorsque l'option figure dans la ligne de commande, la présence de son ou de ses paramètres est obligatoire (sinon Helium affiche un message d'erreur). Pour récupérer le paramètre 1 de l'option on fait

```
char *p = HeGetOption("-poire", NULL);
```

Pour récupérer le paramètre numéro `i` de l'option on fait

```
char *p = HeGetOptionN("-scoubidou", i, NULL);
```

Le deuxième paramètre de HeGetOption (le troisième de HeGetOptionN) est la valeur renvoyée à `p` si l'option n'est pas présente dans la ligne de commande. Donc ici `p` est NULL si l'option est absente. Dans l'exemple suivant

```
char *p = HeGetOption("-poire", "williams");
```

`p` vaut "`williams`" si l'option est absente, sinon `p` vaut le string de l'option.

Voici un exemple récapitulatif :

```
/* examples/misc/args.c */

#include <helium.h>

He_node *princ;

int main (int argc, char *argv[])
{
    int i;
    char *tmp;

    /* Declarations d'options supplementaires a Helium */
    HeDeclArg ("-zero", NULL, NULL, 0, "sans param");
    HeDeclArg ("-un", NULL, "<param1>", 1, "un param");
    HeDeclArg ("-deux", NULL, "<param1 param2>", 2, "deux params");

    /* Affichage des arguments recus avant analyse */
    printf ("Args AVANT HeInit() :\n");
    for (i = 0; i < argc; i++)
        printf ("  %2d  \"%s\"\n", i, argv[i]);

    /* Initialisation d'Helium, ouverture du display et analyse des
       arguments : toutes les options reconnues sont enlevees de argv
       avec leurs parametres */
    HeInit (&argc, &argv);
```



```
    /* Affichage des arguments restant apres analyse */
    printf ("Args APRES HeInit() :\n");
    for (i = 0; i < argc; i++)
        printf ("  %2d  \"%s\"\n", i, argv[i]);

    /* Affichages des options supplementaires reconnues */
    printf ("Options reconnues :\n");
    tmp = HeGetOption ("-zero", NULL);
    if (tmp) printf ("-zero\n");
    tmp = HeGetOption ("-un", NULL);
    if (tmp) printf ("-un \"%s\"\n", tmp);
    tmp = HeGetOption ("-deux", NULL);
    if (tmp) printf ("-deux \"%s\" \"%s\"\n",
        tmp, HeGetOptionN("-deux", 2, ""));

    /* Creation d'une fenetre */
    princ = HeCreateFrame();
    HeSetFrameLabel (princ, "Args");

    /* Boucle d'evenements */
    return HeMainLoop (princ);
}
```

Les ressources de X11 sont des sortes d'options permanentes, très souvent stockées dans le fichier texte `$HOME/.Xdefaults`. Pour connaître les ressources reconnues par Helium, il suffit de lancer votre programme avec l'option "`-res`".

Pour rajouter une ressource dans Helium il suffit de la donner en second paramètre à HeDeclArg (on avait jusqu'à présent laissé ce paramètre à NULL). Une ressource peut avoir au maximum un paramètre.

On peut déclarer une ressource qui correspond à une option de la ligne de commande, ou une ressource qui ne correspond à aucune option de la ligne de commande. Dans le premier cas, la présence de l'option supplante la valeur de la ressource.

Pour connaître le paramètre d'une option/ressource (ou leur présence), on utilise HeGetOption ; pour connaître le paramètre d'une ressource seule, on utilise HeGetRessource.

Si on modifie le fichier `$HOME/.Xdefaults`, il ne faut pas oublier de taper `xrdb -load $HOME/.Xdefaults` pour mettre à jour le serveur X11 (sinon les changements ne sont pas pris en compte).

On peut lire les nombreux commentaires dans `gui/args.c` pour en savoir plus sur les fonctions gérant les options et les ressources dans Helium.

## 9.4 Timeout

On peut demander à Helium d'appeler périodiquement une callback TimeoutProc tous les $n$ millisecondes. Il suffit d'appeler

```
HeAddTimeout (interval, timeout_proc, data);
```

où `interval` est la durée de la période en millisecondes, `timeout_proc` est la callback et `data` est un `void*`. La fonction `HeAddTimeout` renvoie -1 en cas d'erreur, sinon renvoie un entier `handle` > 0 qui sert à indentifier le Timeout. Le prototype de la callback TimeoutProc est

```
int timeout_proc (int handle, void *data)
```

Lorsqu'elle est appelée, la callback reçoit le numéro `handle` du Timeout et la donnée `data` passée en argument à `HeAddTimeout`. La callback doit renvoyer TRUE ou FALSE, pour indiquer si le Timeout doit être supprimé (FALSE) ou réarmé (TRUE). Attention, le décompte du temps commence au retour de la callback (ce n'est pas du temps réel).

Pour supprimer un Timeout en attente, on fait

```
HeRemoveTimeout (handle);
```

La fonction renvoie 0 en cas de succès, -1 en cas d'erreur (mauvais handle). Chaque handle de Timeout est unique ; on peut donc demander de supprimer un Timeout même s'il n'existe déjà plus, sans risquer de supprimer un autre Timeout par mégarde.

Dans l'exemple suivant, on enclenche un Timeout à 1000 ms, qui affiche l'heure en se servant de la fonction système `time()` :

```
/* examples/misc/horloge.c */

#include <helium.h>

He_node *princ, *panel, *mess;
int han1;

int proc1 (int h, void *data)
{
    time_t pt;

    time(&pt);
    HeSetMessageLabel (mess, ctime(&pt));
    return TRUE ;
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Horloge");

    panel = HeCreatePanel (princ);
    mess = HeCreateMessageP (panel, "Quelle heure est-il ?", FALSE);

    han1 = HeAddTimeout (1000, proc1, NULL);

    return HeMainLoop (princ);
}
```

Complément : la fonction `HeGetTime()` renvoie le temps absolu (obtenu en interne avec `gettimeofday()`) sur un double, en secondes virgule microsecondes. Ceci peut permettre de mesurer un taux de transfert ou la durée d'exécution d'une fonction ; voir par exemple : `demo/hsv.c`

## 9.5 Bulles et petits conseils

Une bulle est une petite fenêtre sans décoration, dans laquelle est affiché un petit texte d'une ou plusieures lignes. La bulle se met automatiquement à la taille du texte ; elle



disparait lorsque l'utilisateur clique dessus.

Le code suivant crée une bulle avec un texte sur deux lignes :

```
He_node *bubble = HeCreateBubble ();
HeSetBubbleLabel (bubble, "Ceci est un texte\nsur deux lignes");
```

La chaîne passée à `HeSetBubbleLabel` peut contenir des `'\n'`, qui sont interprétés comme des sauts de lignes. On peut récupérer le texte de la bulle avec

```
char *label = HeGetBubbleLabel (bubble);
```

Pour afficher la bulle, on peut se servir des fonctions `HeSetX`, `HeSetY` et `HeSetShow`, sachant que les coordonnées de la bulle doivent être données en coordonnées absolues pour l'écran. La fonction suivante simplifie l'affichage et le positionnement de la bulle :

```
HeShowBubbleXYF (bubble, ref, x, y, flag);
```

lorsque `ref` est non NULL et `flag` est FALSE, affiche la bulle `bubble` aux coordonnées `x,y` relatives au widget `ref`. Si `ref` est NULL, affiche la bulle en coordonnées `x,y` absolues par rapport à l'écran. Si `flag` est TRUE, ignore `y` et place la bulle juste en dessous de `ref`. La fonction recadre également la bulle pour qu'elle apparaisse entièrement à l'écran.

Dans ce premier exemple, on crée une bulle et un bouton, et on affiche la bulle en dessous du bouton lorsque le bouton est pressé :

```
/* examples/misc/bulle1.c */

#include <helium.h>

He_node *princ, *panel, *bulle;

void affi_bulle (He_node *hn)
{
    HeShowBubbleXYF (bulle, hn, 50, 0, TRUE);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Bulle 1");

    panel = HeCreatePanel (princ);
    HeCreateButtonP (panel, "Affiche bulle", affi_bulle, NULL);

    bulle = HeCreateBubble ();
    HeSetBubbleLabel (bulle, "Cliquez dans cette bulle\n"
                             "pour la faire disparaître");

    return HeMainLoop (princ);
}
```

Dans ce deuxième exemple, on crée une bulle et un canvas ; la bulle est utilisée pour afficher les coordonnées de la souris lorsqu'elle est tirée dans le canvas :

```
/* examples/misc/bulle2.c */

#include <helium.h>

He_node *princ, *canvas, *bulle;

void canvas_event (He_node *hn, He_event *hev)
{
    switch (hev->type) {
        case ButtonPress :
        case MotionNotify :
            if (hev->sb == 1) {
                char bla[80];
                sprintf (bla, "Bulle \nen %d,%d", hev->sx, hev->sy);
                HeSetBubbleLabel (bulle, bla);
                HeShowBubbleXYF (bulle, hn, hev->sx, hev->sy, FALSE);
            }
            break;
        case ButtonRelease :
            if (hev->sb == 1)
                HeSetShow (bulle, FALSE);
            break;
    }
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Bulle 2");

    canvas = HeCreateCanvas (princ);
    HeExpand (canvas, NULL, HE_BOTTOM_RIGHT);
    HeSetCanvasEventProc (canvas, canvas_event);

    bulle = HeCreateBubble ();

    return HeMainLoop (princ);
}
```

Les petits conseils (tips), encore appelés bulles d'aides, sont des petits messages qui apparaissent dans des bulles lorsque la souris survole certains widgets. Plus exactement, le conseil est affiché 500 millisecondes après l'entrée de la souris dans un widget ; il est effacé lorsque la souris sort du widget, si un bouton ou une touche est pressé, ou si la souris passe sur le conseil lui-même.

Pour attacher un petit conseil à un widget `hn` il suffit de faire

```
HeSetTip (hn, "Ceci est un petit conseil");
```

On peut retrouver le conseil attaché à `hn` avec

```
char *label = HeGetTip (hn);
```

Pour supprimer le conseil attaché à `hn`, on fait simplement

```
HeSetTip (hn, NULL);
```

L'exemple suivant montre quels types de widgets supportent ce mécanisme :



```
/* examples/misc/tips.c */

#include <helium.h>

He_node *princ, *panel, *canvas, *tmp;

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Petits conseils (tips)");

    panel = HeCreatePanel (princ);

    tmp = HeCreateButtonP (panel, "Bouton", NULL, NULL);
    HeSetTip (tmp, "Ceci est un bouton");

    tmp = HeCreateToggleLedP (panel, "Led", NULL, TRUE);
    HeSetTip (tmp, "Ceci est un led");

    tmp = HeCreateMessageP (panel, "Message", TRUE);
    HeSetTip (tmp, "Ceci est un message");

    HeSetPanelLayout (panel, HE_LINE_FEED);

    tmp = HeCreateText (panel);
    HeSetTextValue (tmp, "Champ de saisie");
    HeSetTip (tmp, "Ceci est un champ text");

    HeFit (panel);

    canvas = HeCreateCanvas (princ);
    HeJustify (canvas, panel, HE_TOP);
    HeExpand (canvas, NULL, HE_BOTTOM_RIGHT);
    HeSetTip (canvas, "Les tips fonctionnent aussi\n"
                      "pour les Canvas et les GLArea");

    return HeMainLoop (princ);
}
```

## 9.6 Interception des signaux Unix

On peut utiliser les fonctions Unix `signal` ou `sigaction` pour capter un signal Unix et lui assigner un "handler de signal", c'est-à-dire une callback appelée par le système Unix chaque fois que le signal est reçu.

L'inconvénient est que ce handler peut être appelé à un "mauvais moment" pour le toolkit Helium, pendant lequel un appel aux fonctions de Helium ou de Xlib aurait des effets imprévisibles.

Nous conseillons donc d'utiliser les fonctions de Helium : pour capter un signal sig et attacher une callback SignalProc au signal, on fait

```
HeAddSignal (sig, sig_proc, data);
```

où `sig_proc` est la callback et data est un `void*`. La fonction HeAddSignal renvoie 0 si l'attachement a réussi, -1 en cas d'erreur (par exemple si le signal n'existe pas). Le prototype de la callback est

```
void sig_proc (int sig, void *data)
```

où `sig` est le signal capté et `data` est la donnée attachée avec HeAddSignal. Helium garantit que la callback sera appelée dans un moment "sans risque", comme toute autre callback de Helium. À noter, si un même signal est capté plusieurs fois de suite dans un très court labs de temps, Helium n'appelle qu'une seule fois la callback (comme le fait le noyau Unix).

Pour supprimer le mécanisme de capture sur un signal `sig` (et rétablir le handler Unix par défaut) on appelle la fonction suivante, qui renvoie 0 en cas de succès et -1 en cas d'erreur :

```
HeRemoveSignal (sig);
```

Helium ne se réserve aucun signal ; en particulier, le mécanisme des Timeout est gêré par un `select()` et non avec le signal SIGALRM, ce qui vous laisse libre d'utiliser ce signal (mais le mécanisme des Timeout est beaucoup plus puissant).

Dans l'exemple suivant, on capte les signaux SIGHUP, SIGTERM et SIGINT et SIGUSR1. Lancer le programme et essayer `kill -HUP` ou `kill -TERM` ou `kill -INT` ou `kill -USR1` sur le pid du programme, ou encore Ĉ :

```
/* examples/misc/signal.c */

#include <helium.h>

He_node *princ, *panel;

void dialog_proc (char *name, void *client_data)
{
    if (!strcmp (name, "Oui")) HeQuit(0);
}

void ask_quit (He_node *hn)
{
    HeSimpleDialog (
        HE_DIALOG_BELL,
        HE_DIALOG_TITLE,   "Attention",
        HE_DIALOG_MESSAGE, "Etes-vous certain",
        HE_DIALOG_MESSAGE, "de vouloir quitter ?",
        HE_DIALOG_BUTTON,  "Oui",
        HE_DIALOG_BUTTOK,  "Annuler",
        HE_DIALOG_PROC,    dialog_proc,
        0);
}

void quit_proc (He_node *hn)
{
    ask_quit (NULL);
}

void sig1_proc (int sig, void *data)
{
    ask_quit (NULL);
}

void sig2_proc (int sig, void *data)
{
    printf ("sig2_proc: sig = %d, data = \"%s\"\n", sig, (char*)data);
}
```



```
int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Test signaux");
    HeSetFrameCloseProc (princ, ask_quit);

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Quit", quit_proc, NULL);

    HeFit(panel);
    HeFit(princ);

    printf ("pid: %d\n", getpid());
    HeAddSignal (SIGHUP,  sig1_proc, NULL);
    HeAddSignal (SIGTERM, sig1_proc, NULL);
    HeAddSignal (SIGINT,  sig1_proc, NULL);
    HeAddSignal (SIGUSR1, sig2_proc, "string passé en data");

    return HeMainLoop (princ);
}
```

## 9.7 Entrées-sorties

En général, l'écriture ou la lecture sur un descripteur de fichier est un appel bloquant, qui peut avoir pour effet de « geler » l'interface graphique de votre programme : rien à lire (tampon de lecture vide), ou impossible d'écrire (tampon d'écriture plein), etc. À éviter absolument !

Helium fournit un mécanisme qui permet d'être averti lorsqu'on peut réaliser une lecture, une écriture ou une lecture urgente sans être bloqué. L'appel

```
HeAddIO (fd, cond, io_proc, data);
```

demande à Helium de scruter le descripteur de fichier `fd` et d'appeler la callback `io_proc` chaque fois que la condition `cond` est réalisée.

`fd` peut être un descripteur de fichier de toute nature (entrée ou sortie standard, tube, socket, etc).

`cond` est l'une des constantes `HE_IO_READ` (le `fd` est prêt en lecture), `HE_IO_WRITE` (le `fd` est prêt en écriture), `HE_IO_EXCEPT` (le `fd` reçoit les données urgentes, par exemple avec tcp), ou une combinaison bit-à-bit, par exemple `HE_IO_READ | HE_IO_WRITE`.

Le prototype de la callback IoProc est

```
void io_proc (int fd, Ulong rcond, void *data);
```

où `data` est la donnée passée en paramètre à `HeAddIO`, et `rcond` contient les conditions qui sont satisfaites (combinaison bit-à-bit). Pour savoir par exemple si la condition `HE_IO_WRITE` est satisfaite, on teste `(rcond & HE_IO_WRITE)` .

La callback est appelée chaque fois qu'il y a au moins une condition satisfaite ; on peut alors réaliser pendant l'appel de la callback l'opération de lecture ou d'écriture sans risquer d'être bloqué.

La fonction `HeAddIO` renvoie -1 en cas d'erreur, 0 sinon. Chaque appel de `HeAddIO` sur un `fd` écrase le précédent. Pour supprimer les scrutations sur un `fd`, on appelle

```
HeRemoveIO (fd);
```

qui renvoie également -1 en cas d'erreur, 0 sinon.

Dans l'exemple suivant, on scrute l'entrée standard et on affiche dans un champ de saisie Text chaque ligne tapée dans la console :

```
/* examples/misc/conso.c */

#include <helium.h>

He_node *princ, *panel, *text;

void lecture_proc (int fd, Ulong rcond, void *data)
{
    char buf[256];

    read (fd, buf, 256);
    HeSetTextValue (text, buf);
}

int main (int argc, char *argv[])
{
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, "Lecture console");

    panel = HeCreatePanel (princ);
    text = HeCreateText (panel);
    HeSetTextValue (text, "Tapez du texte dans la console");

    /* fd=0 correspond à stdin */
    HeAddIO (0, HE_IO_READ, lecture_proc, NULL);

    return HeMainLoop (princ);

}
```

Dans cet exemple on ouvre un tube, on fait un `fork()` pour dupliquer le processus, et le père envoie des octets au fils. On mesure les taux de transfert à l'aide d'un Timeout tous les 200 ms.

```
/* examples/misc/tube.c */

#include <helium.h>

He_node *princ, *panel, *mess1, *mess2, *mess3;
int fork_val, timeout1, nb_oper = 0;
double nb_octets = 0, temps0;

/* Le taux de transfert dépend de BUFMAX. Si BUFMAX
   est trop grand, la lecture se fait en plusieurs fois */
#define BUFMAX 4096
char buf[BUFMAX];

void tube_write (int fd, Ulong rcond, void *data)
{
```



```
    int n = write (fd, buf, BUFMAX);
    nb_octets += n;
    nb_oper++;
}

void tube_read (int fd, Ulong rcond, void *data)
{
    int n = read (fd, buf, BUFMAX);
    nb_octets += n;
    nb_oper++;
}

int timeout1_proc (int h, void *data)
{
    char bla[80];

    sprintf (bla, "%.0f", nb_octets / 1024);
    HeSetMessageLabel (mess1, bla);

    sprintf (bla, "%.0f", (nb_octets / 1024) / (HeGetTime()-temps0));
    HeSetMessageLabel (mess2, bla);

    sprintf (bla, "%.0f", (double) nb_oper / (HeGetTime()-temps0));
    HeSetMessageLabel (mess3, bla);

   return TRUE ;
}

void butt_quit (He_node *hn)
{
    HeQuit(0);
}

int main (int argc, char *argv[])
{
    int tube[2];

    /* Création tube */
    if (pipe(tube) < 0) { perror("pipe"); exit(1); }

    /* fils = 0, père != 0 */
    fork_val = fork();

    /* Le processus propriétaire d'un connection X doit être unique ;
       il faut donc impérativement que le fork() ait lieu AVANT HeInit,
       car sinon le père et le fils auraient la même connection X. */
    HeInit (&argc, &argv);

    princ = HeCreateFrame ();
    HeSetFrameLabel (princ, fork_val == 0 ?
        "Tube réception (fils)" : "Tube émission (père)");

    panel = HeCreatePanel (princ);

    HeCreateButtonP (panel, "Quit", butt_quit, NULL);
    HeSetPanelLayout (panel, HE_LINE_FEED);
    HeCreateMessageP (panel, fork_val == 0 ?
        "K-octets reçus     :" : "K-octets envoyés   :", TRUE);
    mess1 = HeCreateMessage(panel);
    HeSetPanelLayout (panel, HE_LINE_FEED);
    HeCreateMessageP (panel, "K-octets/seconde   :", TRUE);
    mess2 = HeCreateMessage(panel);
    HeSetPanelLayout (panel, HE_LINE_FEED);
    HeCreateMessageP (panel, "opérations/seconde :", TRUE);
    mess3 = HeCreateMessage(panel);
```

```
    if (fork_val == 0)
         HeAddIO (tube[0], HE_IO_READ,  tube_read,  NULL);
    else HeAddIO (tube[1], HE_IO_WRITE, tube_write, NULL);

    timeout1 = HeAddTimeout (200, timeout1_proc, NULL);
    temps0 = HeGetTime();

    return HeMainLoop (princ);

}
```

Compléments : le programme `demo/tcp.c` est un exemple de client/serveur TCP/IP. La partie client permet de dialoguer avec tout serveur TCP/IP (un alter ego, démons httpd, sendmail, in.pop3d, etc) ; la partie serveur répond à tout client TCP/IP (un alter ego, telnet, etc).

## 9.8 A propos des widgets

Helium est conçu sur un modèle objet très simple, qui est suffisant pour ce que l'on veut en faire, et ne demande pas de mécanismes complexes pour gérer par exemple l'héritage.

Les widgets sont ainsi des objets sur deux niveaux ; le premier niveau contient les propriétés communes à tous les widgets, telles que taille et position, visible/caché, actif/inactif, ClientData, etc ; le second niveau contient les propriétés et les méthodes spécifiques à une classe donnée de widgets, tel que le nom d'un bouton, Layout d'un panel, callback d'un champ de saisie, etc.

Dans cette logique, toutes les propriétés Prop du premier niveau sont accessible par

```
HeSetProp(hn, prop);
prop = HeGetProp (hn);
```

tandis que toutes les propriétés de second niveau Prop spécifiques à une classe Class sont accessibles par

```
HeSetClassProp(hn, prop);
prop = HeGetClassProp (hn);
```

Dans les sources d'Helium, tous les prototypes de fonctions sont déclarés dans les fichiers .h du répertoire gui/ . La plupart d'entre elles ont été vues auparavant dans le tutorial ; voici quelques fonctions complémentaires :

```
void HeSetBorder (He_node *hn, int border);
int HeGetBorder (He_node *hn);
```

utile essentiellement pour les classes Panel et Canvas : ajoute un bord noir autour du widget (c'est en fait le bord du Window X11 associé).

```
Window HeGetWindow (He_node *hn);
```

très utile, renvoie le Window X11 dans lequel est dessiné le widget ; par exemple, dans un Canvas (zone de dessin Xlib), c'est le Window dans lequel on va dessiner. Pour un bouton,



le Window que l'on obtient est en fait le Window du Panel propriétaire du bouton, car un bouton n'est pas un Window X11.

```
void HeBoundingBox (He_node *hn, int *xmax, int *ymax);
```

calcule la boîte englobante des fils du widget `hn` (sur un seul niveau) ; utilisé dans `HeFit`.

```
He_node *HeGetOwner (He_node *hn);
He_node *HeGetNext (He_node *hn);
He_node *HeGetSub (He_node *hn);
```

renvoient le widget du père (le propriétaire), d'un frère ou d'un fils du widget `hn` ; peu utilisé, sauf en interne.

```
Ulong HeGetNat (He_node *hn);
```

renvoie un identificateur entier de la classe du widget `hn` (voir `include/attr.h`).

Lorsqu'on mémorise un widget, il se peut que l'on ne sache pas s'il existera encore lorsqu'on voudra l'utiliser. Dans ce cas on demande à Helium de fournir un identificateur entier unique `id` du widget `hn`

```
int id = HeGetId (He_node *hn);
```

et on stocke l'entier `id`. Lorsqu'on voudra utiliser le widget stocké, on commencera par demander à Helium de retrouver le `hn` correspondant :

```
He_node *hn = HeGetNodeFromId (int id);
```

Si le résultat est NULL, c'est que le widget n'existe plus.

Pour balayer tous les widgets en mémoire on utilise `HeNodeScan` :

```
for (i = 0; hn = HeNodeScan(i) ; i++)
```

## 9.9 Dont je n'ai pas encore parlé

De nombreuses fonctions ne sont pas encore documentées dans le tutorial. En attendant, on peut consulter les sources de libHelium, abondament commentés, situés dans le répertoire `gui/`.