Didacticiel

Table of Contents

Table of Contents

Table of Contents

Didacticiel

Ian Main, slow@intergate.bc.ca

January 24, 1998.

1. Introduction

2. Bien débuter

3. Continuons

4. Placement des widgets

5. Vue d'ensemble des widgets

6. Widgets boutons

7. Widgets divers

17. glib

18. Fichiers rc de GTK

19. Écriture de vos propres widgets

20. Scribble, un programme simple de dessin

21. Conseils pour l'écriture d'applications GTK

22. Contributions

23. Remerciements

24. Copyright

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

1. Introduction

GTK (GIMP Toolkit) a été d'abord développé pour être une boîte à outils pour GIMP (General Image Manipulation Program). GTK est construit sur GDK (GIMP Drawing Kit) qui est, avant tout, une encapsulation des fonctions Xlib. On l'appelle « GIMP toolkit » car il fut créé pour développer GIMP, mais il est désormais utilisé dans plusieurs projets de logiciels libres. Les auteurs sont :

- Peter Mattis petm@xcf.berkeley.edu
- Spencer Kimball spencer@xcf.berkeley.edu
- Josh MacDonald jmacd@xcf.berkeley.edu

GTK est essentiellement une interface de programmation (API) orientée objet. Bien qu'il soit entièrement écrit en C, il est implanté en utilisant la notion de classes et de fonctions de rappel (pointeurs de fonctions).

Un troisième composant, appelé glib, remplace certains appels standard et comporte quelques fonctions supplémentaires pour gérer les listes chaînées, etc. Les fonctions de remplacement sont utilisées pour accroître la portabilité de GTK car certaines de ces fonctions, comme g_strerror(), ne sont pas disponibles ou ne sont pas standard sur d'autres Unix. D'autres comportent des améliorations par rapport aux versions de la libc : g_malloc(), par exemple, facilite le débuggage.

Ce didacticiel tente de décrire du mieux possible GTK, mais il n'est pas exhaustif. Il suppose une bonne connaissance du langage C, et de la façon de créer des programmes C. Il serait très précieux au lecteur d'avoir déjà une expérience de la programmation X, mais cela n'est pas nécessaire. Si l'apprentissage de GTK marque vos débuts dans l'approche des widgets, n'hésitez pas à faire des commentaires sur ce didacticiel et sur les problèmes qu'il vous a posé. Il y a aussi une API C++ pour GTK (GTK−−), si vous préférez utiliser ce langage, consultez plutôt la documentation qui la concerne. Une encapsulation en Objective C et des liaisons Guile sont également disponibles, mais ne seront pas abordées ici.

J'apprécierais beaucoup avoir un écho des problèmes que vous avez rencontré pour apprendre GTK à partir de ce document. De plus, toute suggestion sur son amélioration est la bienvenue.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

2. Bien débuter

La première chose à faire est, bien sûr, de récupérer les sources de GTK et de les installer. Vous pouvez en obtenir la dernière version sur `ftp.gimp.org` dans le répertoire `/pub/gtk`. D'autres sources d'informations se trouvent sur `http://www.gimp.org/gtk`. GTK utilise *autoconf* de GNU pour se configurer. Lorsque vous l'aurez détarré, tapez *./configure −−help* pour consulter la liste des options.

Pour commencer notre introduction à GTK, nous débuterons avec le programme le plus simple qui soit. Celui−ci créera une fenêtre de 200x200 pixels et ne pourra se terminer qu'en le tuant à partir du shell.

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

int main (int argc, char *argv[])
{
    GtkWidget *window;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

Tous les programmes inclueront évidemment le fichier `gtk/gtk.h` qui déclare les variables, fonctions, structures, etc. qui seront utilisées par votre application GTK.

La ligne :

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

appelle la fonction *gtk_init(gint *argc, gchar ***argv)* qui sera appelée dans toutes les applications GTK. Cette fonction configure certaines choses pour nous, comme l'aspect visuel et les couleurs par défaut, puis appelle *gdk_init(gint *argc, gchar ***argv)*. Cette dernière initialise la bibliothèque pour qu'elle puisse être utilisée, configure les gestionnaires de signaux par défaut et vérifie les paramètres passés à notre application via la ligne de commande en recherchant l'un des suivants :

- `--display`
- `--debug-level`
- `--no-xshm`
- `--sync`
- `--show-events`
- `--no-show-events`

Elle les supprime alors de la liste des paramètres, en laissant tout ce qu'elle ne reconnaît pas pour que notre application l'analyse ou l'ignore. Ceci crée un ensemble de paramètres standards acceptés par toutes les applications GTK.

Les deux lignes de code suivantes créent et affichent une fenêtre.

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
```

Le paramètre `GTK_WINDOW_TOPLEVEL` précise que l'on veut que la fenêtre créée suive l'aspect et le placement définis par le gestionnaire de fenêtres. Plutôt que de créer une fenêtre de 0x0, une fenêtre sans fenêtre fille est de 200x200 par défaut : on peut ainsi la manipuler facilement.

La fonction *gtk_widget_show()* informe GTK que l'on a configuré le widget et qu'il peut l'afficher.

La ligne suivante lance la boucle principale de traitement de GTK.

```
gtk_main ();
```

gtk_main() est un autre appel que vous verrez dans toute application GTK. Lorsque le contrôle atteind ce point, GTK se met en attente d'événements X (click sur un bouton, ou appui d'une touche, par exemple), de timeouts ou d'entrées−sorties fichier. Dans notre exemple simple, cependant, les événements sont ignorés.

2.1 « Bonjour tout le monde » en GTK

OK, écrivons un programme avec un widget (bouton). C'est le classique « Bonjour tout le monde » à la sauce GTK.

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

    /* fonction de rappel. Dans cet exemple, les paramètres sont ignorés...
     *  Les fonctions de rappel sont détaillées plus loin. */

void hello (GtkWidget *widget, gpointer data)
{
    g_print ("Bonjour tout le monde.\n");
}

gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
    g_print ("le signal delete_event est survenu.\n");

    /* Si l'on renvoit TRUE dans le gestionnaire du signal "delete_event",
     * GTK émettra le signal "destroy". Retourner FALSE signifie que l'on
     * ne veut pas que la fenêtre soit détruite.
     * Utilisé pour faire apparaître des boîtes de dialogue du type
     * « Êtes−vous sûr de vouloir quitter ? » */

    /* Remplacez FALSE par TRUE et la fenêtre principale sera détruite par
     * un signal « delete_event ». */

    return (FALSE);
}

/* Autre fonction de rappel */

void destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    /* GtkWidget est le type pour déclarer les widgets. */

    GtkWidget *window;
    GtkWidget *button;

    /* Cette fonction est appelée dans toutes les applications GTK.
     * Les paramètres passés en ligne de commande sont analysés et
     * retournés à l'application. */

    gtk_init (&argc, &argv);

    /* Création d'une nouvelle fenêtre. */

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Lorsque la fenêtre reçoit le signal "delete_event"
     * (envoyé par le gestionnaire de fenêtres en utilisant l'option
```

```
         * « close » ou la barre de titre), on lui demande d'appeler la
         * fonction delete_event() définie plus haut. La donnée passée en
         * paramètre à la fonction de rappel est NULL et est ignoré dans le
         * rappel. */

        gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                            GTK_SIGNAL_FUNC (delete_event), NULL);

        /* Ici, on connecte l'évenement "destroy" à un gestionnaire de signal.
         * Cet événement arrive lorsqu'on appelle gtk_widget_destroy() sur la
         * fenêtre, ou si l'on retourne TRUE dans le rappel "delete_event". */

        gtk_signal_connect (GTK_OBJECT (window), "destroy",
                            GTK_SIGNAL_FUNC (destroy), NULL);

        /* Configuration de la largeur du contour de la fenêtre. */

        gtk_container_border_width (GTK_CONTAINER (window), 10);

        /* Création d'un nouveau bouton portant le label
         * "Bonjour tout le monde". */

        button = gtk_button_new_with_label ("Bonjour tout le monde");

        /* Quand le bouton recevra le signal "clicked", il appellera la
         * fonction hello() définie plus haut en lui passant NULL en paramètre. */

        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (hello), NULL);

        /* Ceci provoquera la destruction de la fenêtre par appel de la
         * fonction gtk_widget_destroy(window) lors du signal "clicked".
         * Le signal de destruction pourrait venir de là, ou du
         * gestionnaire de fenêtres. */

        gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                                   GTK_SIGNAL_FUNC (gtk_widget_destroy),
                                   GTK_OBJECT (window));

        /* Insertion du bouton dans la fenêtre (container gtk). */

        gtk_container_add (GTK_CONTAINER (window), button);

        /* L'étape finale consiste à afficher ce nouveau widget... */

        gtk_widget_show (button);

        /* ... et la fenêtre. */

        gtk_widget_show (window);

        /* Toutes les applications GTK doivent avoir un gtk_main().
         * Le déroulement du programme se termine là et attend qu'un
         * événement survienne (touche pressée ou événement souris). */

        gtk_main ();

        return 0;
    }
```

2.2 Compilation de « Bonjour tout le monde »

Supposons que vous avez sauvegardé le code précédent dans un fichier nommé *bonjour.c*, pour le compiler tapez la commande suivante :

```
gcc -Wall -g bonjour.c -o bonjour_monde -L/usr/X11R6/lib \
    -lgtk -lgdk -lglib -lXext -lX11 -lm
```

Les bibliothèques invoquées ci-dessus doivent toutes être dans vos chemins de recherche par défaut, sinon, ajoutez `-L<library directory>` pour que *gcc* recherche dans ces répertoires les bibliothèques nécessaires. Sur mon système Debian GNU/Linux, par exemple, je dois ajouter `-L/usr/X11R6/lib` pour qu'il trouve les bibliothèques X11 (NdT : et c'est pareil sur mon système Red Hat Linux...).

L'ordre des bibliothèques est important. L'éditeur de liens doit connaître les fonctions d'une bibliothèque dont il a besoin avant de les traiter.

Si vous compilez en utilisant des bibliothèques statiques, l'ordre dans lequel vous listez les bibliothèques devient très important. L'exemple donné ci-dessus devrait fonctionner dans tous les cas.

Les bibliothèques que l'on utilise sont :

- La bibliothèque glib (`-lglib`), qui contient diverses fonctions. Seule *g_print()* est utilisée dans cet exemple. GTK est construit au dessus de *glib* et vous aurez donc toujours besoin de celle-ci. Voir la section concernant glib pour plus de détails.
- La bibliothèque GDK (`-lgdk`), l'enveloppe de Xlib.
- La bibliothèque GTK (`-lgtk`), la bibliothèque des widgets, construite au dessus de GDK.
- La bibliothèque Xlib (`-lX11` utilisée par GDK.
- La bibliothèque Xext (`-lXext`). Cette dernière contient le code pour les pixmaps en mémoire partagée et les autres extensions X.
- La bibliothèque mathématique (`-lm`). Elle est utilisée pour différentes raisons par GTK.

2.3 Théorie des signaux et des rappels

Avant de voir en détail le programme « Bonjour tout le monde », nous parlerons d'abord des événements et des fonctions de rappel. GTK est dirigé par les événements, ce qui signifie qu'il restera inactif dans *gtk_main* jusqu'à ce qu'un événement survienne et que le contrôle soit passé à la fonction appropriée.

Ce passage du contrôle est réalisé en utilisant le concept de « signal ». Lorsqu'un événement survient, comme l'appui sur un bouton, le signal approprié sera « émis » par le widget qui a été pressé. C'est de cette façon que GTK réalise la plupart de son travail. Pour qu'un bouton réalise une action, on configure un gestionnaire de signal pour capturer ces signaux et appeler la fonction adéquate. Ceci est fait en utilisant une fonction comme :

```
gint gtk_signal_connect (GtkObject *object,
                         gchar *name,
                         GtkSignalFunc func,
                         gpointer func_data);
```

Où le premier paramètre est le widget qui émettra le signal, et le deuxième est le nom du signal que l'on souhaite intercepter. Le troisième paramètre est la fonction que l'on veut appeler quand le signal est capturé, et le quatrième sont les données que l'on souhaite passer à cette fonction.

La fonction spécifiée par le troisième paramètre s'appelle une « fonction de rappel » et doit être de la forme :

```
void callback_func(GtkWidget *widget, gpointer *callback_data);
```

Où le premier paramètre sera un pointeur vers le widget qui a émis le signal, et le second un pointeur vers les données passées par le dernier paramètre de la fonction *gtk_signal_connect()* décrite plus haut.

Un autre appel utilisé dans l'exemple « Bonjour tout le monde » est :

```
gint gtk_signal_connect_object (GtkObject *object,
                                gchar  *name,
                                GtkSignalFunc func,
                                GtkObject *slot_object);
```

gtk_signal_connect_object() est la même chose que *gtk_signal_connect()* sauf que la fonction de rappel utilise un seul paramètre : un pointeur vers un objet GTK. Lorsqu'on utilise cette fonction pour connecter des signaux, le rappel doit être de cette forme :

```
void callback_func (GtkObject *object);
```

Où l'objet est d'ordinaire un widget. En général, on ne configure pas de rappels pour *gtk_signal_connect_object*. D'habitude, ceux−ci sont utilisés pour appeler une fonction GTK acceptant un simple widget ou objet comme paramètre, comme dans notre exemple.

La raison pour laquelle il y a deux fonctions pour connecter les signaux est simplement de permettre aux fonctions de rappel d'avoir un nombre différent de paramètres. De nombreuses fonctions de la bibliothèque GTK n'acceptent qu'un simple pointeur vers un *GtkWidget* comme paramètre et vous pouvez donc utiliser *gtk_signal_connect_object()* pour celles−ci, tandis que pour vos fonctions vous pouvez avoir besoin d'avoir de fournir plus de données aux fonctions de rappel.

2.4 « Bonjour tout le monde » pas à pas

Maintenant que nous connaissons la théorie, clarifions un peu en progressant à travers le programme « Bonjour tout le monde ».

Voici la fonction de rappel appelée lorsque le bouton est « clicked ». Dans notre exemple, on ignore le widget et les données mais il n'est pas difficile de faire quelque chose avec. Le prochain exemple utilisera le paramètre des données pour nous dire quel bouton a été pressé.

```
void hello (GtkWidget *widget, gpointer *data)
{
    g_print ("Bonjour tout le monde\n");
}
```

Cette fonction de rappel est un peu spéciale. L'événement "delete_event" survient lorsque le gestionnaire de fenêtres l'envoie à l'application. On doit choisir ce qu'il faut faire de ces événements. On peut les ignorer, leur donner une réponse, ou simplement quitter l'application.

La valeur que l'on retourne dans cette fonction de rappel permet à GTK de savoir ce qu'il a à faire. En retournant FALSE, on l'informe que l'on ne veut pas que le signal "destroy" soit émis, afin de laisser notre application tourner. En retournant TRUE, on lui demande d'émettre "destroy" qui appellera à son tour notre gestionnaire du signal "destroy".

```
gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
    g_print ("le signal delete_event est survenu.\n");

    return (FALSE);
```

```
}
```

Voici une autre fonction de rappel qui ne fait que quitter l'application en appelant *gtk_main_quit()*. Il n'y a pas grand chose de plus à dire car elle est plutôt triviale :

```
void destroy (GtkWidget *widget, gpointer *data)
{
    gtk_main_quit ();
}
```

Je suppose que vous connaissez la fonction *main()*... oui, comme les autres programmes C, toutes les applications GTK en ont une.

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

La partie qui suit déclare deux pointeurs sur des structures de type *GtkWidget*. Ceux−ci sont utilisés plus loin pour créer une fenêtre et un bouton.

```
    GtkWidget *window;
    GtkWidget *button;
```

Et revoici notre *gtk_init*. Comme précédemment, il initialise le toolkit et analyse les paramètres de la ligne de commande. Il supprime chaque paramètre reconnu de la liste et modifie *argc* et *argv* pour faire comme si ces paramètres n'avaient jamais existé, laissant notre application analyser les paramètres restants.

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

Création d'une nouvelle fenêtre. C'est plutôt classique. La mémoire est allouée pour une structure *GtkWidget* et *window* pointe donc sur celle−ci. Cela configure une nouvelle fenêtre, mais celle−ci ne sera pas affichée tant que l'on n'a pas appelé *gtk_widget_show(window)* vers la fin de notre programme.

```
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

Voici maintenant un exemple de connexion d'un gestionnaire de signal à un objet : la fenêtre. Le signal "destroy" est capturé. Il est émis lorsqu'on utilise le gestionnaire de fenêtres pour tuer la fenêtre (et que l'on retourne TRUE dans le gestionnaire "delete_event"), ou lorsqu'on utilise l'appel *gtk_widget_destroy()* en lui passant le widget *window* comme objet à détruire. Ici, on appelle juste la fonction *destroy()* définie ci−dessus avec le paramètre NULL, ce qui quitte GTK pour nous.

`GTK_OBJECT` et `GTK_SIGNAL_FUNC` sont des macros qui réalisent les conversions et les vérifications de types pour nous. Elles rendent aussi le code plus lisible.

```
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);
```

La fonction suivante sert à configurer un attribut d'un objet container. Elle configure simplement la fenêtre pour qu'elle ait une zone vide autour d'elle de 10 pixels de large où aucun widget ne pourra se trouver. Il existe d'autres fonctions similaires que nous verrons dans la section sur la Configuration des attributs des widgets

À nouveau, `GTK_CONTAINER` est une macro réalisant la conversion de type.

```
gtk_container_border_width (GTK_CONTAINER (window), 10);
```

Cet appel crée un nouveau bouton. Il alloue l'espace mémoire pour une nouvelle structure GtkWidget, l'initialise et fait pointer *button* vers elle. Ce bouton portera le label « Bonjour tout le monde » lorsqu'il sera affiché.

```
button = gtk_button_new_with_label ("Bonjour tout le monde");
```

Maintenant, prenons ce bouton et faisons lui faire quelque chose d'utile. On lui attache un gestionnaire de signal pour que, lorsqu'il émettra le signal "clicked", notre fonction *hello()* soit appelée. On ignore les paramètres et on ne passe donc que la valeur NULL à la fonction de rappel *hello()*. Évidemment, le signal "clicked" est émis lorsqu'on clique sur le bouton avec la souris.

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (hello), NULL);
```

On utilisera aussi ce bouton pour quitter notre programme, ce qui permettra d'illustrer la façon dont le signal "destroy" peut venir soit du gestionnaire de fenêtres, soit de notre programme. Quand le bouton est "clicked" comme cela est décrit plus haut, il appelle d'abord la fonction de rappel *hello()* puis celle−ci dans l'ordre dans lequel elles sont configurées. On peut avoir autant de fonctions de rappel que l'on désire, elles seront exécutées selon leur ordre de connexion. Puisque la fonction *gtk_widget_destroy()* n'accepte que *GtkWidget *widget* comme paramètre, on utilise ici la fonction *gtk_signal_connect_object()* à la place de *gtk_signal_connect()*.

```
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));
```

Voici un appel de placement, qui sera expliqué en détail plus tard, mais qui est plutôt facile à comprendre. Il indique simplement à GTK que le bouton doit être placé dans la fenêtre où il s'affichera.

```
gtk_container_add (GTK_CONTAINER (window), button);
```

Maintenant, nous avons tout configuré comme on le souhaitait : les gestionnaires de signaux sont en place et le bouton est mis dans la fenêtre où il doit se trouver. On demande alors à GTK de « montrer » les widgets à l'écran. Le widget *window* est affiché en dernier afin que la fenêtre entière surgisse d'un coup plutôt que voir d'abord la fenêtre s'afficher puis ensuite le bouton apparaître à l'intérieur. Il faut dire qu'avec des exemples simples comme celui−ci, vous ne ferez pas la différence.

```
gtk_widget_show(button);

gtk_widget_show (window);
```

Bien sûr, on appelle *gtk_main()* qui attendra les événements venant du serveur X et demandera aux widgets d'émettre les signaux lorsque ces événements surviendront.

```
gtk_main ();
```

Enfin, le *return* final. Il est exécuté lorsque *gtk_quit()* est appelé.

```
return 0;
```

Lorsque l'on clique sur un bouton GTK, le widget émet un signal "clicked". Afin de pouvoir utiliser cette information, notre programme configure un gestionnaire pour capturer ce signal. Ce gestionnaire appelle la fonction de notre choix. Dans notre exemple, lorsque le bouton que l'on a créé est "clicked", la fonction *hello()* est appelée avec le paramètre NULL, puis le gestionnaire suivant de ce signal est à son tour appelé. Il appelle la fonction *gtk_widget_destroy()* en lui passant le widget *window* comme paramètre, ce qui provoque la destruction de celui−ci. Ceci force la fenêtre à envoyer un signal "destroy", qui est capturé à son tour et appelle notre fonction de rappel *destroy()* qui ferme simplement GTK.

Une autre façon de procéder consiste à utiliser le gestionnaire de fenêtres pour détruire la fenêtre. Cela provoquera l'émission du signal "delete_event" qui sera pris en charge par notre gestionnaire *delete_event()*. S'il retourne FALSE, la fenêtre restera telle quelle et rien ne se passera. Retourner TRUE forcera GTK à émettre le signal "destroy" qui, bien sûr, appelera la fonction de rappel *destroy()* provoquant la sortie du GTK.

On remarquera que ces signaux ne sont pas les mêmes que les signaux systèmes Unix et ne sont pas implantés en utilisant ceux−ci, bien que la terminologie employée soit presque identique.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

3. Continuons

3.1 Types de données

Vous avez probablement noté certaines choses qui nécessitent des explications dans les exemples précédents. les *gint*, *gchar*, etc. que vous avez pu voir sont des redéfinitions de *int* et *char*, respectivement. Leur raison d'être est de s'affranchir des dépendances ennuyeuses concernant la taille des types de données simples lorsqu'on réalise des calculs. Un bon exemple est *gint32* qui désignera un entier codé sur 32 bits pour toutes les plateformes, que ce soit une station Alpha 64 bits ou un PC i386 32 bits. Les redéfinitions de type sont très simples et intuitives. Elles sont toutes décrites dans le fichier *glib/glib.h* (qui est inclus par *gtk.h*).

On notera aussi la possibilité d'utiliser un *GtkWidget* lorsque la fonction attend un *GtkObject*. GTK possède une architecture orientée objet, et un widget est un objet.

3.2 Compléments sur les gestionnaires de signaux

Regardons à nouveau la déclaration de *gtk_signal_connect*.

```
gint gtk_signal_connect (GtkObject *object, gchar *name,
                         GtkSignalFunc func, gpointer func_data);
```

Vous avez remarqué que le valeur de retour est de type *gint* ? Il s'agit d'un marqueur qui identifie votre fonction de rappel. Comme on le disait plus haut, on peut avoir autant de fonctions de rappel que l'on a besoin, par signal et par objet, et chacune sera exécutée à son tour, dans l'ordre dans lequel elle a été attachée. Ce marqueur vous permet d'ôter ce rappel de la liste en faisant &;:

```
void gtk_signal_disconnect (GtkObject *object, gint id);
```

Ainsi, en passant le widget dont on veut supprimer le gestionnaire et le marqueur ou identificateur retourné par l'une des fonctions *signal_connect*, on peut déconnecter un gestionnaire de signal.

Une autre fonction permettant de supprimer tous les gestionnaires de signaux pour un objet est :

```
gtk_signal_handlers_destroy (GtkObject *object);
```

Cet appel n'a pas trop besoin d'explications. Il ôte simplement tous les gestionnaires de signaux de l'objet passé en paramètre.

3.3 Un « Bonjour tout le monde » amélioré

Étudions une version légèrement améliorée avec de meilleurs exemples de fonctions de rappel. Ceci permettra aussi d'introduire le sujet suivant : le placement des wigdets.

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

/* Notre nouveau rappel amélioré. La donnée passée à cette fonction est
 * imprimée sur stdout. */

void rappel (GtkWidget *widget, gpointer *data)
{
    g_print ("Re−Bonjour − %s a été pressé\n", (char *) data);
}

/* Un autre rappel */

void delete_event (GtkWidget *widget, GdkEvent *event, gpointer *data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    /* GtkWidget est le type pour déclarer les widgets */

    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;

    /* Cette fonction est appelée dans toutes les applications GTK.
     * Les paramètre passés en ligne de commande sont analysés et
     * retournés à l'application. */

    gtk_init (&argc, &argv);

    /* Création d'une nouvelle fenêtre. */

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Nouvel appel qui intitule notre nouvelle fenêtre
     * "Salut les boutons !" */

    gtk_window_set_title (GTK_WINDOW (window), "Salut les boutons !");

    /* Configuration d'un gestionnaire pour "delete_event" afin de
     * quitter immédiatement GTK. */

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);
```

```
    /* Configuration de la largeur du contour de la fenêtre. */

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* Création d'une boîte pour y placer les widgets.
     * Ceci est décrit en détails plus loin dans la section
     * « placement ». La boîte n'est pas matérialisée, elle est juste
     * utilisée comme moyen d'arranger les widgets. */

    box1 = gtk_hbox_new(FALSE, 0);

    /* On met la boîte dans la fenêtre principale. */

    gtk_container_add (GTK_CONTAINER (window), box1);

    /* On crée un nouveau bouton portant le label « Bouton 1 ». */

    button = gtk_button_new_with_label ("Bouton 1");

    /* Lorsque le bouton est cliqué, on appelle la fonction « rappel »
     * avec un pointeur sur la chaîne « Bouton 1 » comme paramètre. */

    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (rappel), (gpointer) "Bouton 1");

    /* Au lieu d'utiliser gtk_container_add, on place ce bouton dans
     * la boîte invisible qui a été placée dans la fenêtre. */

    gtk_box_pack_start(GTK_BOX(box1), button, TRUE, TRUE, 0);

    /* N'oubliez jamais cette étape qui indique à GTK que la configuration
     * de ce bouton est terminée et qu'il peut être affiché. */

    gtk_widget_show(button);

    /* On fait la même chose pour créer un deuxième bouton. */

    button = gtk_button_new_with_label ("Bouton 2");

    /* On appelle la même fonction de rappel avec un paramètre différent,
     * un pointeur sur la chaîne « Bouton 2 ». */

    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (rappel), (gpointer) "Bouton 2");

    gtk_box_pack_start(GTK_BOX(box1), button, TRUE, TRUE, 0);

    /* L'ordre dans lequel on affiche les boutons n'est pas vraiment
     * important, mais il est préférable d'afficher la fenêtre en dernier
     * pour qu'elle surgisse d'un coup. */

    gtk_widget_show(button);

    gtk_widget_show(box1);

    gtk_widget_show (window);

    /* Le reste est dans gtk_main et on attend que la fête commence ! */

    gtk_main ();

    return 0;
}
```

Compilez ce programme en utilisant les mêmes paramètres que pour l'exemple précédent. Vous remarquerez que, maintenant, il est plus difficile de quitter le programme : vous devez utiliser le gestionnaire de fenêtres ou une commande shell pour le détruire. Un bon exercice pour le lecteur serait d'insérer un troisième bouton « Quitter » qui permettrait de sortir du programme. Vous pouvez aussi jouer avec les options de *gtk_box_pack_start()* en lisant la section suivante. Essayez de redimensionner la fenêtre, et observez son comportement.

Juste une remarque : il existe une autre constante utilisable avec *gtk_window_new()* − GTK_WINDOW_DIALOG. Ceci permet d'interagir de façon un peu différente avec le gestionnaire de fenêtres et doit être utilisé pour les fenêtres temporaires comme les boîtes de dialogue, par exemple.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

4. Placement des widgets

Lorsqu'on crée une application, on veut mettre plus qu'un simple bouton dans une fenêtre. Notre premier exemple « Bonjour le monde » n'utilisait qu'un seul widget et on pouvait donc simplement faire un appel à *gtk_container_add* pour « placer » le widget dans la fenêtre. Mais si l'on désire en mettre plus, comment peut−on contrôler l'endroit où le widget sera positionné ? C'est ici que le placement entre en jeu.

4.1 Théorie des boîtes de placement

La majeure partie du placement est faites en créant des boîtes comme dans l'exemple ci−dessus. Ce sont des widgets containers invisibles où l'on peut placer nos widgets. Elles existent sous deux formes : boîtes horizontales et boîtes verticales. Lorsque l'on place des widgets dans une boîte horizontale, les objets sont insérés horizontalement de gauche à droite ou de droite à gauche selon l'appel utilisé. Dans une boîte verticale, les widgets sont placés de haut en bas ou vice versa. On peut utiliser n'importe quelle combinaison de boîtes à l'intérieur ou à côté d'autres boîtes pour créer l'effet désiré.

Pour créer une nouvelle boîte horizontale, on appelle *gtk_hbox_new()*, et pour les boîtes verticales, *gtk_vbox_new()*. Les fonctions *gtk_box_pack_start()* et *gtk_box_pack_end()* servent à placer les objets à l'intérieur de ces containers. La fonction *gtk_box_pack_start()* placera de haut en bas dans une boîte verticale et de gauche à droite dans une boîte horizontale. *gtk_box_pack_end()* fera le contraire en plaçant de bas en haut et de droite à gauche. En utilisant ces fonctions, on peut aligner à droite ou à gauche nos widgets et même les mélanger de n'importe quelle façon pour obtenir l'effet désiré. Dans la plupart de nos exemples, on utilisera *gtk_box_pack_start()*. Un objet peut être un autre container ou un widget. En fait, de nombreux widgets (dont les boutons) sont eux−mêmes des containers, mais on utilise généralement seulement un label dans un bouton.

En utilisant ces appels, GTK sait où vous voulez placer vos widgets et il peut donc les dimensionner automatiquement et faire d'autres choses bien pratiques. Il existe aussi plusieurs options permettant de préciser comment les widgets doivent être placés. Comme vous pouvez l'imaginer, cette méthode nous donne pas mal de liberté pour placer et créer les widgets.

4.2 Détails sur les boîtes

À cause de cette liberté, le placement des boîtes avec GTK peut paraître déroutant au premier abord. Il existe beaucoup d'options et il n'est pas tout de suite évident de comprendre comment elles s'accordent toutes ensemble. En fait, il y a 5 styles de base différents.

Chaque ligne contient une boîte horizontale (*hbox*) contenant plusieurs boutons. L'appel à *gtk_box_pack* indique la façon dont sont placés tous les boutons dans la hbox. Chaque bouton est placé dans la hbox de la même façon (mêmes paramètres que la fonction *gtk_box_pack_start()*).

Voici la déclaration de la fonction *gtk_box_pack_start*.

```
void gtk_box_pack_start (GtkBox    *box,
                         GtkWidget *child,
                         gint       expand,
                         gint       fill,
                         gint       padding);
```

Le premier paramètre est la boîte dans laquelle on place l'objet, le second est cet objet. Tous les objets sont tous des boutons jusqu'à maintenant, on place donc des boutons dans des boîtes.

Le paramètre *expand* de *gtk_box_pack_start()* ou *gtk_box_pack_end()* contrôle la façon dont le widget est placé dans la boîte. S'il vaut TRUE, les widgets sont disposés dans la boîte de façon à en occuper tout l'espace. S'il vaut FALSE, la boîte est rétrécie pour correspondre à la taille du widget. Mettre *expand* à FALSE vous permettra d'aligner à droite et à gauche vos widgets. Sinon, ils s'élargiront pour occuper toute la boîte. Le même effet pourrait être obtenu en utilisant uniquement une des deux fonctions *gtk_box_pack_start* ou *pack_end*.

Le paramètre *fill* des fonctions *gtk_box_pack* contrôle si de l'espace supplémentaire doit être alloué aux objets eux−mêmes (TRUE), ou si on doit rajouter de l'espace (*padding*) dans la boîte autour des objets (FALSE). Il n'a de sens que si le paramètre *expand* vaut TRUE.

Lorsque l'on crée une nouvelle boîte, on utilise une fonction comme :

```
GtkWidget * gtk_hbox_new (gint homogeneous,
                          gint spacing);
```

Le paramètre *homogeneous* de *gtk_hbox_new* (et c'est la même chose pour *gtk_vbox_new*) vérifie que chaque objet de la boîte ait la même taille (i.e. la même largeur dans une hbox, la même hauteur dans une vbox). S'il vaut TRUE, le paramètre *expand* des fonctions *gtk_box_pack*

sera toujours mis à TRUE.

Quelle est alors la différence entre les paramètres *spacing* (configuré lorsque la boîte est créée) et *padding* (configuré lorque les éléments sont placés) ? *spacing* ajoute de l'espace entre les objets, et *padding* en ajoute de chaque côté d'un objet. La figure suivante devrait éclairer tout cela :

Voici le code utilisé pour créer les images ci−dessus. J'y ai mis beaucoup de commentaires en espérant que vous n'aurez pas de problème pour le relire. Compilez−le et jouez avec les différents paramètres.

4.3 Programme de démonstration des placements

```
#include "gtk/gtk.h"

void
delete_event (GtkWidget *widget, GdkEvent *event, gpointer *data)
{
    gtk_main_quit ();
}

/* Construction d'une nouvelle hbox remplie de boutons. Les paramètres qui
 * nous intéressent sont passés à cette fonction.
 * On n'affiche pas la boîte, mais tout ce qu'elle contient. */

GtkWidget *make_box (gint homogeneous, gint spacing,
                     gint expand, gint fill, gint padding)
{
    GtkWidget *box;
    GtkWidget *button;
    char padstr[80];

    /* Création d'une hbox avec les paramètres homogeneous et spacing
     * voulus. */

    box = gtk_hbox_new (homogeneous, spacing);

    /* Création d'une série de boutons configurés de façon appropriée */

    button = gtk_button_new_with_label ("gtk_box_pack");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("(box,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);
```

```
    button = gtk_button_new_with_label ("button,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    /* Création d'un bouton portant un label dépendant de la valeur
     * du paramètre expand. */

    if (expand == TRUE)
            button = gtk_button_new_with_label ("TRUE,");
    else
            button = gtk_button_new_with_label ("FALSE,");

    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    /* Même chose que ci-dessus mais sous forme abrégée. */

    button = gtk_button_new_with_label (fill ? "TRUE," : "FALSE,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    /* Récupération du paramètre padding sous forme de chaîne. */

    sprintf (padstr, "%d);", padding);

    button = gtk_button_new_with_label (padstr);
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    return box;
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;
    GtkWidget *box2;
    GtkWidget *separator;
    GtkWidget *label;
    GtkWidget *quitbox;
    int which;

    /* Initialisation, à ne jamais oublier ! :) */

    gtk_init (&argc, &argv);

    if (argc != 2) {
        fprintf (stderr, "usage : %s num, où num vaut 1, 2, ou 3.\n", *argv);

        /* Nettoyage dans GTK et sortie avec un code d'erreur de 1 */
        gtk_exit (1);
    }

    which = atoi (argv[1]);

    /* Création de notre fenêtre. */

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Il ne faut jamais oublier de connecter le signal "destroy" à la
     * fenêtre principale. C'est très important pour disposer d'un
     * comportement intuitif adéquat. */

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
```

```
                    GTK_SIGNAL_FUNC (delete_event), NULL);

  gtk_container_border_width (GTK_CONTAINER (window), 10);


  /* Création d'une boîte verticale (vbox) pour y placer les boîtes
   * horizontales.
   * Ceci permet de placer les boîtes horizontales contenant les boutons
   * les unes au dessus des autres dans cette vbox. */

  box1 = gtk_vbox_new (FALSE, 0);

  /* L'exemple à afficher. Ils correspondent aux images ci-dessus. */

  switch (which) {
  case 1:
      /* Création d'un label. */

      label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");

      /* Alignement du label à gauche.  On précisera cette fonction ainsi
       * que les autres dans la section sur les attributs des widgets. */

      gtk_misc_set_alignment (GTK_MISC (label), 0, 0);

      /* Placement du label dans la boîte verticale (vbox box1). Il ne
       * faut pas oublier que les widgets qui s'ajoutent à une vbox sont
       * placés les uns au dessus des autres. */

      gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);

      /* Affichage du label */

      gtk_widget_show (label);

      /* On appelle notre fonction de construction de boîte :
       * homogeneous = FALSE, spacing = 0,
       * expand = FALSE, fill = FALSE, padding = 0 */

      box2 = make_box (FALSE, 0, FALSE, FALSE, 0);
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      /* On appelle notre fonction de construction de boîte :
       * homogeneous = FALSE, spacing = 0,
       * expand = FALSE, fill = FALSE, padding = 0 */

      box2 = make_box (FALSE, 0, TRUE, FALSE, 0);
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      /* Paramètres : homogeneous = FALSE, spacing = 0,
       * expand = TRUE, fill = TRUE, padding = 0 */

      box2 = make_box (FALSE, 0, TRUE, TRUE, 0);
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      /* Création d'un séparateur, on verra cela plus tard, mais ils sont
       * simples à utiliser. */

      separator = gtk_hseparator_new ();

      /* Placement du séparateur dans la vbox. Ne pas oublier que tous les
       * widgets sont placés dans une vbox et qu'il seront placés
```

```
             * verticalement. */

            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);

            /* Création d'un nouveau label et affichage de celui-ci. */

            label = gtk_label_new ("gtk_hbox_new (TRUE, 0);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Paramètres : homogeneous = TRUE, spacing = 0,
             * expand = TRUE, fill = FALSE, padding = 0 */

            box2 = make_box (TRUE, 0, TRUE, FALSE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Paramètres : homogeneous = TRUE, spacing = 0,
             * expand = TRUE, fill = TRUE, padding = 0 */

            box2 = make_box (TRUE, 0, TRUE, TRUE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Un autre séparateur */

            separator = gtk_hseparator_new ();

            /* Les 3 derniers paramètres de gtk_box_pack_start sont :
             * expand = FALSE, fill = TRUE, padding = 5. */

            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);

            break;

        case 2:

            /* Création d'un label, box1 est une vbox identique à
             * celle créée au début de main() */

            label = gtk_label_new ("gtk_hbox_new (FALSE, 10);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Paramètres : homogeneous = FALSE, spacing = 10,
             * expand = TRUE, fill = FALSE, padding = 0 */

            box2 = make_box (FALSE, 10, TRUE, FALSE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Paramètres : homogeneous = FALSE, spacing = 10,
             * expand = TRUE, fill = TRUE, padding = 0 */

            box2 = make_box (FALSE, 10, TRUE, TRUE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            separator = gtk_hseparator_new ();

            /* Les 3 derniers paramètres de gtk_box_pack_start sont :
```

```
             * expand = FALSE, fill = TRUE, padding = 5. */

            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);

            label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Paramètres : homogeneous = FALSE, spacing = 0,
             * expand = TRUE, fill = FALSE, padding = 10 */

            box2 = make_box (FALSE, 0, TRUE, FALSE, 10);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Paramètres : homogeneous = FALSE, spacing = 0,
             * expand = TRUE, fill = TRUE, padding = 10 */

            box2 = make_box (FALSE, 0, TRUE, TRUE, 10);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            separator = gtk_hseparator_new ();

            /* Les 3 derniers paramètres de gtk_box_pack_start sont :
             * expand = FALSE, fill = TRUE, padding = 5. */

            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);
            break;

        case 3:

            /* Ceci est une démonstration de la possibilité d'utiliser
             * gtk_box_pack_end() pour aligner les widgets à droite.
             * On crée d'abord une nouvelle boîte comme d'habitude. */

            box2 = make_box (FALSE, 0, FALSE, FALSE, 0);

            /* On crée le label qui sera mis à la fin. */

            label = gtk_label_new ("end");

            /* On le place en utilisant gtk_box_pack_end(), il est ainsi
             * mis à droite de la hbox créée par l'appel à make_box(). */

            gtk_box_pack_end (GTK_BOX (box2), label, FALSE, FALSE, 0);

            /* Affichage du label. */

            gtk_widget_show (label);

            /* Placement de box2 dans box1 (la vbox, vous vous rappelez ? :) */

            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Séparateur pour le bas. */

            separator = gtk_hseparator_new ();

            /* Configuration du séparateur en 400x5 pixels.
             * La hbox que l'on a créée aura donc 400 pixels de large,
```

```
             * et le label "end" sera séparé des autres de la hbox.
             * Sinon, tous les widgets de la hbox seraient placés les plus
             * près possible les uns des autres. */

            gtk_widget_set_usize (separator, 400, 5);

            /* Placement du séparateur dans la vbox (box1)
             * créée au debut de main(). */

            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);
        }

        /* Création d'une nouvelle hbox.. vous pouvez en utiliser autant que
         * que vous en avez besoin ! */

        quitbox = gtk_hbox_new (FALSE, 0);

        /* Notre bouton pour quitter. */

        button = gtk_button_new_with_label ("Quit");

        /* Configuration du signal pour détruire la fenêtre. Ceci enverra le
         * signal "destroy" à la fenêtre. Ce signal sera à son tour capturé
         * par notre gestionnaire de signal défini plus haut. */

        gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                                   GTK_SIGNAL_FUNC (gtk_widget_destroy),
                                   GTK_OBJECT (window));

        /* Placement du bouton dans la « quitbox ».
         * Les 3 derniers paramètres de gtk_box_pack_start sont :
         * expand = TRUE, fill = FALSE, padding = 0. */

        gtk_box_pack_start (GTK_BOX (quitbox), button, TRUE, FALSE, 0);

        /* Placement de la quitbox dans la vbox (box1) */

        gtk_box_pack_start (GTK_BOX (box1), quitbox, FALSE, FALSE, 0);

        /* Placement de la vbox (box1), qui contient maintenant tous nos
         * widgets, dans la fenêtre principale. */

        gtk_container_add (GTK_CONTAINER (window), box1);

        /* Affichage */

        gtk_widget_show (button);
        gtk_widget_show (quitbox);

        gtk_widget_show (box1);

        /* Affichage de la fenêtre en dernier */

        gtk_widget_show (window);

        /* Ne pas oublier notre fonction principale. */

        gtk_main ();

        /* Le contrôle revient ici lorsque gtk_main_quit() est appelée,
         * jusqu'à ce que  gtk_exit() soitutilisée. */

        return 0;
    }
```

4.4 Placement avec les tables

Étudions une autre méthode de placement : les tables. Elles peuvent s'avérer très utiles dans certaines situations.

En utilisant des tables, on crée une grille dans laquelle on peut placer les widgets. Ceux−ci peuvent occuper tous les endroits que l'on désire.

La première chose à faire est, bien sûr, d'étudier la fonction *gtk_table_new* :

```
GtkWidget* gtk_table_new (gint rows,
                          gint columns,
                          gint homogeneous);
```

Le premier paramètre est le nombre de lignes de la table et le deuxième, le nombre de colonnes.

Le paramètre *homogeneous* s'occupe de la façon dont les cases de la table seront dimensionnées. Si homogeneous vaut TRUE, les cases prennent la taille du plus grand widget de la table. S'il vaut FALSE, la taille des cases dépend du widget le plus haut de la ligne et du plus large de cette colonne.

Le nombre de lignes et colonnes va de 0 à n, où n est le nombre spécifié dans l'appel à *gtk_table_new*. Ainsi, avec *rows* = 2 et *columns* = 2, la table ressemblera à ceci :

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

On notera que le système de coordonnées part du coin en haut à gauche. Pour placer un widget dans une case, ou utilise la fonction suivante :

```
void gtk_table_attach (GtkTable      *table,
                       GtkWidget     *child,
                       gint           left_attach,
                       gint           right_attach,
                       gint           top_attach,
                       gint           bottom_attach,
                       gint           xoptions,
                       gint           yoptions,
                       gint           xpadding,
                       gint           ypadding);
```

Où le premier paramètre (*table*) est la table que l'on a créée et le second (*child*) est le widget que l'on veut placer dans la table.

Les paramètres *left_attach* et *right_attach* spécifient l'emplacement du widget et le nombre de cases à utiliser. Par exemple, si on veut placer un bouton dans le coin inférieur droit de la table décrite plus haut et que l'on désire ne remplir QUE cette case, *left_attach* vaudra 1, *right_attach* vaudra 2; *top_attach* vaudra 1 et *bottom_attach* vaudra 2.

Si on veut un widget occupant toute la ligne supérieure de notre table, on utilisera les valeurs 0, 2, 0, 1.

Les paramètres *xoptions* et *yoptions* servent à préciser les options de placement et peuvent être combinées par un OU logique pour permettre des options multiples.

Ces options sont :

- GTK_FILL − Si la case de la table est plus large que le widget, et que GTK_FILL est spécifié, le widget s'élargira pour occuper toute la place disponible.
- GTK_SHRINK − Si la table a moins de place qu'il ne lui en faut (généralement, à cause d'un redimensionnement de la fenêtre par l'utilisateur), les widgets sont alors simplement poussés vers le bas de la fenêtre et disparaissent. Si GTK_SHRINK est spécifié, les widgets se réduiront en même temps que la table.
- GTK_EXPAND − Cette option provoque l'extension de la table pour qu'elle utilise tout l'espace restant dans la fenêtre.

Le paramêtres de *padding* jouent le même rôle que pour les boîtes, il créent une zone libre, spécifiée en pixels, autour du widget.

gtk_table_attach() a BEAUCOUP d'options. Voici donc une fonction−raccourci :

```
void gtk_table_attach_defaults (GtkTable   *table,
                                GtkWidget  *widget,
                                gint        left_attach,
                                gint        right_attach,
                                gint        top_attach,
                                gint        bottom_attach);
```

xoptions et *options* valent par défaut GTK_FILL | GTK_EXPAND, et *xpadding* et *ypadding* valent 0. Les autres paramètres sont les mêmes que ceux de la fonction précédente.

Il existe aussi les fonctions *gtk_table_set_row_spacing()* et *gtk_table_set_col_spacing()*. Elles permettent de placer des espaces après une ligne ou une colonne.

```
void gtk_table_set_row_spacing (GtkTable      *table,
                                gint           row,
                                gint           spacing);
```

et

```
void gtk_table_set_col_spacing  (GtkTable      *table,
                                 gint           column,
                                 gint           spacing);
```

Pour les colonnes, l'espace est ajouté à droite de la colonne et pour les lignes, il est ajouté en dessous.

On peut aussi configurer un espacement pour toutes les lignes et/ou colonnes avec :

```
void gtk_table_set_row_spacings (GtkTable *table,
                                 gint      spacing);
```

Et,

```
void gtk_table_set_col_spacings (GtkTable  *table,
                                 gint       spacing);
```

Avec ces appels, la dernière ligne et la dernière colonne n'ont pas d'espace supplémentaire.

4.5 Exemple de placement avec table

Pour le moment, étudiez l'exemple sur les tables (testgtk.c) distribué avec les sources de GTK.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

5. Vue d'ensemble des widgets

Les étapes pour créer un widget en GTK sont :

1. *gtk_*_new()* − une des fonctions disponibles pour créer un nouveau widget. Ces fonctions sont décrites dans cette section.
2. Connexion de tous les signaux que l'on souhaite utiliser avec les gestionnaires adéquats.
3. Configuration des attributs du widget.
4. Placement du widget dans un container en utilisant un appel approprié comme *gtk_container_add()* ou *gtk_box_pack_start()*.
5. Affichage du widget grâce à *gtk_widget_show()*.

gtk_widget_show() permet à GTK de savoir que l'on a fini de configurer les attributs du widget et qu'il est prêt à être affiché. On peut aussi utiliser *gtk_widget_hide()* pour le faire disparaître. L'ordre dans lequel on affiche les widgets n'est pas important, mais il est préférable d'afficher la fenêtre en dernier pour qu'elle surgisse d'un seul coup plutôt que de voir les différents widgets apparaître à l'écran au fur et à mesure. Les fils d'un widget (une fenêtre est aussi un widget) ne seront pas affichés tant que la fenêtre elle−même n'est pas affichée par la fonction *gtk_widget_show()*.

5.1 Conversions de types

Vous remarquerez, au fur et à mesure que vous progressez, que GTK utilise un système de coercition de type. Celle−ci est toujours réalisée en utilisant des macros qui vérifient si l'objet donné peut être converti et qui réalisent cette coercition. Les macros que vous rencontrerez le plus sont :

- GTK_WIDGET(widget)
- GTK_OBJECT(object)
- GTK_SIGNAL_FUNC(function)
- GTK_CONTAINER(container)
- GTK_WINDOW(window)
- GTK_BOX(box)

Elles sont toutes utilisées pour convertir les paramètres des fonctions. Vous les verrez dans les exemples et, en règle générale, vous saurez les utiliser simplement en regardant la déclaration d'une fonction.

Comme vous pouvez le voir dans la hiérarchie de classes ci−dessous, tous les *GtkWidgets* dérivent d'une classe de base *GtkObject*. Ceci signifie que vous pouvez utiliser un widget à chaque fois qu'une fonction requiert un objet − il suffit d'utiliser la macro GTK_OBJECT().

Par exemple :

```
gtk_signal_connect(GTK_OBJECT(button), "clicked",
                   GTK_SIGNAL_FUNC(fonction_rappel), donnee_de_rappel);
```

Cet appel convertit le bouton en objet et fournit une conversion pour le pointeur de fonction vers la fonction de rappel.

De nombreux widgets sont aussi des containers. Si vous regardez la hiérarchie de classe ci−dessous, vous remarquerez que beaucoup de widgets viennent de la classe *GtkContainer*. N'importe lequel de ces widgets peut être utilisé avec la macro GTK_CONTAINER pour être passé en paramètre à une fonction qui attend un container.

Malheureusement, ces macros ne peuvent être couvertes en détail dans ce didacticiel, Je vous recommande donc de jeter un coup d'oeil sur les fichier en−têtes GTK : ils peuvent s'avérer très instructifs. En fait, il n'est pas difficile de comprendre comment fonctionne un widget, il suffit d'étudier les déclarations des fonctions.

5.2 La hiérarchie des widgets

Voici l'arbre de la hiérarchie de classes utilisées pour implanter les widgets.

```
GtkObject
+−− GtkData
|   \−− GtkAdjustment
|
\−− GtkWidget
    +−− GtkContainer
    |   +−− GtkBin
    |   |   +−− GtkAlignment
    |   |   +−− GtkFrame
    |   |   |   *−− GtkAspectFrame
    |   |   |
    |   |   +−− GtkItem
    |   |   |   +−− GtkListItem
    |   |   |   +−− GtkMenuItem
    |   |   |   |   +−− GtkCheckMenuItem
    |   |   |   |       *−− GtkRadioMenuItem
    |   |   |   |
    |   |   |   *−− GtkTreeItem
    |   |   |
    |   |   +−− GtkViewport
    |   |   \−− GtkWindow
    |   |       +−− GtkDialog
    |   |       \−− GtkFileSelection
    |   |
    |   +−− GtkBox
    |   |   +−− GtkHBox
    |   |   \−− GtkVBox
    |   |       +−− GtkColorSelection
    |   |       \−− GtkCurve
    |   |
    |   +−− GtkButton
    |   |   +−− GtkOptionMenu
    |   |   \−− GtkToggleButton
    |   |       \−− GtkCheckButton
    |   |           \−− GtkRadioButton
    |   |
    |   +−− GtkList
    |   +−− GtkMenuShell
    |   |   +−− GtkMenu
    |   |   \−− GtkMenuBar
    |   |
```

```
|   +-- GtkNotebook
|   +-- GtkScrolledWindow
|   +-- GtkTable
|   \-- GtkTree
|
+-- GtkDrawingArea
+-- GtkEntry
+-- GtkMisc
|   +-- GtkArrow
|   +-- GtkImage
|   +-- GtkLabel
|   \-- GtkPixmap
|
+-- GtkPreview
+-- GtkProgressBar
+-- GtkRange
|   +-- GtkScale
|   |   +-- GtkHScale
|   |   \-- GtkVScale
|   |
|   \-- GtkScrollbar
|       +-- GtkHScrollbar
|       \-- GtkVScrollbar
|
+-- GtkRuler
|   +-- GtkHRuler
|   \-- GtkVRuler
|
\-- GtkSeparator
    +-- GtkHSeparator
    \-- GtkVSeparator
```

5.3 Widgets sans fenêtre

Les widgets suivants n'ont pas de fenêtre associée. Si vous voulez capturer des événements, vous devez utiliser *GtkEventBox*. Reportez−vous à la section sur Le widget EventBox

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPaned
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkViewport
GtkAspectFrame
GtkFrame
GtkVPaned
GtkHPaned
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

Nous continuerons notre exploration de GTK en examinant chaque widget tour à tour, créant quelques fonctions simples pour les afficher. Une autre source intéressante est le programme *testgtk.c* livré avec GTK. Il se trouve dans le répertoire *gtk/*

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

6. Widgets boutons

6.1 Boutons normaux

On a déjà presque vu tout ce qu'il y avait à voir sur le widget bouton. Il est très simple. Cependant, il y a deux façons de créer un bouton. On peut utiliser *gtk_button_new_with_label()* pour créer un bouton avec un label, ou *gtk_button_new()* pour créer un bouton vide. Dans ce dernier cas, c'est à vous de placer un label ou un pixmap sur celui−ci. Pour ce faire, créez une boîte, puis placez vos objets dans celle−ci en utilisant la fonction habituelle *gtk_box_pack_start*, utilisez alors *gtk_container_add* pour placer la boîte dans le bouton.

Voici un exemple d'utilisation de *gtk_button_new()* pour créer un bouton contenant une image et un label. J'ai séparé du reste le code qui crée une boîte pour que vous puissiez l'utiliser dans vos programmes.

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


/* Création d'une hbox avec une image et un label. Cette fonction
 * retourne la boîte... */

GtkWidget *xpm_label_box (GtkWidget *parent, gchar *xpm_filename,
                          gchar *label_text)
{
    GtkWidget *box1;
    GtkWidget *label;
    GtkWidget *pixmapwid;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    GtkStyle *style;

    /* Création de la boite pour un xpm et un label */

    box1 = gtk_hbox_new (FALSE, 0);
    gtk_container_border_width (GTK_CONTAINER (box1), 2);

    /* Choix d'un style de bouton... Je suppose que c'est pour obtenir
     * la couleur du fond. Si quelqu'un connaît la vraie raison, qu'il
     * m'éclaire sur ce point. */

    style = gtk_widget_get_style(parent);

    /* Chargement de xpm pour créer une image */

    pixmap = gdk_pixmap_create_from_xpm (parent->window, &mask,
                                         &style->bg[GTK_STATE_NORMAL],
                                         xpm_filename);
    pixmapwid = gtk_pixmap_new (pixmap, mask);

    /* Création d'un label  */

    label = gtk_label_new (label_text);
```

```
    /* placement de l'image et du label dans la boîte */

    gtk_box_pack_start (GTK_BOX (box1),
                        pixmapwid, FALSE, FALSE, 3);

    gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 3);

    gtk_widget_show(pixmapwid);
    gtk_widget_show(label);

    return (box1);
}

/* Notre fonction de rappel habituelle */

void callback (GtkWidget *widget, gpointer *data)
{
    g_print ("Bonjour − %s a été pressé\n", (char *) data);
}


int main (int argc, char *argv[])
{
    /* GtkWidget est le type utilisé pour déclarer les widgets */

    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;

    gtk_init (&argc, &argv);

    /* Création d'une fenêtre */

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Pixmap'd Buttons!");

    /* Il est préférable de faire cela pour toutes les fenêtres */

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_exit), NULL);


    /* Configuration du bord de la fenêtre */

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* Création d'un bouton */

    button = gtk_button_new ();

    /* Vous devriez être habitué à voir ces fonctions maintenant */

    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (callback), (gpointer) "cool button");

    /* Appel de notre fonction de création de boîte */

    box1 = xpm_label_box(window, "info.xpm", "cool button");

    /* Placement et affichage de tous nos widgets */

    gtk_widget_show(box1);
```

```
    gtk_container_add (GTK_CONTAINER (button), box1);

    gtk_widget_show(button);

    gtk_container_add (GTK_CONTAINER (window), button);

    gtk_widget_show (window);

    /* Le reste est dans gtk_main */
    gtk_main ();

    return 0;
}
```

La fonction *xpm_label_box()* peut être utilisée pour placer des xpms et des labels sur tout widget qui peut être container.

6.2 Boutons commutateurs

Les boutons commutateurs ressemblent beaucoup aux boutons normaux, sauf qu'ils seront toujours alternativement dans un état ou dans un autre. Le changement d'état s'effectue par un click. Ils peuvent être enfoncés et, lorsqu'on clique dessus, ils se relèvent. Re−cliquez, et ils se renfoncent.

Les boutons commutateurs sont la base des cases à cocher ou des boutons radio, donc la plupart des appels utilisés pour les boutons commutateurs sont hérités par les cases à cocher et les boutons radio. J'insisterai là dessus quand nous les aborderons.

Création d'un bouton commutateur :

```
GtkWidget* gtk_toggle_button_new (void);

GtkWidget* gtk_toggle_button_new_with_label (gchar *label);
```

Comme vous pouvez l'imaginer, elles fonctionnent comme celles des boutons normaux. La première crée un bouton commutateur vide et la deuxième un bouton commutateur contenant déjà un label.

Pour récupérer l'état d'un commutateur et cela comprend aussi les cases à cocher et les boutons radio, on utilise une macro comme nous le montrons dans l'exemple qui suit et qui teste l'état du commutateur dans une fonction de rappel. Le signal qui nous intéresse et qui est émis par les boutons commutateurs (ce qui comprend aussi les cases à cocher et les boutons radio), est le signal "toggled". Pour vérifier l'état de ces boutons, on configure un gestionnaire de signal qui capture "toggled" et utilise la macro pour déterminer l'état. La fonction de rappel ressemblera à ceci :

```
void rappel_bouton_commutateur (GtkWidget *widget, gpointer data)
{
    if (GTK_TOGGLE_BUTTON(widget)−>active)
    {
        /* Si l'on est ici, c'est que le bouton est relâché. */

    } else {

        /* le bouton est enfoncé */
    }
}
```

L'appel qui suit peut être utilisé pour configurer l'état d'un bouton commutateur et de ses descendants, les cases à cocher et les boutons radio. On lui passe notre bouton en premier paramètre et TRUE ou FALSE pour spécifier s'il doit être relâché ou enfoncé. Par défaut, il est relâché (FALSE).

```
void gtk_toggle_button_set_state (GtkToggleButton *toggle_button,
                                  gint state);
```

On notera que lorsqu'on utilise cette fonction, et que l'état est modifié, cela force le bouton à émettre un signal "clicked".

```
void gtk_toggle_button_toggled (GtkToggleButton *toggle_button);
```

Cet appel ne fait que commuter le bouton et émettre le signal "toggled".

6.3 Cases à cocher

Les cases à cocher héritent de nombreuses propriétés et fonctions des boutons commutateurs, mais ont un aspect différent. Au lieu d'être des boutons contenant du texte, ce sont de petits carrés avec un texte sur leur droite. Il sont souvent utilisés pour valider ou non des options dans les applications.

Les deux fonctions de création sont identiques à celles des boutons normaux.

```
GtkWidget* gtk_check_button_new (void);

GtkWidget* gtk_check_button_new_with_label (gchar *label);
```

La fonction *new_with_label* crée une case à cocher avec un texte à coté d'elle.

La vérification de l'état d'une case à cocher est identique à celle des boutons commutateurs.

6.4 Boutons radio

Les boutons radio ressemblent aux cases à cocher sauf qu'ils sont groupés de façon à ce qu'un seul d'entre−eux puisse être sélectionné à un moment donné. Ils sont utilisés par les applications lorsqu'il s'agit d'effectuer un choix dans une liste d'options.

La création d'un bouton radio s'effectue grâce à l'un des appels suivants :

```
GtkWidget* gtk_radio_button_new (GSList *group);

GtkWidget* gtk_radio_button_new_with_label (GSList *group,
                                            gchar *label);
```

On notera le paramètre supplémentaire de ces fonctions. Elles nécessitent un groupe pour réaliser correctement leur tâche. Le premier appel doit passer NULL au premier paramètre puis on peut créer un groupe en utilisant :

```
GSList* gtk_radio_button_group (GtkRadioButton *radio_button);
```

On passe alors ce groupe en premier paramètre des appels suivants aux fonctions de création. Il est préférable, aussi, de préciser quel bouton doit être choisi par défaut avec la fonction :

```
void gtk_toggle_button_set_state (GtkToggleButton *toggle_button,
                                  gint state);
```

Celle−ci est décrite dans la section sur les boutons commutateurs et fonctionne exactement de la même façon.

[Mettre ici un exemple d'utilisation de tout cela car je crois que cela ferait beaucoup de bien...]

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

7. Widgets divers

7.1 Labels

Les labels sont très utilisés dans GTK et sont relativement simples. Ils n'émettent pas de signaux car ils n'ont pas de fenêtre X qui leur est associée. Si vous avez besoin de capturer des signaux ou de faire des coupures (« clippings »), utilisez un widget EventBox.

Pour créer un label, on utilise :

```
GtkWidget* gtk_label_new (char *str);
```

Où l'unique paramètre est la chaîne de caractères que l'on veut que le label affiche.

Pour changer le texte d'un label après sa création, on utilise la fonction :

```
void gtk_label_set (GtkLabel  *label,
                    char      *str);
```

où le premier paramètre est le label que l'on veut modifier, que l'on convertit en utilisant la macro GTK_LABEL(), et le second est la nouvelle chaîne.

L'espace nécessaire à la nouvelle chaîne sera automatiquement ajusté si nécessaire.

Pour récupérer la chaîne courante, on utilise la fonction :

```
void gtk_label_get (GtkLabel  *label,
                    char     **str);
```

où le premier paramètre est le label dont on veut récupérer la chaîne et le second sert à retourner cette chaîne.

7.2 Le widget bulle d'aide

Ce sont les petits textes qui surgissent lorsque vous laissez votre pointeur sur un bouton ou un autre widget pendant quelques secondes. Ils sont faciles à utiliser, on ne donnera donc pas d'exemple. Si vous voulez voir du code, consultez le programme *testgtk.c* distribué avec GTK.

Certains widgets (comme les labels) ne fonctionnent pas avec les bulles d'aide.

Le premier appel que vous utiliserez sera pour créer une nouvelle bulle d'aide. Vous n'avez besoin que de le faire une fois dans une fonction donnée. Le *GtkTooltip* que cette fonction

retourne peut être utilisé pour créer plusieurs bulles d'aide.

```
GtkTooltips *gtk_tooltips_new (void);
```

Lorsque vous avez créé une nouvelle bulle d'aide et le widget sur lequel vous voulez l'utiliser, vous n'avez qu'à faire cet appel pour la configurer :

```
void gtk_tooltips_set_tips   (GtkTooltips *tooltips,
                              GtkWidget   *widget,
                              gchar       *tips_text);
```

Les paramètres sont la bulle d'aide déjà créée, suivi du widget pour lequel vous voulez voir apparaître cette bulle et le texte que vous voulez qu'elle contienne.

Voici un petit exemple :

```
GtkTooltips *tooltips;
GtkWidget *button;
...
tooltips = gtk_tooltips_new ();
button = gtk_button_new_with_label ("bouton 1");
...
gtk_tooltips_set_tips (tooltips, button, "C'est le bouton 1");
```

D'autres fonctions peuvent être utilisées avec les bulles d'aide. Je ne ferais que les énumérer et les décrire brièvement.

```
void gtk_tooltips_destroy    (GtkTooltips *tooltips);
```

Destruction de bulles d'aide.

```
void gtk_tooltips_enable     (GtkTooltips *tooltips);
```

Activation d'un ensemble de bulles d'aide désactivées.

```
void gtk_tooltips_disable    (GtkTooltips *tooltips);
```

Désactivation d'un ensemble de bulles d'aide activées.

```
void gtk_tooltips_set_delay  (GtkTooltips *tooltips,
                              gint         delay);
```

Configure le nombre de millisecondes pendant lequel le pointeur soit se trouver sur le widget avant que la bulle d'aide n'apparaisse. Par défaut, ce délai est de 1000 millisecondes, soit 1 seconde.

```
void      gtk_tooltips_set_tips (GtkTooltips *tooltips,
                                 GtkWidget   *widget,
                                 gchar    *tips_text);
```

Change le texte d'une bulle d'aide déjà créée.

```
void gtk_tooltips_set_colors (GtkTooltips *tooltips,
                              GdkColor    *background,
                              GdkColor    *foreground);
```

Configure les couleurs de fond et de premier plan des bulles d'aides. Je ne sais toujours pas comment spécifier les couleurs...

Et c'est tout concernant les fonctions associées aux bulles d'aide. C'est plus que vous ne vouliez sûrement en savoir :)

7.3 Barres de progression

Les barres de progression sont utilisées pour afficher la progression d'une opération. Elles sont très simple à utiliser comme vous pourrez le constater en étudiant le code ci−dessous. Commençons d'abord par l'appel permettant de créer une nouvelle barre.

```
GtkWidget *gtk_progress_bar_new (void);
```

Maintenant que la barre est créée, nous pouvons l'utiliser.

```
void gtk_progress_bar_update (GtkProgressBar *pbar, gfloat percentage);
```

Le premier paramètre est la barre de progression sur laquelle on veut agir, et le second est le pourcentage « effectué », signifiant le remplissage de la barres de 0 à 100 % (réel compris entre 0 et 1).

Les barres de progression sont généralement utilisées avec les délais d'expiration ou autres fonctions identiques (voir la section sur Expirations, fonctions d'E/S et d'attente) pour donner l'illusion du multi−tâches. Toutes emploient la fonction *gtk_progress_bar_update* de la même façon.

Voici un exemple de barre de progression mise à jour par des expirations. Ce code montre aussi comment réinitialiser une barre.

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

static int ptimer = 0;
int pstat = TRUE;

/* Cette fonction incrémente et met à jour la barre de progression,
 * elle la réinitialise si pstat vaut FALSE */

gint progress (gpointer data)
{
    gfloat pvalue;

    /* récupération de la valeur courante de la barre */

    pvalue = GTK_PROGRESS_BAR (data)->percentage;

    if ((pvalue >= 1.0) || (pstat == FALSE)) {
        pvalue = 0.0;
        pstat = TRUE;
    }
    pvalue += 0.01;

    gtk_progress_bar_update (GTK_PROGRESS_BAR (data), pvalue);

    return TRUE;
}

/* Cette fonction signale une réinitialisation de la barre */

void progress_r (void)
{
    pstat = FALSE;
}
```

```
void destroy (GtkWidget *widget, gpointer *data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *label;
    GtkWidget *table;
    GtkWidget *pbar;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (destroy), NULL);

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new(3,2,TRUE);
    gtk_container_add (GTK_CONTAINER (window), table);

    label = gtk_label_new ("Exemple de barre de progression");
    gtk_table_attach_defaults(GTK_TABLE(table), label, 0,2,0,1);
    gtk_widget_show(label);

    /* Crée une barre, la place dans la table et l'affiche */

    pbar = gtk_progress_bar_new ();
    gtk_table_attach_defaults(GTK_TABLE(table), pbar, 0,2,1,2);
    gtk_widget_show (pbar);

    /* Configure le délai d'expiration pour gérer automatiquement la
     *  mise à jour de la barre */

    ptimer = gtk_timeout_add (100, progress, pbar);

    /* Ce bouton indique à la barre qu'elle doit se réinitialiser */

    button = gtk_button_new_with_label ("Reset");
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (progress_r), NULL);
    gtk_table_attach_defaults(GTK_TABLE(table), button, 0,1,2,3);
    gtk_widget_show(button);

    button = gtk_button_new_with_label ("Annuler");
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (destroy), NULL);

    gtk_table_attach_defaults(GTK_TABLE(table), button, 1,2,2,3);
    gtk_widget_show (button);

    gtk_widget_show(table);
    gtk_widget_show(window);

    gtk_main ();

    return 0;
}
```

Dans ce petit programme, il y a quatre parties concernant le fonctionnement général des barres de progression, nous les étudierons dans l'ordre de leurs appels.

```
pbar = gtk_progress_bar_new ();
```

Cet appel crée une nouvelle barre, nommée *pbar*.

```
ptimer = gtk_timeout_add (100, progress, pbar);
```

Cet appel utilise des délais d'expiration pour permettre un intervalle de temps constant. ces délais ne sont pas nécessaires à l'utilisation des barres de progression.

```
pvalue = GTK_PROGRESS_BAR (data)->percentage;
```

Ce code assigne à *pvalue* la valeur du pourcentage de la barre.

```
gtk_progress_bar_update (GTK_PROGRESS_BAR (data), pvalue);
```

Finalement, ce code met à jour la barre avec la valeur de *pvalue*.

Et c'est tout ce qu'il y a à savoir sur les barres de progression. Amusez-vous bien.

7.4 Boîtes de dialogue

Les widgets boîtes de dialogue sont très simples : ce sont simplement des fenêtres avec plusieurs choses déjà placées dedans. La structure d'une boîte de dialogue est :

```
struct GtkDialog
{
      GtkWindow window;

      GtkWidget *vbox;
      GtkWidget *action_area;
};
```

Comme vous le voyez, cela crée simplement une fenêtre et la place dans une vbox suivie d'un séparateur et d'une hbox pour la « zone d'action ».

Le widget boîte de dialogue peut servir à produire des messages pour l'utilisateur ainsi qu'à d'autres tâches. Il est vraiment rudimentaire et il n'y a qu'une seule fonction pour les boîtes de dialogue :

```
GtkWidget* gtk_dialog_new (void);
```

Ainsi, pour créer un nouveau dialogue, on utilise :

```
GtkWidget window;
window = gtk_dialog_new ();
```

Ceci créera la boîte de dialogue et c'est maintenant à vous de l'utiliser. Vous pouvez, par exemple, placer un bouton dans la zone d'action en faisant quelque chose comme :

```
button = ...
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button,
                    TRUE, TRUE, 0);
gtk_widget_show (button);
```

Et vous pouvez aussi ajouter un label à la zone de la vboxb :

```
label = gtk_label_new ("Les boîtes de dialogues sont pratiques");
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->vbox), label, TRUE,
                    TRUE, 0);
gtk_widget_show (label);
```

Comme exemple d'utilisation d'une boîte de dialogue, vous pourriez mettre deux boutons dans la zone d'action (un bouton « Annuler » et un bouton « Ok ») et un label dans la zone de la vbox posant une question à l'utilisateur ou signalant une erreur, etc. Vous pouvez alors attacher un signal différent à chacun des boutons et réaliser l'opération que l'utilisateur a choisie.

7.5 Pixmaps

Les pixmaps sont des structures de données contenant des images. Celles-ci peuvent être utilisées à différents endroits, mais le plus souvent comme icônes dans le bureau X Window. Un bitmap est un pixmap de 2 couleurs.

Pour utiliser des pixmaps avec GTK, on doit d'abord construire une structure *GdkPixmap* en utilisant les fonctions de la couche GDK. Les pixmaps peuvent soit être créés à partir de données en memoire, ou à partir de données lues dans un fichier. Nous utiliserons chacun des appels pour créer un pixmap.

```
GdkPixmap *gdk_bitmap_create_from_data( GdkWindow *window,
                                        gchar     *data,
                                        gint      width,
                                        gint      height );
```

Cette fonction sert à créer un pixmap mono-plan (2 couleurs) à partir de données en mémoire. Chaque bit de la donnée *data*. *width* et *height* sont exprimés en pixels. Le pointeur vers un *GdkWindow* pointe sur la fenêtre courante car les ressources d'un pixmap n'ont de signification que dans le contexte de l'écran où il doit s'afficher.

```
GdkPixmap* gdk_pixmap_create_from_data( GdkWindow  *window,
                                        gchar      *data,
                                        gint        width,
                                        gint        height,
                                        gint        depth,
                                        GdkColor   *fg,
                                        GdkColor   *bg );
```

Cette fonction est utilisée pour créer un pixmap d'une profondeur donnée (nombre de couleurs) à partir de la donnée spécifiée pas *data*. *fg* et *bg* sont les couleurs à utiliser pour l'avant et l'arrière-plan.

```
GdkPixmap* gdk_pixmap_create_from_xpm( GdkWindow  *window,
                                       GdkBitmap **mask,
                                       GdkColor   *transparent_color,
                                       const gchar *filename );
```

Le format XPM est une représentation des pixmaps reconnue par le système X Window. Il est largement utilisé et de nombreux utilitaires pour créer des fichiers d'images à ce format sont disponibles. Le fichier *filename* doit contenir une image dans ce format qui sera chargée dans la structure pixmap. Le masque *mask* indique quels sont les bits opaques du pixmap. Tous les autres bits sont colorisés en utilisant la couleur spécifiée par *transparent_color*. Un exemple d'utilisation est présenté ci-dessous.

```
GdkPixmap* gdk_pixmap_create_from_xpm_d (GdkWindow  *window,
                                         GdkBitmap **mask,
                                         GdkColor   *transparent_color,
                                         gchar     **data);
```

De petites images peuvent être intégrées dans un programme sous la forme de données *data* au format XPM. Un pixmap est créé en utilisant ces données au lieu de les lire dans un fichier. Un exemple de telles données est :

```
/* XPM */
static const char * xpm_data[] = {
"16 16 3 1",
"       c None",
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "};

void gdk_pixmap_destroy( GdkPixmap  *pixmap );
```

Lorsqu'on a utilisé un pixmap et que l'on en a plus besoin tout de suite, il est préférable de libérer la ressource en utilisant un appel à *gdk_pixmap_destroy*. Les pixmaps doivent être considérées comme une ressource précieuse.

Quand le pixmap est créé, on peut l'afficher comme un widget GTK. On doit créer un widget pixmap qui contiendra le pixmap GDK. Ceci est réalisé de la façon suivante :

```
GtkWidget* gtk_pixmap_new( GdkPixmap  *pixmap,
                           GdkBitmap  *mask );
```

Les autres fonctions pour les widgets pixmap sont :

```
guint gtk_pixmap_get_type( void );
void  gtk_pixmap_set( GtkPixmap  *pixmap,
                      GdkPixmap  *val,
                      GdkBitmap  *mask);
void  gtk_pixmap_get( GtkPixmap  *pixmap,
                      GdkPixmap **val,
                      GdkBitmap **mask);
```

gtk_pixmap_set sert à changer le pixmap pris en charge par le widget. *val* est le pixmap créé par le GDK.

Voici un exemple illustrant l'utilisation d'un pixmap dans un bouton :

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

```
/* données XPM d'une icône "Ouvrir fichier" */
static const char * xpm_data[] = {
"16 16 3 1",
"       c None",
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "};


/* Termine l'application lorsqu'elle est appelée
 * via le signal "delete_event" */

void close_application( GtkWidget *widget, GdkEvent *event, gpointer *data )
{
    gtk_main_quit();
}


/* Invoquée lorsque le bouton est cliqué. Affiche simplement
 * un message. */

void button_clicked( GtkWidget *widget, gpointer *data )
{
    printf( "bouton cliqué\n" );
}


int main( int argc, char *argv[] )
{
    /* GtkWidget est le type pour déclarer les widgets */

    GtkWidget *window, *pixmapwid, *button;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    GtkStyle *style;

    /* Crée la fenêtre principale et attache le signal "delete_event" pour
     * terminer l'application */

    gtk_init( &argc, &argv );
    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_signal_connect( GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (close_application), NULL );
    gtk_container_border_width( GTK_CONTAINER (window), 10 );
    gtk_widget_show( window );

    /* Utilisation de GDK pour créer le pixmap */
```

```
    style = gtk_widget_get_style( window );
    pixmap = gdk_pixmap_create_from_xpm_d( window->window,  &mask,
                                           &style->bg[GTK_STATE_NORMAL],
                                           (gchar **)xpm_data );

    /* Création d'un widget pixmap GTK pour contenir le pixmap GDK */

    pixmapwid = gtk_pixmap_new( pixmap, mask );
    gtk_widget_show( pixmapwid );

    /* Création d'un bouton pour contenir le widget pixmap */

    button = gtk_button_new();
    gtk_container_add( GTK_CONTAINER(button), pixmapwid );
    gtk_container_add( GTK_CONTAINER(window), button );
    gtk_widget_show( button );

    gtk_signal_connect( GTK_OBJECT(button), "clicked",
                        GTK_SIGNAL_FUNC(button_clicked), NULL );

    /* Affichage de la fenêtre */
    gtk_main ();

    return 0;
}
```

Pour charger un fichier à partir d'un fichier XPM appelé *icon0.xpm* se trouvant dans le répertoire courant, on aurait créé le pixmap ainsi :

```
    /* Charge un pixmap à partir d'un fichier */

    pixmap = gdk_pixmap_create_from_xpm( window->window, &mask,
                                         &style->bg[GTK_STATE_NORMAL],
                                         "./icon0.xpm" );
    pixmapwid = gtk_pixmap_new( pixmap, mask );
    gtk_widget_show( pixmapwid );
    gtk_container_add( GTK_CONTAINER(window), pixmapwid );
```

Utilisation des formes

Un désavantage de l'utilisation des pixmaps est que l'objet affiché est toujours rectangulaire, quelle que soit l'image. On voudrait pouvoir créer des bureaux et des applications avec des icônes ayant des formes plus naturelles. Par exemple, pour une interface de jeu, on aimerait avoir des boutons ronds à pousser. Pour faire cela, on doit utiliser des fenêtres avec des formes.

Une fenêtre avec forme est simplement un pixmap dont les pixels du fond sont transparents. Ainsi, lorsque l'image d'arrière−plan est multicolore, on ne la cache pas avec le bord de notre icône. L'exemple suivant affiche une image de brouette sur le bureau.

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

/* XPM */
static char * WheelbarrowFull_xpm[] = {
"48 48 64 1",
"       c None",
".      c #DF7DCF3CC71B",
"X      c #965875D669A6",
"o      c #71C671C671C6",
"O      c #A699A289A699",
"+      c #965892489658",
"@      c #8E38410330C2",
```

```
"#      c #D75C7DF769A6",
"$      c #F7DECF3CC71B",
"%      c #96588A288E38",
"&      c #A69992489E79",
"*      c #8E3886178E38",
"=      c #104008200820",
"-      c #596510401040",
";      c #C71B30C230C2",
":      c #C71B9A699658",
">      c #618561856185",
",      c #20811C712081",
"<      c #104000000000",
"1      c #861720812081",
"2      c #DF7D4D344103",
"3      c #79E769A671C6",
"4      c #861782078617",
"5      c #41033CF34103",
"6      c #000000000000",
"7      c #49241C711040",
"8      c #492445144924",
"9      c #082008200820",
"0      c #69A618611861",
"q      c #B6DA71C65144",
"w      c #410330C238E3",
"e      c #CF3CBAEAB6DA",
"r      c #71C6451430C2",
"t      c #EFBEDB6CD75C",
"y      c #28A208200820",
"u      c #186110401040",
"i      c #596528A21861",
"p      c #71C661855965",
"a      c #A69996589658",
"s      c #30C228A230C2",
"d      c #BEFBA289AEBA",
"f      c #596545145144",
"g      c #30C230C230C2",
"h      c #8E3882078617",
"j      c #208118612081",
"k      c #38E30C300820",
"l      c #30C2208128A2",
"z      c #38E328A238E3",
"x      c #514438E34924",
"c      c #618555555965",
"v      c #30C2208130C2",
"b      c #38E328A230C2",
"n      c #28A228A228A2",
"m      c #41032CB228A2",
"M      c #104010401040",
"N      c #492438E34103",
"B      c #28A2208128A2",
"V      c #A699596538E3",
"C      c #30C21C711040",
"Z      c #30C218611040",
"A      c #965865955965",
"S      c #618534D32081",
"D      c #38E31C711040",
"F      c #082000000820",
"                                        ",
"          .XoO                          ",
"         +@#$%o&                        ",
"         *=-;#::o+                      ",
"           >,<12#:34                    ",
"             45671#:X3                  ",
"               +89<02qwo                ",
"e*                >,67;ro               ",
```

```
"ty>                  459@>+&&                    ",
"$2u+                   ><ipas8*                  ",
"%$;=*                 *3:.Xa.dfg>                ",
"Oh$;ya              *3d.a8j,Xe.d3g8+             ",
" Oh$;ka           *3d$a8lz,,xxc:.e3g54           ",
"  Oh$;kO        *pd$%svbzz,sxxxxfX..&wn>         ",
"   Oh$@mO    *3dthwlsslszjzxxxxxxx3:td8M4       ",
"    Oh$@g& *3d$XNlvvvlllm,mNwxxxxxxxfa.:,B*     ",
"     Oh$@,Od.czllllzlmmqV@V#V@fxxxxxxxf:%j5&   ",
"      Oh$1hd5lllslllCCZrV#r#:#2AxxxxxxxxxcdwM*  ",
"       OXq6c.%8vvvllZZiqqApA:mq:Xxcpcxxxxxfdc9* ",
"        2r<6gde3bllZZrVi7S@SV77A::qApxxxxxxfdcM ",
"        :,q-6MN.dfmZZrrSS:#riirDSAX@Af5xxxxxfevo",
"         +A26jguXtAZZZC7iDiCCrVVii7Cmmmxxxxxx%3g",
"          *#16jszN..3DZZZZrCVSA2rZrV7Dmmwxxxx&en",
"           p2yFvzssXe:fCZZCiiD7iiZDiDSSZwwxx8e*>",
"           OA1<jzxwwc:$d%NDZZZZCCCZCCZZCmxxfd.B ",
"            3206Bwxxszx%et.eaAp77m77mmmf3&eeeg* ",
"             @26MvzxNzvlbwfpdettttttttttt.c,n&  ",
"             *;16=lsNwwNwgsvslbwwvccc3pcfu<o    ",
"              p;<69BvwwsszslllbBlllllllu<5+     ",
"              OS0y6FBlvvvzvzss,u=Blllj=54       ",
"               c1-699Blvllllu7k96MMMg4          ",
"               *10y8n6FjvllllB<166668           ",
"                S-kg+>666<M<996-y6n<8*          ",
"                p71=4 m69996kD8Z-66698&&        ",
"                &i0ycm6n4 ogk17,0<6666g         ",
"                 N-k-<>     >=01-kuu666>        ",
"                 ,6ky&      &46-10ul,66,        ",
"                 Ou0<>       o66y<ulw<66&       ",
"                  *kk5       >66By7=xu664       ",
"                   <<M4      466lj<Mxu66o       ",
"                   *>>       +66uv,zN666*       ",
"                             566,xxj669         ",
"                             4666FF666>         ",
"                              >966666M          ",
"                               oM6668+          ",
"                                 *4             ",
"                                                ",
"                                               "};


/* Termine l'application lorsqu'elle est appelée
 * (via le signal "delete_event"). */

void close_application( GtkWidget *widget, GdkEvent *event, gpointer *data )
{
    gtk_main_quit();
}


int main (int argc, char *argv[])
{
    GtkWidget *window, *pixmap, *fixed;
    GdkPixmap *gdk_pixmap;
    GdkBitmap *mask;
    GtkStyle *style;
    GdkGC *gc;

    /* crée la fenêtre principale et attache le signal "delete_event"
     * pour terminer l'application. On notera que la fenêtre principale
     * n'a pas de barre de titre car nous la faisons surgir. */

    gtk_init (&argc, &argv);
    window = gtk_window_new( GTK_WINDOW_POPUP );
```

```
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (close_application), NULL);
    gtk_widget_show (window);

    /* Création du pixmap et du widget pixmap */

    style = gtk_widget_get_default_style();
    gc = style->black_gc;
    gdk_pixmap = gdk_pixmap_create_from_xpm_d( window->window, &mask,
                                               &style->bg[GTK_STATE_NORMAL],
                                               WheelbarrowFull_xpm );
    pixmap = gtk_pixmap_new( gdk_pixmap, mask );
    gtk_widget_show( pixmap );

    /* Pour afficher le pixmap, on utilise un widget fixe pour placer
     * le pixmap */

    fixed = gtk_fixed_new();
    gtk_widget_set_usize( fixed, 200, 200 );
    gtk_fixed_put( GTK_FIXED(fixed), pixmap, 0, 0 );
    gtk_container_add( GTK_CONTAINER(window), fixed );
    gtk_widget_show( fixed );

    /* On masque tout sauf l'image elle-même */

    gtk_widget_shape_combine_mask( window, mask, 0, 0 );

    /* Affichage de la fenêtre */

    gtk_widget_set_uposition( window, 20, 400 );
    gtk_widget_show( window );
    gtk_main ();

    return 0;
}
```

Pour rendre l'image de la brouette sensible aux clics, on peut lui attacher le signal "button_press_event" pour lui faire faire quelque chose. Les quelques lignes suivantes font que l'image sera sensible à un clic souris qui provoquera l'arrêt de l'application.

```
gtk_widget_set_events( window,
                       gtk_widget_get_events( window ) |
                       GDK_BUTTON_PRESS_MASK );

gtk_signal_connect( GTK_OBJECT(window), "button_press_event",
                    GTK_SIGNAL_FUNC(close_application), NULL );
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

8. Widgets containers

8.1 Bloc-notes

Le widget bloc-notes est un ensemble de « pages » qui se chevauchent. Chaque page contient des informations différentes. Récemment, ce widget est devenu plus commun dans la

programmation des interfaces graphiques et c'est un bon moyen de montrer des blocs d'information similaires qui justifient une séparation de leur affichage.

Le premier appel de fonction que l'on doit connaître est, vous l'aviez deviné, celui qui crée un widget bloc−notes.

```
GtkWidget* gtk_notebook_new (void);
```

Lorsque le bloc−notes a été créé, il y a 12 fonctions permettant de travailler sur les blocs−notes. Étudions−les séparément.

La première permet de positionner les indicateurs de pages. Ceux−ci (désignés par le mot « tab » (signet)), peuvent se trouver en haut, en bas, à gauche ou à droite des pages.

```
void gtk_notebook_set_tab_pos (GtkNotebook *notebook, GtkPositionType pos);
```

GtkPositionType peut prendre les valeurs suivantes qu'il n'est pas nécessaire d'expliquer :

- GTK_POS_LEFT
- GTK_POS_RIGHT
- GTK_POS_TOP
- GTK_POS_BOTTOM

GTK_POS_TOP est la valeur par défaut.

La fonction suivante permet d'ajouter des pages à un bloc−notes. Il y a trois façons d'ajouter des pages. Regardons les deux premières qui sont très semblables.

```
void gtk_notebook_append_page (GtkNotebook *notebook, GtkWidget *child,
                               GtkWidget *tab_label);

void gtk_notebook_prepend_page (GtkNotebook *notebook, GtkWidget *child,
                                GtkWidget *tab_label);
```

Ces fonctions ajoutent des pages au bloc−notes**notebook* en les insérant à la fin (*append*) ou au début (*prepend*). **child* est le widget qui est placé dans la page du bloc−notes, et **tab_label* est le label de la page qui est ajoutée.

La troisième fonction ajoutant une page à un bloc−notes conserve toutes les propriétés des deux précédentes, mais elle nous permet en plus de spécifier la position où l'on désire insérer cette page.

```
void gtk_notebook_insert_page (GtkNotebook *notebook, GtkWidget *child,
                               GtkWidget *tab_label, gint position);
```

Les paramètres sont les mêmes que *_append_* et *_prepend_* sauf qu'il y en a un de plus : *position*. Celui−ci sert à spécifier l'endroit où cette page sera insérée.

Maintenant que nous savons insérer une page, voyons comment en supprimer une.

```
void gtk_notebook_remove_page (GtkNotebook *notebook, gint page_num);
```

Cette fonction ôte la page spécifiée par *page_num* du widget **notebook*.

Pour connaître la page courante d'un bloc−notes, on dispose de la fonction :

```
gint gtk_notebook_current_page (GtkNotebook *notebook);
```

Les deux fonctions suivantes permettent de passer à la page suivante ou précédente d'un bloc−notes. Il suffit de faire l'appel de la fonction adéquate avec le widget sur lequel on veut opérer. Remarque : lorsqu'on est sur la dernière page du bloc−notes et que l'on appelle *gtk_notebook_next_page*, on revient à la première page. De même, si l'on est sur la première page et que l'onappelle *gtk_notebook_prev_page*, on se retrouve sur sa dernière page.

```
void gtk_notebook_next_page (GtkNoteBook *notebook);
void gtk_notebook_prev_page (GtkNoteBook *notebook);
```

La fonction qui suit permet de choisir la page « active ». Si vous voulez ouvrir le bloc−notes à la page 5, par exemple, vous utiliserez cette fonction. Sans elle, le bloc−notes s'ouvre sur sa première page par défaut.

```
void gtk_notebook_set_page (GtkNotebook *notebook, gint page_num);
```

Les deux fonctions suivantes ajoutent ou ôtent les indicateurs de page et le contour du bloc−notes, respectivement.

```
void gtk_notebook_set_show_tabs (GtkNotebook *notebook, gint show_tabs);
void gtk_notebook_set_show_border (GtkNotebook *notebook, gint show_border);
```

show_tabs et *show_border* peuvent valoir TRUE ou FALSE (0 ou 1).

Voyons maintenant un exemple, il est tiré du code de *testgtk.c* de la distribution GTK et montre l'utilisation des 13 fonctions. Ce petit programme crée une fenêtre contenant un bloc−notes et six boutons. Le bloc−notes contient 11 pages, ajoutées par trois moyens différents : à la fin, au milieu et au début. Les boutons permettent de faire tourner les indicateurs de page, ajouter/ôter les indicateurs et le contour, ôter une page, passer à la page suivante et précédente, et sortir du programme.

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

/* Rotation des indicateurs de page  */

void rotate_book (GtkButton *button, GtkNotebook *notebook)
{
    gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos +1) %4);
}

/* Ajout/Suppression des indicateurs de pages et des contours */

void tabsborder_book (GtkButton *button, GtkNotebook *notebook)
{
    gint tval = FALSE;
    gint bval = FALSE;
    if (notebook->show_tabs == 0)
            tval = TRUE;
    if (notebook->show_border == 0)
            bval = TRUE;

    gtk_notebook_set_show_tabs (notebook, tval);
    gtk_notebook_set_show_border (notebook, bval);
}

/* Suppression d'une page */

void remove_book (GtkButton *button, GtkNotebook *notebook)
{
    gint page;
```

```
    page = gtk_notebook_current_page(notebook);
    gtk_notebook_remove_page (notebook, page);

    /* Il faut rafraîchir  le widget --
     * ce qui force le widget à se redessiner. */

      gtk_widget_draw(GTK_WIDGET(notebook), NULL);
}

void delete (GtkWidget *widget, GdkEvent *event, gpointer *data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;
    GtkWidget *notebook;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *checkbutton;
    int i;
    char bufferf[32];
    char bufferl[32];

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete), NULL);

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new(2,6,TRUE);
    gtk_container_add (GTK_CONTAINER (window), table);

    /* Création d'un bloc-notes, placement des indicateurs de page. */

    notebook = gtk_notebook_new ();
    gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
    gtk_table_attach_defaults(GTK_TABLE(table), notebook, 0,6,0,1);
    gtk_widget_show(notebook);

    /* Ajoute un groupe de pages à la fin du bloc-notes. */

    for (i=0; i < 5; i++) {
        sprintf(bufferf, "Append Frame %d", i+1);
        sprintf(bufferl, "Page %d", i+1);

        frame = gtk_frame_new (bufferf);
        gtk_container_border_width (GTK_CONTAINER (frame), 10);
        gtk_widget_set_usize (frame, 100, 75);
        gtk_widget_show (frame);

        label = gtk_label_new (bufferf);
        gtk_container_add (GTK_CONTAINER (frame), label);
        gtk_widget_show (label);

        label = gtk_label_new (bufferl);
        gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame, label);
    }
```

```
/* Ajoute une page à un endroit précis. */

checkbutton = gtk_check_button_new_with_label ("Cochez moi !");
gtk_widget_set_usize(checkbutton, 100, 75);
gtk_widget_show (checkbutton);

label = gtk_label_new ("Emplacement de la nouvelle page");
gtk_container_add (GTK_CONTAINER (checkbutton), label);
gtk_widget_show (label);
label = gtk_label_new ("Ajout de page");
gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkbutton, label, 2);

/* Ajout de pages au début du bloc-notes */

for (i=0; i < 5; i++) {
    sprintf(bufferf, "Prepend Frame %d", i+1);
    sprintf(bufferl, "Page %d", i+1);

    frame = gtk_frame_new (bufferf);
    gtk_container_border_width (GTK_CONTAINER (frame), 10);
    gtk_widget_set_usize (frame, 100, 75);
    gtk_widget_show (frame);

    label = gtk_label_new (bufferf);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_widget_show (label);

    label = gtk_label_new (bufferl);
    gtk_notebook_prepend_page (GTK_NOTEBOOK(notebook), frame, label);
}

/* Configuration de la page de départ (page 4) */

gtk_notebook_set_page (GTK_NOTEBOOK(notebook), 3);


/* Création des boutons */

button = gtk_button_new_with_label ("Fermer");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (delete), NULL);
gtk_table_attach_defaults(GTK_TABLE(table), button, 0,1,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("Page suivante");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_notebook_next_page,
                           GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 1,2,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("Page précédente");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) gtk_notebook_prev_page,
                           GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 2,3,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("Position des indicateurs");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           (GtkSignalFunc) rotate_book, GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 3,4,1,2);
gtk_widget_show(button);
```

```
    button = gtk_button_new_with_label ("Indicateurs/Contours oui/non");
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               (GtkSignalFunc) tabsborder_book,
                               GTK_OBJECT (notebook));
    gtk_table_attach_defaults(GTK_TABLE(table), button, 4,5,1,2);
    gtk_widget_show(button);

    button = gtk_button_new_with_label ("Oter page");
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               (GtkSignalFunc) remove_book,
                               GTK_OBJECT(notebook));
    gtk_table_attach_defaults(GTK_TABLE(table), button, 5,6,1,2);
    gtk_widget_show(button);

    gtk_widget_show(table);
    gtk_widget_show(window);

    gtk_main ();

    return 0;
}
```

En espérant que ceci vous aide à créer des blocs−notes pour vos applications GTK.

8.2 Fenêtres avec barres de défilement

Les fenêtres avec barres de défilement servent à créer des zones défilantes à l'intérieur d'une vraie fenêtre. On peut insérer n'importe quel widget dans ces fenêtres, ils seront accessibles quelle que soit leur taille en utilisant les barres de défilement.

La fonction suivante sert à créer une fenêtre avec barre de défilement :

```
GtkWidget* gtk_scrolled_window_new (GtkAdjustment *hadjustment,
                                    GtkAdjustment *vadjustment);
```

Le premier paramètre est l'ajustement horizontal, et le second l'ajustement vertical. Ils sont presque toujours positionnés à NULL.

```
void gtk_scrolled_window_set_policy      (GtkScrolledWindow *scrolled_window,
                                          GtkPolicyType      hscrollbar_policy,
                                          GtkPolicyType      vscrollbar_policy);
```

Cela permet de configurer le fonctionnement des barres de défilement. Le premier paramètre est la fenêtre à défilement que l'on veut modifier, le second configure le fonctionnement de la barre horizontale et le troisième celui de la barre verticale.

Ce fonctionnement peut être GTK_POLICY AUTOMATIC ou GTK_POLICY_ALWAYS. GTK_POLICY_AUTOMATIC décidera automatiquement de votre besoin en barres de défilement, alors que GTK_POLICY_ALWAYS mettra toujours celles−ci.

Voici un exemple simple qui place 100 boutons commutateurs dans une fenêtre à défilement. Je n'ai commenté que les parties qui sont nouvelles pour vous.

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

void destroy(GtkWidget *widget, gpointer *data)
{
    gtk_main_quit();
```

```
}

int main (int argc, char *argv[])
{
    static GtkWidget *window;
    GtkWidget *scrolled_window;
    GtkWidget *table;
    GtkWidget *button;
    char buffer[32];
    int i, j;

    gtk_init (&argc, &argv);

    /* Création d'une boîte de dialogue pour y placer la fenêtre à défilement.
     * Une boîte de dialogue est une fenêtre comme les autres sauf qu'elle contient
     * une vbox et un séparateur horizontal. Ce n'est qu'un raccourci pour créer de
     * zones de dialogue. */

    window = gtk_dialog_new ();
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        (GtkSignalFunc) destroy, NULL);
    gtk_window_set_title (GTK_WINDOW (window), "dialog");
    gtk_container_border_width (GTK_CONTAINER (window), 0);

    /* Création d'une fenêtre à défilement. */

    scrolled_window = gtk_scrolled_window_new (NULL, NULL);

    gtk_container_border_width (GTK_CONTAINER (scrolled_window), 10);

    /* La gestion des barres est soit GTK_POLICY AUTOMATIC, soit  GTK_POLICY_ALWAYS
     * GTK_POLICY_AUTOMATIC décide automatiquement s'il faut ou non des barres,
     * GTK_POLICY_ALWAYS met toujours des barres
     * Le premier paramètre correspond à la barre horizontale,
     * le second à la barre verticale. */

    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);

    /* Création d'une boîte de dialogue */

    gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                        TRUE, TRUE, 0);
    gtk_widget_show (scrolled_window);

    /* Création d'une table de 10x10 cases. */

    table = gtk_table_new (10, 10, FALSE);

    /* Configure l'espace des lignes et des colonnes de 10 pixels */

    gtk_table_set_row_spacings (GTK_TABLE (table), 10);
    gtk_table_set_col_spacings (GTK_TABLE (table), 10);

    /* Place la table fans la fenêtre à défilement */

    gtk_container_add (GTK_CONTAINER (scrolled_window), table);
    gtk_widget_show (table);

    /* Crée une grille de boutons commutateurs dans la table */

    for (i = 0; i < 10; i++)
            for (j = 0; j < 10; j++) {
                sprintf (buffer, "bouton (%d,%d)\n", i, j);
                button = gtk_toggle_button_new_with_label (buffer);
```

```
                gtk_table_attach_defaults (GTK_TABLE (table), button,
                                           i, i+1, j, j+1);
                gtk_widget_show (button);
            }

        /* Ajoute un bouton « Fermer » en bas de la boîte de dialogue */

        button = gtk_button_new_with_label ("Fermer");
        gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                                   (GtkSignalFunc) gtk_widget_destroy,
                                   GTK_OBJECT (window));

        /* On met ce bouton en « bouton par défaut ». */

        GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
        gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button, TRUE, T

        /* Récupère le bouton par défaut. Le fait de presser la touche « Entrée »
         * activera le bouton. */

        gtk_widget_grab_default (button);
        gtk_widget_show (button);

        gtk_widget_show (window);

        gtk_main();

        return(0);
    }
```

Essayez de changer la taille de la fenêtre et faites attention aux réactions des barres de défilement. On peut aussi utiliser la fonction *gtk_widget_set_usize()* pour configurer la taille par défaut de la fenêtre et des autres widgets.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

9. Widgets listes

Le widget *GtkList* sert de container vertical pour des widgets *GtkListItem*.

Un widget *GtkList* possède sa propre fenêtre pour recevoir les événements et sa propre couleur de fond qui est habituellement blanche. Comme il est directement dérivé de *GtkContainer*, il peut être traité comme tel en utilisant la macro GTK_CONTAINER(List) : voir le widget *GtkContainer* pour en savoir plus.

On doit d'abord connaître l'utilisation des *GList* et des fonctions *g_list_*()* qui leur sont liées pour pouvoir utiliser pleinement le widget *GtkList*.

Un champ de la structure d'un widget *GtkList* nous intéresse particulièrement :

```
struct _GtkList
{
  ...
  GList *selection;
  guint selection_mode;
```

```
  ...
};
```

Le champ *selection* d'un *GtkList* pointe sur une liste chaînée de tous les items qui sont sélectionnés, ou vaut NULL si aucune sélection n'est faite. Ainsi, pour connaître la sélection courante, on consulte le champ *GTK_LIST()−>selection* mais on ne doit pas le modifier car ses champs internes sont gérés par les fonctions *gtk_list_*()*.

Le champ *selection_mode* détermine les options de sélection d'un *GtkList* et donc le contenu du champ du *GTK_LIST()−>selection* :

selection_mode peut avoir l'une des valeurs suivantes :

- GTK_SELECTION_SINGLE − *selection* vaut NULL ou contient un pointeur vers un seul item sélectionné.
- GTK_SELECTION_BROWSE − *selection* vaut NULL si la liste ne contient aucun widget ou seulement des widgets non sensitifs. Sinon, ce champ contient un pointeur vers une seule structure Glist, et donc vers exactement un item.
- GTK_SELECTION_MULTIPLE − *selection* vaut NULL si aucun item n'est sélectionné ou pointe vers le premier item sélectionné. Ce dernier point à son tour vers le second item, etc.
- GTK_SELECTION_EXTENDED − *selection* vaut toujours NULL.

La valeur par défaut est GTK_SELECTION_MULTIPLE.

9.1 Signaux

```
void GtkList::selection_changed (GtkList *LIST)
```

Ce signal sera invoqué à chaque fois que le champ sélection d'un GtkList a changé. Cela arrive lorsqu'un fils d'un GtkList a été sélectionné ou désélectionné.

```
void GtkList::select_child (GtkList *LIST, GtkWidget *CHILD)
```

Ce signal est invoqué lorsqu'un fils du GtkList va être sélectionné. Ceci arrive principalement lors d'appels à *gtk_list_select_item(), gtk_list_select_child()* et lors d'appuis de boutons. Quelques fois, il est indirectement déclenché lorsque des fils sont ajoutés ou supprimés du GtkList.

```
void GtkList::unselect_child (GtkList *LIST, GtkWidget *CHILD)
```

Ce signal est invoqué lorsqu'un fils du GtkList va être désélectionné. Cela arrive principalement lors d'appels à *gtk_list_unselect_item(), gtk_list_unselect_child()*, et lors d'appuis de boutons. Quelques fois, il est indirectement déclenché lorsque des fils sont ajoutés ou supprimés du GtkList.

9.2 Fonctions

```
guint gtk_list_get_type (void)
```

Retourne l'identificateur de type « GtkList ».

```
GtkWidget* gtk_list_new (void)
```

Crée un nouvel objet « GtkList ». Le nouveau widget est retourné sous la forme d'un pointeur vers un objet « GtkWidget ». NULL est retourné en cas d'erreur.

```
void gtk_list_insert_items (GtkList *LIST, GList *ITEMS, gint POSITION)
```

Insère des items dans *LIST*, à partir de *POSITION*. *ITEMS* est une liste doublement chaînée où chaque noeud doit pointer vers un nouveau *GtkListItem*. Les noeuds *GList* de *ITEMS* sont pris en charge par *LIST*.

```
void gtk_list_append_items (GtkList *LIST, GList *ITEMS)
```

Insère des items à la fin de *LIST* selon le même principe que *gtk_list_insert_items*. Les noeuds *GList* de *ITEMS* sont pris en charge par *LIST*.

```
void gtk_list_prepend_items (GtkList *LIST, GList *ITEMS)
```

Insère des items au début de *LIST* selon le même principe que *gtk_list_insert_items*. Les noeuds *GList* de *ITEMS* sont pris en charge par *LIST*.

```
void gtk_list_remove_items (GtkList *LIST, GList *ITEMS)
```

Ôte des items de *LIST*. *ITEMS* est une liste doublement chaînée dont chaque noeud pointe vers un fils direct de *LIST*. Il est de la responsabilité de l'appelant de faire un appel à *g_list_free(ITEMS)* après cela. L'appelant doit aussi détruire lui−même les items.

```
void gtk_list_clear_items (GtkList *LIST, gint START, gint END)
```

Ôte et détruit des items de *LIST*. Un widget est concerné si sa position courante dans *LIST* est dans l'intervalle spécifié par *START* et *END*.

```
void gtk_list_select_item (GtkList *LIST, gint ITEM)
```

Invoque le signal *GtkList::select_child* pour un item spécifié par sa position courante dans *LIST*.

```
void gtk_list_unselect_item (GtkList *LIST, gint ITEM)
```

Invoque le signal *GtkList::unselect_child* pour un item spécifié par sa position courante dans *LIST*.

```
void gtk_list_select_child (GtkList *LIST, GtkWidget *CHILD)
```

Invoque le signal *GtkList::select_child* pour le fils *CHILD* spécifié.

```
void gtk_list_unselect_child (GtkList *LIST, GtkWidget *CHILD)
```

Invoque le signal *GtkList::unselect_child* pour le fils *CHILD* spécifié.

```
gint gtk_list_child_position (GtkList *LIST, GtkWidget *CHILD)
```

Retourne la position de *CHILD* dans *LIST*. Retourne −1 en cas d'erreur.

```
void gtk_list_set_selection_mode (GtkList *LIST, GtkSelectionMode MODE)
```

Configure *LIST* dans le mode de sélection *MODE* qui peut être GTK_SELECTION_SINGLE, GTK_SELECTION_BROWSE, GTK_SELECTION_MULTIPLE ou GTK_SELECTION_EXTENDED.

```
GtkList* GTK_LIST (gpointer OBJ)
```

Convertit un pointeur générique en « <\em GtkList*\ ». Voir *Standard Macros::*, pour plus d'informations.

```
GtkListClass* GTK_LIST_CLASS (gpointer CLASS)
```

Convertit un pointeur générique en « GtkListClass* ». Voir *Standard Macros::*, pour plus d'informations.

```
gint GTK_IS_LIST (gpointer OBJ)
```

Détermine si un pointeur générique référence un objet « GtkList ». Voir *Standard Macros::*, pour plus d'informations.

9.3 Exemple

Voici un programme affichant les changements de sélection dans une *GtkList* et permettant d'« emprisonner » des items en les sélectionnant avec le bouton droit de la souris.

```
/* Compilez ce programme avec :
 * $ gcc -L/usr/X11R6/lib/ -I/usr/local/include/ -lgtk -lgdk -lglib -lX11 -lm -Wall
 */
#include        <gtk/gtk.h>
#include        <stdio.h>

/* Chaîne pour stocker les données dans les items de la liste. */

const   gchar   *list_item_data_key="list_item_data";


/* prototypes des gestionnaires de signaux que l'on connectera au widget GtkList. *

static  void    sigh_print_selection    (GtkWidget      *gtklist,
                                         gpointer       func_data);
static  void    sigh_button_event       (GtkWidget      *gtklist,
                                         GdkEventButton *event,
                                         GtkWidget      *frame);


/* fonction principale pour configurer l'interface utilisateur */

gint main (int argc, gchar *argv[])
{
    GtkWidget       *separator;
    GtkWidget       *window;
    GtkWidget       *vbox;
    GtkWidget       *scrolled_window;
    GtkWidget       *frame;
    GtkWidget       *gtklist;
    GtkWidget       *button;
    GtkWidget       *list_item;
    GList           *dlist;
    guint           i;
    gchar           buffer[64];


    /* initialise gtk (et donc gdk) */

    gtk_init(&argc, &argv);


    /* Création d'une fenêtre pour placer tous les widgets.
     * Connexion de  gtk_main_quit() à l'événement "destroy" de
```

```
 * la fenêtre afin de prendre en charge les événements « fermeture d'une
 * fenêtre » du gestionnaire de fenêtre. */

window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "Exemple de widget GtkList");
gtk_signal_connect(GTK_OBJECT(window),
                   "destroy",
                   GTK_SIGNAL_FUNC(gtk_main_quit),
                   NULL);


/* À l'intérieur de la fenêtre, on a besoin d'une boîte pour placer
 * verticalement les widgets. */

vbox=gtk_vbox_new(FALSE, 5);
gtk_container_border_width(GTK_CONTAINER(vbox), 5);
gtk_container_add(GTK_CONTAINER(window), vbox);
gtk_widget_show(vbox);

/* Fenêtre à défilement pour placer le widget GtkList à l'intérieur. */

scrolled_window=gtk_scrolled_window_new(NULL, NULL);
gtk_widget_set_usize(scrolled_window, 250, 150);
gtk_container_add(GTK_CONTAINER(vbox), scrolled_window);
gtk_widget_show(scrolled_window);

/* Création du widget GtkList
 * Connexion du gestionnaire de signal sigh_print_selection() au signal
 * "selection_changed" du GtkList pour afficher les items sélectionnés
 * à chaque fois que la sélection change. */

gtklist=gtk_list_new();
gtk_container_add(GTK_CONTAINER(scrolled_window), gtklist);
gtk_widget_show(gtklist);
gtk_signal_connect(GTK_OBJECT(gtklist),
                   "selection_changed",
                   GTK_SIGNAL_FUNC(sigh_print_selection),
                   NULL);

/* Création d'une « Prison » pour y mettre un item. */

frame=gtk_frame_new("Prison");
gtk_widget_set_usize(frame, 200, 50);
gtk_container_border_width(GTK_CONTAINER(frame), 5);
gtk_frame_set_shadow_type(GTK_FRAME(frame), GTK_SHADOW_OUT);
gtk_container_add(GTK_CONTAINER(vbox), frame);
gtk_widget_show(frame);

/* Connexion du gestionnaire de signal sigh_button_event() au signal
 * « mise au arrêts » des items du GtkList. */

gtk_signal_connect(GTK_OBJECT(gtklist),
                   "button_release_event",
                   GTK_SIGNAL_FUNC(sigh_button_event),
                   frame);

/* Création d'un séparateur. */

separator=gtk_hseparator_new();
gtk_container_add(GTK_CONTAINER(vbox), separator);
gtk_widget_show(separator);

/* Création d'un bouton et connexion de son signal "clicked" à la
 * destruction de la fenêtre. */
```

```
    button=gtk_button_new_with_label("Fermeture");
    gtk_container_add(GTK_CONTAINER(vbox), button);
    gtk_widget_show(button);
    gtk_signal_connect_object(GTK_OBJECT(button),
                              "clicked",
                              GTK_SIGNAL_FUNC(gtk_widget_destroy),
                              GTK_OBJECT(window));


    /* Création de 5 items, chacun ayant son propre label.
     * Ajout de ceux-ci au GtkList en utilisant gtk_container_add().
     * On interroge le texte du label et on l'associe avec
     * list_item_data_key à chaque item. */

    for (i=0; i<5; i++) {
        GtkWidget       *label;
        gchar           *string;

        sprintf(buffer, "ListItemContainer avec Label #%d", i);
        label=gtk_label_new(buffer);
        list_item=gtk_list_item_new();
        gtk_container_add(GTK_CONTAINER(list_item), label);
        gtk_widget_show(label);
        gtk_container_add(GTK_CONTAINER(gtklist), list_item);
        gtk_widget_show(list_item);
        gtk_label_get(GTK_LABEL(label), &string);
        gtk_object_set_data(GTK_OBJECT(list_item),
                            list_item_data_key,
                            string);
    }
    /* Création de 5 autres labels. Cette fois-ci, on utilise
     * gtk_list_item_new_with_label(). On ne peut interroger la chaîne
     * des labels car on n'a pas les pointeurs de labels et on associe
     * donc simplement le list_item_data_key de chaque item ayant la même
     * chaîne de texte pour l'ajouter au items que l'on place dans une liste
     * doublement chaînée (GList). On les ajoute alors par un simple appel à
     * gtk_list_append_items().
     * Comme on utilise g_list_prepend() pour mettre les items dans la liste
     * doublement chaînée, leur ordre sera décroissant (au lieu d'être croissant si
     * on utilisait g_list_append()). */

    dlist=NULL;
    for (; i<10; i++) {
        sprintf(buffer, "Item avec le label %d", i);
        list_item=gtk_list_item_new_with_label(buffer);
        dlist=g_list_prepend(dlist, list_item);
        gtk_widget_show(list_item);
        gtk_object_set_data(GTK_OBJECT(list_item),
                            list_item_data_key,
                            "Item avec label intégré");
    }
    gtk_list_append_items(GTK_LIST(gtklist), dlist);

    /* Enfin, on veut voir la fenêtre... */

    gtk_widget_show(window);

    /* Lancement de la boucle principale de gtk */

    gtk_main();

    /* On arrive ici après que gtk_main_quit() ait été appelée lorsque
     * la fenêtre principale a été détruite. */

}
```

```
/* Gestionnaire de signal connecté aux événements boutons presser/relâcher
 * du GtkList. */

void
sigh_button_event       (GtkWidget      *gtklist,
                         GdkEventButton *event,
                         GtkWidget      *frame)
{
    /* On ne fait quelque chose que si le troisième bouton (celui de droite) a été
     * relâché. */

    if (event->type==GDK_BUTTON_RELEASE &&
        event->button==3) {
        GList           *dlist, *free_list;
        GtkWidget       *new_prisoner;

        /* On recherche l'item sélectionné à ce moment précis.
         * Ce sera notre prisonnier ! */

        dlist=GTK_LIST(gtklist)->selection;
        if (dlist)
                new_prisoner=GTK_WIDGET(dlist->data);
        else
                new_prisoner=NULL;

        /* On recherche les items déjà prisonniers et on les
         * remet dans la liste.
         * Il ne faut pas oublier de libérer la liste doublement
         * chaînée que gtk_container_children() retourne. */

        dlist=gtk_container_children(GTK_CONTAINER(frame));
        free_list=dlist;
        while (dlist) {
            GtkWidget       *list_item;

            list_item=dlist->data;

            gtk_widget_reparent(list_item, gtklist);

            dlist=dlist->next;
        }
        g_list_free(free_list);

        /* Si l'on a un nouveau prisonnier, on l'ôte du GtkList et on le place
         * dans le cadre « Prison ». On doit désélectionner l'item avant.

        if (new_prisoner) {
            GList   static_dlist;

            static_dlist.data=new_prisoner;
            static_dlist.next=NULL;
            static_dlist.prev=NULL;

            gtk_list_unselect_child(GTK_LIST(gtklist),
                                    new_prisoner);
            gtk_widget_reparent(new_prisoner, frame);
        }
    }
}

/* Gestionnaire de signal appelé lorsque le GtkList
 * émet le signal "selection_changed". */

void
```

```
sigh_print_selection    (GtkWidget      *gtklist,
                         gpointer       func_data)
{
    GList   *dlist;

    /* Recherche dans la liste doublement chaînée des items sélectionnés
     * du GtkList, à faire en lecture seulement ! */

    dlist=GTK_LIST(gtklist)->selection;

    /* S'il n'y a pas d'items sélectionné, il n'y a rien d'autre à faire que
     * de le dire à l'utilisateur. */

    if (!dlist) {
        g_print("Sélection nettoyée\n");
        return;
    }
    /* Ok, on a une sélection et on l'affiche. */

    g_print("La sélection est ");

    /* On récupère l'item dans la liste doublement chaînée
     * puis on interroge la donnée associée par list_item_data_key
     * et on l'affiche. */

    while (dlist) {
        GtkObject       *list_item;
        gchar           *item_data_string;

        list_item=GTK_OBJECT(dlist->data);
        item_data_string=gtk_object_get_data(list_item,
                                             list_item_data_key);
        g_print("%s ", item_data_string);

        dlist=dlist->next;
    }
    g_print("\n");
}
```

9.4 Widget item de liste

Le widget *GtkListItem* sert de container pour contenir un fils, lui fournissant des fonctions de sélection/déséselection exactement comme le widget GtkList les demande pour ses fils.

Un *GtkListItem* a sa propre fenêtre pour recevoir les événements et a sa propre couleur de fond, habituellement blanche.

Comme il est directement dérivé d'un *GtkItem*, il peut être traité comme tel en utilisant la macro GTK_ITEM(ListItem), reportez−vous à la section sur le widget GtkItem pour plus de détail sur celui−ci. Habituellement, un *GtkListItem* contient juste un label pour identifier, par exemple, un nom de fichier dans un *GtkList* −− la fonction appropriée *gtk_list_item_new_with_label()* est donc fournie. Le même effet peut être obtenu en créant un *GtkLabel* à part, en configurant son alignement avec *xalign*=0 et *yalign*=0.5 suivi d'un ajout ultérieur au *GtkListItem*.

Tout comme on n'est pas forcé d'ajouter un *GtkLabel* à un *GtkListItem*, on peut aussi ajouter un *GtkVBox* ou un *GtkArrow* etc. à un *GtkListItem*.

9.5 Signaux

Un *GtkListItem* ne crée pas de nouveaux signaux par lui−même, mais hérite de ceux d'un *GtkItem*. Voir *GtkItem::*, pour plus d'informations.

9.6 Fonctions

```
guint gtk_list_item_get_type (void)
```

Retourne l'identificateur du type « GtkListItem ».

```
GtkWidget* gtk_list_item_new (void)
```

Création d'un objet *GtkListItem*. Le nouveau widget est retourné sous la forme d'un pointeur vers un objet *GtkWidget*. NULL est retourné en cas d'erreur.

```
GtkWidget* gtk_list_item_new_with_label (gchar *LABEL)
```

Création d'un objet *GtkListItem* ayant un simple *GtkLabel* comme seul fils. Le nouveau widget est retourné sous la forme d'un pointeur vers un objet *GtkWidget*. NULL est retourné en cas d'erreur.

```
void gtk_list_item_select (GtkListItem *LIST_ITEM)
```

Cette fonction est surtout un emballage de *gtk_item_select (GTK_ITEM (list_item))* qui émettra le signal *GtkItem::select*. Voir *GtkItem::*, pour plus d'informations.

```
void gtk_list_item_deselect (GtkListItem *LIST_ITEM)
```

Cette fonction est surtout un emballage de *gtk_item_deselect (GTK_ITEM (list_item))* qui émettra le signal *GtkItem::deselect*. Voir *GtkItem::*, pour plus d'informations.

```
GtkListItem* GTK_LIST_ITEM (gpointer OBJ)
```

Convertit un pointeur générique en *GtkListItem**. Voir *Standard Macros::* pour plus d'informations.

```
GtkListItemClass* GTK_LIST_ITEM_CLASS (gpointer CLASS)
```

Convertit un pointeur générique en *GtkListItemClass**. Voir *Standard Macros::* pour plus d'informations.

```
gint GTK_IS_LIST_ITEM (gpointer OBJ)
```

Détermine si un pointeur générique se réfère à un objet *GtkListItem*. Voir *Standard Macros::* pour plus d'informations.

9.7 Exemple

L'exemple des GtkList couvre aussi l'utilisation des GtkListItem. Étudiez−le attentivement.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

10. Widgets sélections de fichiers

Le widget sélection de fichier est un moyen simple et rapide pour afficher un fichier dans une boîte de dialogue. Il est complet, avec des boutons Ok, Annuler et Aide. C'est un bon moyen de raccourcir les temps de programmation.

Pour créer une boîte de sélection de fichier, on utilise :

```
GtkWidget* gtk_file_selection_new (gchar *title);
```

Pour configurer le nom de fichier, par exemple pour aller dans un répertoire précis ou donner un nom de fichier par défaut, on utilise la fonction :

```
void gtk_file_selection_set_filename (GtkFileSelection *filesel, gchar *filename);
```

Pour récupérer le texte que l'utilisateur a entré, ou sur lequel il a cliqué, on utilisera la fonction :

```
gchar* gtk_file_selection_get_filename (GtkFileSelection *filesel);
```

Des pointeurs permettent d'accéder aux widgets contenus dans la widget de sélection de fichiers. Ce sont :

- dir_list
- file_list
- selection_entry
- selection_text
- main_vbox
- ok_button
- cancel_button
- help_button

Le plus souvent, on utilise les pointeurs *ok_button*, *cancel_button*, et *help_button* pour préciser leurs utilisations.

Voici un exemple emprunté à *testgtk.c* et modifié pour fonctionner tout seul. Comme vous le verrez, il n'y a pas grand chose à faire pour créer un wigdget de sélection de fichier. Cependant, dans cet exemple, si le bouton Aide apparaît à l'écran, il ne fait rien car aucun signal ne lui est attaché.

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

/* Récupère le nom de fichier sélectionné et l'affiche sur la console. */

void file_ok_sel (GtkWidget *w, GtkFileSelection *fs)
{
    g_print ("%s\n", gtk_file_selection_get_filename (GTK_FILE_SELECTION (fs)));
}

void destroy (GtkWidget *widget, gpointer *data)
{
    gtk_main_quit ();
}

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

```
        GtkWidget *filew;

        gtk_init (&argc, &argv);

        /* Création d'un widget de sélection de fichier. */

        filew = gtk_file_selection_new ("File selection");

        gtk_signal_connect (GTK_OBJECT (filew), "destroy",
                            (GtkSignalFunc) destroy, &filew);

        /* Connexion de ok_button à la fonction file_ok_sel() */

        gtk_signal_connect (GTK_OBJECT (GTK_FILE_SELECTION (filew)->ok_button),
                            "clicked", (GtkSignalFunc) file_ok_sel, filew );

        /* Connexion de cancel_button pour détruire le widget */

        gtk_signal_connect_object (GTK_OBJECT (GTK_FILE_SELECTION (filew)->cancel_butto
                                   "clicked", (GtkSignalFunc) gtk_widget_destroy,
                                   GTK_OBJECT (filew));

        /* Configuration du nom de fichier, comme s'il s'agissait d'un dialogue de
         * sauvegarde et que nous donnions un nom de fichier par défaut. */

        gtk_file_selection_set_filename (GTK_FILE_SELECTION(filew),
                                         "penguin.png");

        gtk_widget_show(filew);
        gtk_main ();
        return 0;
    }
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

11. Widgets Menu

Il y a deux façons de créer des menus, la facile et la compliquée. Les deux ont leur utilité, mais on peut généralement utiliser l'*usine à menus* (c'est la méthode facile...). La méthode « compliquée » consiste à créer tous les menus en utilisant directement les appels. La méthode facile consiste à utiliser les appels *gtk_menu_factory*. C'est beaucoup plus simple, mais chaque approche a ses avantages et inconvénients.

L'usine à menus est beaucoup plus facile à utiliser, elle facilite aussi l'ajout d'autres menus. Par contre, écrire quelques fonctions permettant de créer des menus en utilisant la méthode manuelle peut être le début d'un long chemin avant une quelconque utilisation. Avec l'usine à menus, il n'est pas possible d'ajouter des images ou des « / » aux menus.

11.1 Création manuelle de menus

Selon la tradition pédagogique, nous commencerons par le plus compliqué :)

Regardons les fonctions utilisées pour créer les menus. La première sert à créer un nouveau menu.

```
GtkWidget *gtk_menu_bar_new()
```

Cette fonction crée une nouvelle barre de menu. On utilise la fonction *gtk_container_add* pour la placer dans une fenêtre, ou les fonctions *box_pack* pour la placer dans une boîte − la même que pour les boutons.

```
GtkWidget *gtk_menu_new();
```

Cette fonction retourne un pointeur vers un nouveau menu, il n'est jamais montré (avec *gtk_widget_show*), il ne fait que contenir les items du menu. Ceci deviendra plus clair lorsque nous étudierons l'exemple ci−dessous.

Les deux appels suivants servent à créer des items de menu qui seront placés dans le menu.

```
GtkWidget *gtk_menu_item_new()
```

et

```
GtkWidget *gtk_menu_item_new_with_label(const char *label)
```

Ces appels servent à créer les menus qui doivent être affichés. On doit bien faire la différence entre un « menu » qui est créé avec *gtk_menu_new()* et un « item de menu » créé avec les fonctions *gtk_menu_item_new()*. L'item de menu sera un véritable bouton avec une action associée alors qu'un menu sera un container contenant les items.

```
gtk_menu_item_append()

gtk_menu_item_set_submenu()
```

Les fonctions *gtk_menu_new_with_label()* et *gtk_menu_new()* sont telles que vous les attendiez après avoir étudié les boutons. L'une crée un nouvel item de menu contenant déjà un label, et l'autre ne fait que créer un item de menu vide.

Voici les étapes pour créer une barre de menu avec des menus attachés :

- Créer un nouveau menu avec *gtk_menu_new()*
- Créer un item de menu avec *gtk_menu_item_new()*. Ce sera la racine du menu, le texte apparaissant ici sera aussi sur la barre de menu.
- Utiliser plusieurs appels à *gtk_menu_item_new()* pour chaque item que l'on désire dans le menu. Utiliser *gtk_menu_item_append()* pour placer chacun de ces items les uns après les autres. Cela crée une liste d'items de menu.
- Utiliser *gtk_menu_item_set_submenu()* pour attacher les items de menus nouvellement créés à l'item de menu racine (celui créé à la seconde étape).
- Créer une nouvelle barre de menu avec *gtk_menu_bar_new()*. Cette étape ne doit être faite qu'une fois lorsque l'on crée une série de menu sur une seule barre de menus.
- Utiliser *gtk_menu_bar_append()* pour placer le menu racine dans la barre de menu.

La création d'un menu surgissant est presque identique. La différence est que le menu n'est pas posté « automatiquement » par une barre de menu, mais explicitement en appelant la fonction *gtk_menu_popup()* par un événement « bouton pressé ».

Suivez ces étapes

- Créer une fonction de gestion d'événement. Elle doit avoir le prototype

```
static gint handler(GtkWidget *widget, GdkEvent
```

```
*event);
```

et elle utilisera l'événement *event* pour savoir où faire surgir le menu.

- Ce gestionnaire, si l'événement est un appui sur un bouton souris, traite *event* comme un événement bouton (ce qu'il est) et l'utilise, de la façon indiquée dans le code d'exemple, pour passer l'information à *gtk_menu_popup()*.
- Lier ce gestionnaire à un widget avec :

```
gtk_signal_connect_object(GTK_OBJECT(widget),
"event", GTK_SIGNAL_FUNC (handler),
GTK_OBJECT(menu));
```

où *widget* est le widget auquel vous le liez, *handler* est le gestionnaire, et *menu* est un menu créé avec *gtk_menu_new()*. Cela peut être un menu qui est aussi posté par une barre de menu, comme le montre l'exemple.

11.2 Exemple de menu manuel

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

static gint button_press (GtkWidget *, GdkEvent *);
static void menuitem_response (GtkWidget *, gchar *);


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

    GtkWidget *window;
    GtkWidget *menu;
    GtkWidget *menu_bar;
    GtkWidget *root_menu;
    GtkWidget *menu_items;
    GtkWidget *vbox;
    GtkWidget *button;
    char buf[128];
    int i;

    gtk_init (&argc, &argv);

    /* Création d'un fenêtre */

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW (window), "Test de Menu GTK");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                       (GtkSignalFunc) gtk_exit, NULL);

    /* Initialise le widget menu -- Attention : n'appelez jamais
     * gtk_show_widget() pour le widget menu !!!
     * C'est le menu qui contient les items de menu, celui qui surgira
     * lorsque vous cliquez sur le « menu racine » de l'application. */

    menu = gtk_menu_new();

    /* Voici le menu racine dont le label sera le nom du menu affiché sur la barre
     * de menu. Il n'a pas de gestionnaire de signal attaché car il ne fait
     * qu'afficher le reste du menu lorsqu'il est pressé. */

    root_menu = gtk_menu_item_new_with_label("Menu racine");

    gtk_widget_show(root_menu);
```

```
/* Puis, on crée une petite boucle créant trois entrées pour « menu test »
 * Notez l'appel à gtk_menu_append(). Ici, on ajoute une liste d'items à
 * notre menu. Normalement, on devrait aussi capturer le signal "clicked"
 * pour chacun des items et configurer une fonction de rappel pour lui,
 * mais on l'a omis ici pour gagner de la place. */

for(i = 0; i < 3; i++)
    {
        /* Copie des noms dans buf. */

        sprintf(buf, "Sous−menu Test − %d", i);

        /* Création d'un nouveau item de menu avec un nom... */

        menu_items = gtk_menu_item_new_with_label(buf);

        /* ...et ajout de celui−ci dans le menu. */

        gtk_menu_append(GTK_MENU (menu), menu_items);

        /* On fait quelque chose d'intéressant lorsque l'item est
         * sélectionné. */

        gtk_signal_connect (GTK_OBJECT(menu_items), "activate",
            GTK_SIGNAL_FUNC(menuitem_response), (gpointer)
            g_strdup(buf));

        /* Affichage du widget. */

        gtk_widget_show(menu_items);
    }

/* Maintenant, on spécifié que nous voulons que notre nouveau « menu »
 * soit le menu du « menu racine ». */

gtk_menu_item_set_submenu(GTK_MENU_ITEM (root_menu), menu);

/* Création d'une vbox pour y mettre un menu et un bouton. */

vbox = gtk_vbox_new(FALSE, 0);
gtk_container_add(GTK_CONTAINER(window), vbox);
gtk_widget_show(vbox);

/* Création d'une barre de menus pour contenir les menus. Puis, on
 * l'ajoute à notre fenêtre principale. */

menu_bar = gtk_menu_bar_new();
gtk_box_pack_start(GTK_BOX(vbox), menu_bar, FALSE, FALSE, 2);
gtk_widget_show(menu_bar);

/* Création d'un bouton pour y attacher le menu. */

button = gtk_button_new_with_label("Pressez moi");
gtk_signal_connect_object(GTK_OBJECT(button), "event",
    GTK_SIGNAL_FUNC (button_press), GTK_OBJECT(menu));
gtk_box_pack_end(GTK_BOX(vbox), button, TRUE, TRUE, 2);
gtk_widget_show(button);

/* Finalement, on ajoute l'item de menu à la barre de menu −−
 * c'est l'item de menu racine sur lequel je me suis déchaîné ;−) */

gtk_menu_bar_append(GTK_MENU_BAR (menu_bar), root_menu);

/* Affichage de la fenêtre. */
```

```
    gtk_widget_show(window);

    gtk_main ();

    return 0;
}



/* On répond à un appui sur le bouton en postant un nouveau menu passé comme
 * un widget.
 *
 * On remarque que le paramètre "widget" est le menu à poster, PAS le bouton
 * qui a été pressé. */


static gint button_press (GtkWidget *widget, GdkEvent *event)
{

    if (event->type == GDK_BUTTON_PRESS) {
        GdkEventButton *bevent = (GdkEventButton *) event;
        gtk_menu_popup (GTK_MENU(widget), NULL, NULL, NULL, NULL,
                        bevent->button, bevent->time);

        /* On indique à l'appelant que l'on a géré cet événement. */

        return TRUE;
    }

    /* On indique à l'appelant que l'on n'a pas géré cet événement. */

    return FALSE;
}


/* Affiche une chaîne lorsqu'un item de menu est choisi. */

static void menuitem_response (GtkWidget *widget, gchar *string)
{
    printf("%s\n", string);
}
```

Vous pouvez aussi configurer un item de menu pour qu'il ne soit pas sélectionnable et, en utilisant une table de raccourcis clavier, lier des touches aux fonctions du menu.

11.3 Utilisation de GtkMenuFactory

Maintenant que nous avons exploré la voie difficile, nous allons voir l'utilisation des appels *gtk_menu_factory*.

11.4 Exemple d'usine à menu

Voici un exemple utilisant l'usine à menu de GTK. Le premier fichier est *menus.h*. Nous séparerons *menus.c* et *main.c* à cause des variables globales utilisées dans le fichier *menus.c*.

```
#ifndef __MENUS_H__
#define __MENUS_H__

#ifdef __cplusplus
extern "C" {
```

```
#endif /* __cplusplus */

void get_main_menu (GtkWidget **menubar, GtkAcceleratorTable **table);
void menus_create(GtkMenuEntry *entries, int nmenu_entries);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __MENUS_H__ */
```

Voici le fichier *menus.c* :

```
#include <gtk/gtk.h>
#include <strings.h>

#include "main.h"


static void menus_remove_accel(GtkWidget * widget, gchar * signal_name, gchar * pat
static gint menus_install_accel(GtkWidget * widget, gchar * signal_name, gchar key,
void menus_init(void);
void menus_create(GtkMenuEntry * entries, int nmenu_entries);


/* Structure GtkMenuEntry utilisée pour créer les menus. Le premier champ
 * est la chaîne de définition du menu. Le second, la touche de raccourci
 * utilisée pour accéder à cette fonction du menu avec le clavier.
 * Le troisième est la fonction de rappel à utiliser lorsque l'item de menu
 * est choisi (par la touche de raccourci ou avec la souris). Le dernier
 * élément est la donnée à passer à la fonction de rappel. */


static GtkMenuEntry menu_items[] =
{
        {"<Main>/Fichier/Nouveau", "<control>N", NULL, NULL},
        {"<Main>/Fichier/Ouvrir", "<control>O", NULL, NULL},
        {"<Main>/Fichier/Sauver", "<control>S", NULL, NULL},
        {"<Main>/Fichier/Sauver sous", NULL, NULL, NULL},
        {"<Main>/Fichier/<separator>", NULL, NULL, NULL},
        {"<Main>/Fichier/Quitter", "<control>Q", file_quit_cmd_callback, "OK, c'est
        {"<Main>/Options/Test", NULL, NULL, NULL}
};

/* Calcul du nombre d'éléments de menu_item */

static int nmenu_items = sizeof(menu_items) / sizeof(menu_items[0]);

static int initialize = TRUE;
static GtkMenuFactory *factory = NULL;
static GtkMenuFactory *subfactory[1];
static GHashTable *entry_ht = NULL;

void get_main_menu(GtkWidget ** menubar, GtkAcceleratorTable ** table)
{
    if (initialize)
            menus_init();

    if (menubar)
            *menubar = subfactory[0]->widget;
    if (table)
            *table = subfactory[0]->table;
}
```

```
void menus_init(void)
{
    if (initialize) {
        initialize = FALSE;

        factory = gtk_menu_factory_new(GTK_MENU_FACTORY_MENU_BAR);
        subfactory[0] = gtk_menu_factory_new(GTK_MENU_FACTORY_MENU_BAR);

        gtk_menu_factory_add_subfactory(factory, subfactory[0], "<Main>");
        menus_create(menu_items, nmenu_items);
    }
}

void menus_create(GtkMenuEntry * entries, int nmenu_entries)
{
    char *accelerator;
    int i;

    if (initialize)
            menus_init();

    if (entry_ht)
            for (i = 0; i < nmenu_entries; i++) {
                accelerator = g_hash_table_lookup(entry_ht, entries[i].path);
                if (accelerator) {
                    if (accelerator[0] == '\0')
                            entries[i].accelerator = NULL;
                    else
                            entries[i].accelerator = accelerator;
                }
            }
    gtk_menu_factory_add_entries(factory, entries, nmenu_entries);

    for (i = 0; i < nmenu_entries; i++)
            if (entries[i].widget) {
                gtk_signal_connect(GTK_OBJECT(entries[i].widget), "install_accelera
                                   (GtkSignalFunc) menus_install_accel,
                                   entries[i].path);
                gtk_signal_connect(GTK_OBJECT(entries[i].widget), "remove_accelerat
                                   (GtkSignalFunc) menus_remove_accel,
                                   entries[i].path);
            }
}

static gint menus_install_accel(GtkWidget * widget, gchar * signal_name, gchar key,
{
    char accel[64];
    char *t1, t2[2];

    accel[0] = '\0';
    if (modifiers & GDK_CONTROL_MASK)
            strcat(accel, "<control>");
    if (modifiers & GDK_SHIFT_MASK)
            strcat(accel, "<shift>");
    if (modifiers & GDK_MOD1_MASK)
            strcat(accel, "<alt>");

    t2[0] = key;
    t2[1] = '\0';
    strcat(accel, t2);

    if (entry_ht) {
        t1 = g_hash_table_lookup(entry_ht, path);
        g_free(t1);
    } else
```

```
                entry_ht = g_hash_table_new(g_string_hash, g_string_equal);

        g_hash_table_insert(entry_ht, path, g_strdup(accel));

        return TRUE;
    }

    static void menus_remove_accel(GtkWidget * widget, gchar * signal_name, gchar * pat
    {
        char *t;

        if (entry_ht) {
            t = g_hash_table_lookup(entry_ht, path);
            g_free(t);

            g_hash_table_insert(entry_ht, path, g_strdup(""));
        }
    }

    void menus_set_sensitive(char *path, int sensitive)
    {
        GtkMenuPath *menu_path;

        if (initialize)
                menus_init();

        menu_path = gtk_menu_factory_find(factory, path);
        if (menu_path)
                gtk_widget_set_sensitive(menu_path->widget, sensitive);
        else
                g_warning("Impossible de configurer la sensitivité d'un menu qui n'exis
    }
```

Voici *main.h* :

```
#ifndef __MAIN_H__
#define __MAIN_H__


#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

void file_quit_cmd_callback(GtkWidget *widget, gpointer data);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __MAIN_H__ */
```

Et, enfin, *main.c* :

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

#include "main.h"
#include "menus.h"


int main(int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *main_vbox;
    GtkWidget *menubar;
```

```
    GtkAcceleratorTable *accel;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect(GTK_OBJECT(window), "destroy",
                       GTK_SIGNAL_FUNC(file_quit_cmd_callback),
                       "WM destroy");
    gtk_window_set_title(GTK_WINDOW(window), "Usine à menu");
    gtk_widget_set_usize(GTK_WIDGET(window), 300, 200);

    main_vbox = gtk_vbox_new(FALSE, 1);
    gtk_container_border_width(GTK_CONTAINER(main_vbox), 1);
    gtk_container_add(GTK_CONTAINER(window), main_vbox);
    gtk_widget_show(main_vbox);

    get_main_menu(&menubar, &accel);
    gtk_window_add_accelerator_table(GTK_WINDOW(window), accel);
    gtk_box_pack_start(GTK_BOX(main_vbox), menubar, FALSE, TRUE, 0);
    gtk_widget_show(menubar);

    gtk_widget_show(window);
    gtk_main();

    return(0);
}

/* Juste une démonstration du fonctionnement des fonctions de rappel
 * lorsqu'on utilise l'usine à menus. Souvent, on met tous les rappels
 * des menus dans un fichier séparé, ce qui assure une meilleure
 * organisation. */

void file_quit_cmd_callback (GtkWidget *widget, gpointer data)
{
    g_print ("%s\n", (char *) data);
    gtk_exit(0);
}
```

Un *makefile* pour que cela soit plus facile à compiler :

```
CC      = gcc
PROF    = -g
C_FLAGS =  -Wall $(PROF) -L/usr/local/include -DDEBUG
L_FLAGS =  $(PROF) -L/usr/X11R6/lib -L/usr/local/lib
L_POSTFLAGS = -lgtk -lgdk -lglib -lXext -lX11 -lm
PROGNAME = at

O_FILES = menus.o main.o

$(PROGNAME): $(O_FILES)
        rm -f $(PROGNAME)
        $(CC) $(L_FLAGS) -o $(PROGNAME) $(O_FILES) $(L_POSTFLAGS)

.c.o:
        $(CC) -c $(C_FLAGS) $<

clean:
        rm -f core *.o $(PROGNAME) nohup.out
distclean: clean
        rm -f *~
```

Pour l'instant, il n'y a que cet exemple. Une explication et de nombreux commentaires seront intégrés plus tard.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

12. Widgets non documentés

On a besoin de leurs auteurs! :). Participez à notre didacticiel.

Si vous devez utiliser un de ces widgets non documentés, je vous recommande fortement de consulter leurs fichiers en−têtes respectifs dans la distribution GTK. Les noms de fonctions du GTK sont très parlantes. Lorsque vous avez compris comment les choses fonctionnent, il n'est pas difficile de savoir comment utiliser un widget à partir des déclarations de ses fonctions. Cela, avec quelques exemples de codes pris ailleurs, devrait ne pas poser de problème.

Lorsque vous avez compris toutes les fonctions d'un nouveau widget non documenté, pensez à écrire un didacticiel pour que les autres puissent bénéficier du temps que vous y avez passé.

12.1 Entrées de texte

12.2 Sélections de couleurs

12.3 Contrôle d'intervalle

12.4 Règles

12.5 Boîtes de texte

12.6 Prévisualisation

(Ceci peut devoir être réécrit pour suivre le style du reste de ce didacticiel).

```
Les prévisualisateurs servent à plusieurs choses dans GIMP/GTK. La
plus importante est celle−ci : les images de haute qualité peuvent
occuper des dizaines de mega−octets en mémoire − facilement ! Toute
opération sur une image aussi grosse implique un temps de traitement
élevé. Si cela vous prend 5 à 10 essais (i.e. 10 à 20 étapes puisque
vous devez recommencer lorsque vous avez fait une erreur) pour choisir
la bonne modification, cela prendra littéralement des heures pour
produire la bonne image − pour peu que vous ne manquiez pas de mémoire
avant. Ceux qui on passé des heures dans les chambres noires de
développement couleur connaissent cette sensation. Les
prévisualisations sont notre planche de salut !

L'aspect pénible de l'attente n'est pas le seul problème. souvent, il
est utile de comparer les versions « Avant » et « Après » côte à côte
ou, au pire l'une après l'autre. Si vous travaillez avec de grosses
images et des attentes de 10 secondes, l'obtention des versions «
Avant » et « Après » est, pour le moins, difficile. Pour des images de
30Mo (4"x6", 600dpi, 24 bits), la comparaison côte à côte est
impossible pour la plupart des gens, et la comparaison séquentielle
```

n'est guère mieux. Les prévisualisations sont notre planche de salut !

Mais il y a plus. Les prévisualisations permettent les pré-prévisualisations côte à côte. En d'autres termes, vous écrivez un plug-in (par exemple la simulation filterpack) qui aurait plusieurs prévisualisations de ce-que-ce-serait-si-vous-faisiez-ceci. Une approche comme celle ci agit comme une sorte de palette de prévisualisation et est très pratique pour les petites modifications. Utilisons les prévisualisations !

Encore plus : pour certains plug-ins une intervention humaine en temps réel, spécifique aux images, peut s'avérer nécessaire. Dans le plug-in SuperNova, par exemple, on demande à l'utilisateur d'entrer les coordonnées du centre de la future supernova. La façon la plus simple de faire cela, vraiment, est de présenter une prévisualisation à l'utilisateur et de lui demander de choisir interactivement le point. Utilisons les prévisualisations !

Enfin, quelques utilisations diverses : on peut utiliser les prévisualisations, même lorsqu'on ne travaille pas avec de grosses images. Elles sont utiles, par exemple, lorsqu'on veut avoir un rendu de motifs complexes. (Testez le vénérable plug-in Diffraction et d'autres !). Comme autre exemple, regardez le plug-in de rotation de couleur (travail en cours). Vous pouvez aussi utiliser les prévisualisations pour des petits logos dans vos plug-ins et même pour une photo de vous, l'Auteur. Utilisons les prévisualisations !

Quand ne pas utiliser les prévisualisations

N'utilisez pas les prévisualisations pour les graphes, les tracés, etc. GDK est bien plus rapide pour ça. N'utilisez les que pour les images !

Utilisons les prévisualisations !

Vous pouvez mettre une prévisualisation dans à peu près n'importe quoi. Dans une vbox, une hbox, un bouton, etc. Mais elles donnent leur meilleur d'elles-mêmes dans des cadres resserrés autour d'elles. Les prévisualisations n'ont, par elles-mêmes, aucun contour et semblent plates sans eux. (Bien sûr, si c'est cet aspect que vous voulez...). Les cadres serrés fournissent les bordures nécessaires.

[Image][Image]

Les prévisualisations sont, à bien des égards, comme tous les autres widgets de GTK (avec tout ce que cela implique) sauf qu'il disposent d'une fonctionnalité supplémentaire : ils doivent être remplis avec une image ! Nous traiterons d'abord exclusivement de l'aspect GTK des prévisualisations, puis nous verrons comment les remplir.

```
                              /* Création d'un widget prévisualisation,
                               * configuration de sa taille et affichage */
GtkWidget *preview;
preview=gtk_preview_new(GTK_PREVIEW_COLOR)
                              /* Autre option :
                              GTK_PREVIEW_GRAYSCALE);*/

gtk_preview_size (GTK_PREVIEW (preview), WIDTH, HEIGHT);
gtk_widget_show(preview);
my_preview_rendering_function(preview);
```

Ah oui, comme je le disais, les prévisualisations rendent mieux dans des cadres :

```
GtkWidget *create_a_preview(int        Width,
                            int        Height,
                            int        Colorfulness)
{
  GtkWidget *preview;
  GtkWidget *frame;

  frame = gtk_frame_new(NULL);
  gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
  gtk_container_border_width (GTK_CONTAINER(frame),0);
  gtk_widget_show(frame);

  preview=gtk_preview_new (Colorfulness?GTK_PREVIEW_COLOR
                                       :GTK_PREVIEW_GRAYSCALE);
  gtk_preview_size (GTK_PREVIEW (preview), Width, Height);
  gtk_container_add(GTK_CONTAINER(frame),preview);
  gtk_widget_show(preview);

  my_preview_rendering_function(preview);
  return frame;
}
```

Ceci est ma prévisualisation de base. Cette fonction retourne le cadre « père », on peut ainsi le placer ailleurs dans notre interface. Bien sûr, on peut passer le cadre « père » en paramètre à cette fonction. Dans de nombreuses situations, toutefois, le contenu de la prévisualisation est changée continuellement par notre application. En ce cas, on peut passer un pointeur vers une prévisualisation à la fonction <em/create_a_preview()/ et avoir ainsi un contrôle sur elle plus tard.

Un point plus important qui pourra un jour vous faire économiser beaucoup de temps. Quelques fois, il est souhaitable de mettre un label à votre prévisualisation. Par exemple, on peut nommer la prévisualisation contenant l'image originale « Original » et celle contenant l'image modifiée « Moins Originale ». Il peut vous arriver de placer la prévisualisation avec le label approprié dans une vbox. L'effet inattendu est que si le label est plus large que la prévisualisation (taille de cette dernière, taille de la fonte du label, etc), le cadre s'élargit et ne convient plus à la prévisualisation. Le même problème se passera probablement dans d'autres situations aussi.

[Image]

La solution consiste à placer la prévisualisation et le label dans une table de 2x2 en les attachant avec les paramètres suivants (c'est l'une des possibilités, bien sûr. La clé consiste à ne pas mettre GTK_FILL dans le second attachement)«nbsp;:

```
gtk_table_attach(GTK_TABLE(table),label,0,1,0,1,
                 0,
                 GTK_EXPAND|GTK_FILL,
                 0,0);
gtk_table_attach(GTK_TABLE(table),frame,0,1,1,2,
                 GTK_EXPAND,
                 GTK_EXPAND,
                 0,0);
```

Et voici le résultat :

[Image]

Divers

Rendre une prévisualisation cliquable se fait très facilement en la plaçant dans un

Remplir une prévisualisation

Afin de nous familiariser avec les bases de ce remplissage, créons le motif suivant

[Image]

```
void
my_preview_rendering_function(GtkWidget     *preview)
{
#define SIZE 100
#define HALF (SIZE/2)

  guchar *row=(guchar *) malloc(3*SIZE); /* 3 bits par point */
  gint i, j;                             /* Coordonnées     */
  double r, alpha, x, y;

  if (preview==NULL) return; /* J'ajoute généralement ceci quand je */
                             /* veux éviter des plantages stupides  */
                             /* Vous devez vous assurer que tout a  */
                             /* été correctement initialisé !       */

  for (j=0; j < ABS(cos(2*alpha)) ) { /* Sommes-nous dans la forme ? */
                                      /* glib.h contient ABS(x).   */
      row[i*3+0] = sqrt(1-r)*255;     /* Definit rouge             */
      row[i*3+1] = 128;               /* Definit vert              */
      row[i*3+2] = 224;               /* Definit bleu              */
    }                                 /* "+0" est pour l'alignement ! */
    else {
      row[i*3+0] = r*255;
      row[i*3+1] = ABS(sin((float)i/SIZE*2*PI))*255;
      row[i*3+2] = ABS(sin((float)j/SIZE*2*PI))*255;
    }
  }
  gtk_preview_draw_row( GTK_PREVIEW(preview),row,0,j,SIZE);

  /* Insère "row" dans "preview" en partant du point de */
  /* coordonnées (0,j) première colonne, j_ième ligne allant de SIZE */
  /* pixels vers la droite */
}

  free(row); /* on récupère un peu d'espace */
  gtk_widget_draw(preview,NULL); /* qu'est-ce que ça fait ? */
  gdk_flush(); /* et ça ?  */
}
```

Ceux qui n'utilisent pas GIMP en ont suffisamment vu pour faire déjà beaucoup de choses. Pour ceux qui l'utilisent, j'ai quelques précisions à ajouter.

Prévisualisation d'image

Il est pratique de conserver une version réduite de l'image ayant juste assez de pixels pour remplir la prévisualisation. Ceci est possible en choisissant chaque énième pixel où n est le ratio de la taille de l'image par rapport à la taille de la visualisation. Toutes les opérations suivantes (y compris le remplissage des prévisualisations) sont alors réalisées seulement sur le nombre réduit de pixels. Ce qui suit est mon implantation de la réduction d'image (Gardez à l'esprit que je n'ai que quelques notions de base en C !).

(ATTENTION : CODE NON TESTÉ !!!)

```
typedef struct {
  gint      width;
  gint      height;
  gint      bbp;
  guchar    *rgb;
  guchar    *mask;
} ReducedImage;

enum {
  SELECTION_ONLY,
  SELCTION_IN_CONTEXT,
  ENTIRE_IMAGE
};

ReducedImage *Reduce_The_Image(GDrawable *drawable,
                               GDrawable *mask,
                               gint LongerSize,
                               gint Selection)
{
  /* Cette fonction réduit l'image à la taille de prévisualisation choisie */
  /* La taille de la prévisualisation est déterminée par LongerSize, i.e.  */
  /* la plus grande des deux dimensions. Ne fonctionne qu'avec des images  */
  /* RGB !                                                                 */

  gint RH, RW;          /* Hauteur et Largeur réduites                     */
  gint width, height;   /* Largeur et Hauteur de la surface à réduire      */
  gint bytes=drawable->bpp;
  ReducedImage *temp=(ReducedImage *)malloc(sizeof(ReducedImage));

  guchar *tempRGB, *src_row, *tempmask, *src_mask_row,R,G,B;
  gint i, j, whichcol, whichrow, x1, x2, y1, y2;
  GPixelRgn srcPR, srcMask;
  gint NoSelectionMade=TRUE; /* Suppose que l'on traite l'image entière    */

  gimp_drawable_mask_bounds (drawable->id, &x1, &y1, &x2, &y2);
  width  = x2-x1;
  height = y2-y1;
  /* S'il y a une SELECTION, on récupère ses frontières ! */

  if (width != drawable->width && height != drawable->height)
    NoSelectionMade=FALSE;
  /* On vérifie si l'utilisateur a rendu une sélection active         */
  /* Ceci sera important plus tard, lorsqu'on créera un masque réduit */

  /* Si on veut prévisualiser l'image entière, supprimer ce qui suit ! */
  /* Bien sûr, s'il n'y a pas de sélection, cela n'a aucun effet !     */
  if (Selection==ENTIRE_IMAGE) {
    x1=0;
    x2=drawable->width;
    y1=0;
    y2=drawable->height;
  }

  /* Si on veut prévisualiser une sélection avec une surface qui l'entoure, */
  /* on doit l'agrandir un petit peu. Considérez ça comme une devinette.   */

  if (Selection==SELECTION_IN_CONTEXT) {
    x1=MAX(0,                x1-width/2.0);
    x2=MIN(drawable->width,  x2+width/2.0);
    y1=MAX(0,                y1-height/2.0);
    y2=MIN(drawable->height, y2+height/2.0);
  }

  /* Calcul de la largeur et de la hauteur de la surface à réduire. */
```

```
  width  = x2-x1;
  height = y2-y1;

  /* Les lignes ci-dessous déterminent la dimension qui sera le coté */
  /* le plus long. Cette idée est empruntée au plug-in Supernova.    */
  /* Je soupçonne que j'aurais pu y penser moi-même, mais la vérité  */
  /* doit être dite. Le plagiat pue !                                */

  if (width>height) {
    RW=LongerSize;
    RH=(float) height * (float) LongerSize/ (float) width;
  }
  else {
    RH=LongerSize;
    RW=(float)width * (float) LongerSize/ (float) height;
  }

  /* L'image entière est réduite dans une chaîne ! */

  tempRGB   = (guchar *) malloc(RW*RH*bytes);
  tempmask  = (guchar *) malloc(RW*RH);

  gimp_pixel_rgn_init (&srcPR, drawable, x1, y1, width, height, FALSE, FALSE);
  gimp_pixel_rgn_init (&srcMask, mask, x1, y1, width, height, FALSE, FALSE);

  /* Réservation pour sauver une ligne d'image et une ligne du masque */
  src_row       = (guchar *) malloc (width*bytes);
  src_mask_row  = (guchar *) malloc (width);

  for (i=0; i < RH; i++) {
    whichrow=(float)i*(float)height/(float)RH;
    gimp_pixel_rgn_get_row (&srcPR, src_row, x1, y1+whichrow, width);
    gimp_pixel_rgn_get_row (&srcMask, src_mask_row, x1, y1+whichrow, width);

    for (j=0; j < RW; j++) {
      whichcol=(float)j*(float)width/(float)RW;

      /* Pas de sélection = chaque point est complètement sélectionné ! */

      if (NoSelectionMade)
        tempmask[i*RW+j]=255;
      else
        tempmask[i*RW+j]=src_mask_row[whichcol];

      /* Ajout de la ligne à la longue chaîne qui contient maintenant */
      /* l'image !                                                    */

      tempRGB[i*RW*bytes+j*bytes+0]=src_row[whichcol*bytes+0];
      tempRGB[i*RW*bytes+j*bytes+1]=src_row[whichcol*bytes+1];
      tempRGB[i*RW*bytes+j*bytes+2]=src_row[whichcol*bytes+2];

      /* On s'accroche aussi à l'alpha */
      if (bytes==4)
        tempRGB[i*RW*bytes+j*bytes+3]=src_row[whichcol*bytes+3];
    }
  }
  temp->bpp=bytes;
  temp->width=RW;
  temp->height=RH;
  temp->rgb=tempRGB;
  temp->mask=tempmask;
  return temp;
}

La suite est une fonction de prévisualisation qui utilise le même type
```

```
<em/ReducedImage/ !  On remarque qu'elle utilise une fausse
transparence (au moyen de <em/fake_transparancy/ qui est défini comme
suit :

gint fake_transparency(gint i, gint j)
{
  if ( ((i%20)- 10) * ((j%20)- 10)>0   )
    return 64;
  else
    return 196;
}

Voici maintenant la fonction de prévisualisation«nbsp;:

void
my_preview_render_function(GtkWidget     *preview,
                           gint          changewhat,
                           gint          changewhich)
{
  gint Inten, bytes=drawable->bpp;
  gint i, j, k;
  float partial;
  gint RW=reduced->width;
  gint RH=reduced->height;
  guchar *row=malloc(bytes*RW);;


  for (i=0; i < RH; i++) {
    for (j=0; j < RW; j++) {

      row[j*3+0] = reduced->rgb[i*RW*bytes + j*bytes + 0];
      row[j*3+1] = reduced->rgb[i*RW*bytes + j*bytes + 1];
      row[j*3+2] = reduced->rgb[i*RW*bytes + j*bytes + 2];

      if (bytes==4)
        for (k=0; k<3; k++) {
          float transp=reduced->rgb[i*RW*bytes+j*bytes+3]/255.0;
          row[3*j+k]=transp*a[3*j+k]+(1-transp)*fake_transparency(i,j);
        }
    }
    gtk_preview_draw_row( GTK_PREVIEW(preview),row,0,i,RW);
  }

  free(a);
  gtk_widget_draw(preview,NULL);
  gdk_flush();
}

Fonctions applicables

guint           gtk_preview_get_type           (void);
/* Aucune idée */
void            gtk_preview_uninit             (void);
/* Aucune idée */
GtkWidget*      gtk_preview_new                (GtkPreviewType   type);
/* Décrite ci-dessous */
void            gtk_preview_size               (GtkPreview      *preview,
                                                gint             width,
                                                gint             height);
/* Permet de changer la taille d'une prévisualisation existante */
/* Apparamment, il y a un bug dans GTK qui rend ce traitement   */
/* hasardeux. Une méthode pour corriger ce problème consiste à  */
/* changer manuellement la taille de la fenêtre contenant la    */
/* prévisualisation après avoir changé la taille de la          */
/* prévisualisation.                                            */
```

```
void            gtk_preview_put               (GtkPreview      *preview,
                                               GdkWindow       *window,
                                               GdkGC           *gc,
                                               gint             srcx,
                                               gint             srcy,
                                               gint             destx,
                                               gint             desty,
                                               gint             width,
                                               gint             height);
/* Aucune idée */

void            gtk_preview_put_row           (GtkPreview      *preview,
                                               guchar          *src,
                                               guchar          *dest,
                                               gint             x,
                                               gint             y,
                                               gint             w);
/* Aucune idée */

void            gtk_preview_draw_row          (GtkPreview      *preview,
                                               guchar          *data,
                                               gint             x,
                                               gint             y,
                                               gint             w);
/* Décrite dans le texte */

void            gtk_preview_set_expand        (GtkPreview      *preview,
                                               gint             expand);
/* Aucune idée */

/* Aucune piste pour celles qui suivent mais devrait être */
/* un standard pour la plupart des widgets.               */

void            gtk_preview_set_gamma         (double           gamma);
void            gtk_preview_set_color_cube    (guint            nred_shades,
                                               guint            ngreen_shades,
                                               guint            nblue_shades,
                                               guint            ngray_shades);
void            gtk_preview_set_install_cmap  (gint             install_cmap);
void            gtk_preview_set_reserved      (gint             nreserved);
GdkVisual*      gtk_preview_get_visual        (void);
GdkColormap*    gtk_preview_get_cmap          (void);
GtkPreviewInfo* gtk_preview_get_info          (void);

That's all, folks!
```

12.7 Courbes

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

13. Widget EventBox

Il n'est disponible que dans *gtk+970916.tar.gz* et les distributions ultérieures.

Certains widgets GTK n'ont pas de fenêtre X associée, il se dessinent donc sur leurs parents. À

cause de cela, ils ne peuvent recevoir d'événements et, s'ils ont une taille incorrecte, ils ne peuvent pas se mettre en place correctement : on peut alors avoir des surimpressions douteuses, etc. Si vous avez besoin de ces widgets, *EventBox* est fait pour vous.

Au premier abord, le widget *EventBox* peut apparaître comme totalement dénué d'intérêt. Il ne dessine rien à l'écran et ne répond à aucun évenement. Cependant, il joue un rôle − il fournit une fenêtre X pour son widget fils. Ceci est important car de nombreux widgets GTK n'ont pas de fenêtre X associée. Ne pas avoir de fenêtre permet d'économiser de la mémoire mais a aussi quelques inconvénients. Un widget sans fenêtre ne peut recevoir d'événement, et ne réalise aucune mise en place de ce qu'il contient. Bien que le nom « EventBox » insiste sur la fonction de gestion d'événement, le widget peut aussi être utilisé pour la mise en place (et plus... voir l'exemple ci−dessous).

Pour créer un widget EventBox, on utilise :

```
GtkWidget* gtk_event_box_new (void);
```

Un widget fils peut alors être ajouté à cet *EventBox* :

```
gtk_container_add (GTK_CONTAINER(event_box), widget);
```

L'exemple suivant montre l'utilisation d'un *EventBox* − un label est créé et mis en place sur une petite boîte, et configuré pour qu'un clic souris sur le label provoque la fin du programme.

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

int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *event_box;
    GtkWidget *label;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Event Box");

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_exit), NULL);

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* Création d'un EventBox et ajout de celui−ci dans la fenêtre. */

    event_box = gtk_event_box_new ();
    gtk_container_add (GTK_CONTAINER(window), event_box);
    gtk_widget_show (event_box);

    /* Création d'un long label */

    label = gtk_label_new ("Cliquez ici pour quitter, quitter, quitter, quitter, qu
    gtk_container_add (GTK_CONTAINER (event_box), label);
    gtk_widget_show (label);

    /* Placement serré. */

    gtk_widget_set_usize (label, 110, 20);

    /* Attachement d'une action à celui−ci. */
```

```
    gtk_widget_set_events (event_box, GDK_BUTTON_PRESS_MASK);
    gtk_signal_connect (GTK_OBJECT(event_box), "button_press_event",
                        GTK_SIGNAL_FUNC (gtk_exit), NULL);

    /* Encore une fois, vous avez besoin d'une fenêtre X pour... */

    gtk_widget_realize (event_box);
    gdk_window_set_cursor (event_box->window, gdk_cursor_new (GDK_HAND1));

    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

14. Configuration des attributs de widget

Cette section décrit les fonctions opérant sur les widgets. Elles peuvent être utilisées pour configurer le style, l'espacement, la taille, etc.

(Je devrais peut être faire une section uniquement consacrée aux raccourcis clavier.)

```
void       gtk_widget_install_accelerator (GtkWidget           *widget,
                                           GtkAcceleratorTable *table,
                                           gchar               *signal_name,
                                           gchar                key,
                                           guint8               modifiers);

void       gtk_widget_remove_accelerator  (GtkWidget           *widget,
                                           GtkAcceleratorTable *table,
                                           gchar               *signal_name);

void       gtk_widget_activate            (GtkWidget           *widget);

void       gtk_widget_set_name            (GtkWidget           *widget,
                                           gchar               *name);
gchar*     gtk_widget_get_name            (GtkWidget           *widget);

void       gtk_widget_set_sensitive       (GtkWidget           *widget,
                                           gint                 sensitive);

void       gtk_widget_set_style           (GtkWidget           *widget,
                                           GtkStyle            *style);

GtkStyle*    gtk_widget_get_style     (GtkWidget *widget);

GtkStyle*    gtk_widget_get_default_style    (void);

void       gtk_widget_set_uposition       (GtkWidget           *widget,
                                           gint                 x,
                                           gint                 y);
void       gtk_widget_set_usize           (GtkWidget           *widget,
                                           gint                 width,
```

```
                                                gint                 height);

void        gtk_widget_grab_focus           (GtkWidget           *widget);

void        gtk_widget_show                 (GtkWidget           *widget);

void        gtk_widget_hide                 (GtkWidget           *widget);
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

15. Temporisations, fonctions d'E/S et d'attente

15.1 Temporisations

Vous pouvez vous demander comment faire pour que GTK fasse quelque chose d'utile lorsqu'il est dans *gtk_main*. En fait, on a plusieurs options. L'utilisation des fonctions suivantes permet de créer une temporisation qui sera appelée tous les *interval* millisecondes.

```
gint gtk_timeout_add (guint32 interval,
                      GtkFunction function,
                      gpointer data);
```

Le premier paramètre est le nombre de millisecondes entre les appels à notre fonction. Le deuxième est la fonction à appeler et le troisième est la donnée passée à cette fonction de rappel. La valeur retournée est un « marqueur » de type entier qui pourra être utilisé pour arrêter la temporisation en appelant :

```
void gtk_timeout_remove (gint tag);
```

On peut aussi stopper la fonction de temporisation en faisant retourner zéro ou FALSE à notre fonction de rappel. Évidemment, cela veut dire que si vous voulez que votre fonction continue à être appelée, elle doit retourner une valeur non nulle, ou TRUE.

La déclaration de votre fonction de rappel doit ressembler à ça :

```
gint timeout_callback (gpointer data);
```

15.2 Surveillance des E/S

Une autre caractéristique intéressante du GTK est la possibilité de vérifier les données d'un descripteur de fichier (celles retournées par *open*(2) ou *socket*(2)). C'est particulièrement pratique pour les applications réseau. La fonction suivante permet cette vérification :

```
gint gdk_input_add (gint source,
                    GdkInputCondition condition,
                    GdkInputFunction  function,
                    gpointer data);
```

Le premier paramètre est le descripteur de fichier que l'on veut étudier, le second spécifie ce qu'on veut que le GDK recherche. Cela peut être :

GDK_INPUT_READ − Appel *function* lorsqu'il y a une donnée prête à être lue dans le descripteur de fichier.

GDK_INPUT_WRITE − Appel de *function* lorsque le descripteur de fichier est prêt pour une écriture.

Je suis sûr que vous vous doutez, maintenant, que le troisième paramètre est la fonction que l'on veut appeler lorsque les conditions ci−dessus sont satisfaites. Le dernier paramètre est la donnée à passer à cette fonction.

La valeur retournée est un marqueur qui pourra être utilisé pour dire au GDK de cesser de surveiller ce descripteur à l'aide de la fonction :

```
void gdk_input_remove (gint tag);
```

La fonction de rappel doit être déclarée de la façon suivante :

```
void input_callback (gpointer data, gint source,
                     GdkInputCondition condition);
```

15.3 Fonctions d'attente

Que se passe−t'il si vous avez une fonction qui doit être appelée lorsque rien d'autre ne se passe ? On utilise la fonction suivante qui force GTK à appeler *function* lorsqu'on est en phase d'inaction ;

```
gint gtk_idle_add (GtkFunction function,
                   gpointer data);

void gtk_idle_remove (gint tag);
```

Je n'expliquerai pas la signification des paramètres car ils ressemblent beaucoup à ceux déjà vus ci−dessus. La fonction pointée par le premier paramètre de *gtk_idle_add()* sera appelée à chaque occasion. Comme pour les autres, retourner FALSE empêchera la fonction d'attente d'être appelée.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

16. Gestion des sélections

16.1 Introduction

Un type de communication inter−processus gérée par GTK est les *sélections*. Une sélection identifie un morceau de données, par exemple une portion de texte sélectionnée par l'utilisateur avec la souris. Seule une application sur un écran (le *propriétaire*) peut posséder une sélection particulière à un moment donné, ainsi lorsqu'une sélection est réclamée par une application, le propriétaire précédent doit indiquer à l'utilisateur que la sélection a été abandonnée. Les autres applications peuvent demander le contenu d'une sélection sous différentes formes appelées *cibles*. Il peut y avoir un nombre quelconque de sélections, mais la plupart des applications X

n'en gèrent qu'une, la *sélection primaire*.

Dans la plupart des cas, une application GTK n'a pas besoin de gérer elle−même les sélections. Les widgets standards, comme le widget Entrée de texte, possèdent déjà la capacité de réclamer la sélection lorsqu'il le faut (par exemple, lorsque l'utilisateur glisse au dessus d'un texte) et de récupérer le contenu de la sélection détenue par un autre widget ou une autre application (par exemple, lorsque l'utilisateur clique avec le deuxième bouton de la souris). Cependant, il peut il y avoir des cas dans lesquels vous voulez donner aux autres widgets la possibilité de fournir la sélection, ou vous désirez récupérer des cibles non supportées par défaut.

Un concept fondamental dans la compréhension du fonctionnement des sélections est celui d'*atome*. Un atome est un entier qui définit de façon unique une chaîne (sur un affichage particulier). Certains atomes sont prédéfinis par le serveur X et, dans certains cas, des constantes définies dans *gtk.h* correspondent à ces atomes. Par exemple, la constante GDK_PRIMARY_SELECTION correspond à la chaîne "PRIMARY". Dans d'autres cas, on doit utiliser les fonctions *gdk_atom_intern()*, pour obtenir l'atome correspondant à une chaîne, et *gdk_atom_name()*, pour obtenir le nom d'un atome. Les sélections et les cibles sont identifiés par des atomes.

16.2 Récupération de la sélection

La récupération de la sélection est un processus asynchrone. Pour démarrer le processus, on appelle :

```
gint gtk_selection_convert   (GtkWidget           *widget,
                              GdkAtom              selection,
                              GdkAtom              target,
                              guint32              time)
```

Cela *convertit* la sélection dans la forme spécifiée par *target*. Si tout est possible, le paramètre *time* sera le moment de l'événement qui a déclenché la sélection. Ceci aide à s'assurer que les événements arrivent dans l'ordre où l'utilisateur les a demandé. Cependant, si cela n'est pas possible (par exemple, lorsque la conversion a été déclenchée par un signal "clicked"), alors on peut utiliser la macro GDK_CURRENT_TIME.

Quand le propriétaire de la sélection répond à la requête, un signal "selection_received" est envoyé à notre application. Le gestionnaire de ce signal reçoit un pointeur vers une structure `GtkSelectionData` définie ainsi :

```
struct _GtkSelectionData
{
  GdkAtom selection;
  GdkAtom target;
  GdkAtom type;
  gint    format;
  guchar *data;
  gint    length;
};
```

selection et *target* sont les valeurs que l'on a donné dans notre appel *gtk_selection_convert()*. *type* est un atome qui identifie le type de données retourné par le propriétaire de la sélection. Quelques valeurs possibles sont : "STRING", une chaîne de caractères latin−1, "ATOM", une série d'atomes, "INTEGER", un entier, etc. La plupart des cibles ne peuvent retourner qu'un type. *format* donne la longueur des unités (les caractères, par exemple) en bits. Habituellement, on ne se préoccupe pas de cela lorsqu'on reçoit des données. *data* est un pointeur vers la donnée retournée et *length* donne la longueur en octets de la donnée retournée. Si *length* est négative,

cela indique qu'une erreur est survenue et que la sélection ne peut être récupérée. Ceci peut arriver si aucune application n'est propriétaire de la sélection, ou si vous avez demandé une cible que l'application ne sait pas gérer. Le tampon est garanti d'être un octet plus long que *length* ; l'octet supplémentaire sera toujours zéro, et il n'est donc pas nécessaire de faire une copie de chaîne simplement pour qu'elle soit terminée par zéro (comme doivent l'être toutes les chaînes C).

Dans l'exemple qui suit, on récupère la cible spéciale "TARGETS", qui est une liste de toutes les cibles en lesquelles la sélection peut être convertie.

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

void selection_received (GtkWidget *widget,
                         GtkSelectionData *selection_data,
                         gpointer data);

/* Gestionnaire de signal invoqué lorsque l'utilisateur clique sur
 * le bouton « Obtenir les cibles ». */

void get_targets (GtkWidget *widget, gpointer data)
{
  static GdkAtom targets_atom = GDK_NONE;

  /* Obtention de l'atome correspondant à la chaîne "TARGETS" */

  if (targets_atom == GDK_NONE)
    targets_atom = gdk_atom_intern ("TARGETS", FALSE);

  /* Demande de la cible "TARGETS" pour la sélection primaire */

  gtk_selection_convert (widget, GDK_SELECTION_PRIMARY, targets_atom,
                         GDK_CURRENT_TIME);
}

/* Gestionnaire de signal appelé quand le propriétaire des sélections
 * retourne la donnée. */

void selection_received (GtkWidget *widget, GtkSelectionData *selection_data,
                    gpointer data)
{
  GdkAtom *atoms;
  GList *item_list;
  int i;

  /* **** IMPORTANT **** On vérifie si la récupération s'est bien passée. */

  if (selection_data->length < 0)
    {
      g_print ("Selection retrieval failed\n");
      return;
    }

  /* On s'assure que l'on a obtenu la donnée sous la forme attendue. */

  if (selection_data->type != GDK_SELECTION_TYPE_ATOM)
    {
      g_print ("La sélection \"TARGETS\" n'a pas été retournée sous la forme d'atom
      return;
    }

  /* Affichage des atomes reçus. */

  atoms = (GdkAtom *)selection_data->data;
```

```
  item_list = NULL;
  for (i=0; i<selection_data->length/sizeof(GdkAtom); i++)
    {
      char *name;
      name = gdk_atom_name (atoms[i]);
      if (name != NULL)
        g_print ("%s\n",name);
      else
        g_print ("(atome incorrect)\n");
    }

  return;
}

int
main (int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *button;

  gtk_init (&argc, &argv);

  /* Création de la fenêtre de l'application. */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Sélections");
  gtk_container_border_width (GTK_CONTAINER (window), 10);

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  /* Création d'un bouton pour obtenir les cibles */

  button = gtk_button_new_with_label ("Obtenir les cibles");
  gtk_container_add (GTK_CONTAINER (window), button);

  gtk_signal_connect (GTK_OBJECT(button), "clicked",
                      GTK_SIGNAL_FUNC (get_targets), NULL);
  gtk_signal_connect (GTK_OBJECT(button), "selection_received",
                      GTK_SIGNAL_FUNC (selection_received), NULL);

  gtk_widget_show (button);
  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

16.3 Fournir la sélection

Fournir la sélection est un peu plus compliqué. On doit enregistrer les gestionnaires qui seront appelés lorsque notre sélection est demandée. Pour chaque paire sélection/cible que l'on gèrera, on fera un appel à :

```
void gtk_selection_add_handler (GtkWidget           *widget,
                                GdkAtom              selection,
                                GdkAtom              target,
                                GtkSelectionFunction function,
                                GtkRemoveFunction    remove_func,
                                gpointer             data);
```

widget, *selection* et *target* identifient les requêtes que ce gestionnaire gèrera. S'il ne vaut pas NULL, *remove_func* sera appelé lorsque le gestionnaire de signal est supprimé. Ceci est utile, par exemple, pour des langages interprétés qui doivent garder une trace du nombre de références à *data*.

La fonction de rappel *function* doit avoir la signature suivante :

```
typedef void (*GtkSelectionFunction) (GtkWidget *widget,
                                      GtkSelectionData *selection_data,
                                      gpointer data);
```

Le *GtkSelectionData* est le même qu'au dessus, mais, cette fois, nous sommes responsables de l'initialisation de ses champs *type*, *format*, *data*, et *length*. (Le champ *format* est important ici − le serveur X l'utilise pour savoir si la donnée doit être échangée par octet ou non. Habituellement, ce sera 8 (un caractère), ou 32 (un entier)). Cette initialisation est faite en utilisant l'appel :

```
void gtk_selection_data_set (GtkSelectionData *selection_data,
                             GdkAtom           type,
                             gint              format,
                             guchar           *data,
                             gint              length);
```

Cette fonction s'occupe de faire une copie correcte des données afin que l'on n'ait pas à se soucier du reste. (On ne doit pas remplir ces champs à la main).

Lorsque cela est demandé par l'utilisateur, on réclame la possession de la sélection en appelant :

```
gint gtk_selection_owner_set (GtkWidget           *widget,
                              GdkAtom              selection,
                              guint32              time);
```

Si une autre application réclame la possession de la sélection, on recevra un "selection_clear_event".

Comme exemple de fourniture de sélection, l'exemple suivant ajoute une fonctionnalité de sélection à un bouton commutateur. Lorsque ce bouton est appuyé, le programme réclame la sélection primaire. La seule cible supportée (à part certaines cibles fournies par GTK lui−même, comme « TARGETS ») est « STRING ». Lorsque celle−ci est demandée, on retourne une représentation de l'heure sous forme de chaîne.

```
#include <gtk/gtk.h>
#include <time.h>

/* Fonction de rappel appelée lorsque l'utilisateur commute la sélection. */

void selection_toggled (GtkWidget *widget, gint *have_selection)
{
  if (GTK_TOGGLE_BUTTON(widget)->active)
    {
      *have_selection = gtk_selection_owner_set (widget,
                                                 GDK_SELECTION_PRIMARY,
                                                 GDK_CURRENT_TIME);
      /* Si la demande de sélection échoue, on remet le bouton en position sortie.

      if (!*have_selection)
        gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON(widget), FALSE);
    }
  else
```

```
    {
      if (*have_selection)
        {
          /* Avant de nettoyer la selection en mettant son propriétaire à NULL,
           * on vérifie que nous sommes bien son propriétaire actuel. */

          if (gdk_selection_owner_get (GDK_SELECTION_PRIMARY) == widget->window)
            gtk_selection_owner_set (NULL, GDK_SELECTION_PRIMARY,
                                     GDK_CURRENT_TIME);
          *have_selection = FALSE;
        }
    }
}

/* Appelée lorsqu'une autre application demande la sélection. */

gint selection_clear (GtkWidget *widget, GdkEventSelection *event,
                      gint *have_selection)
{
  *have_selection = FALSE;
  gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON(widget), FALSE);

  return TRUE;
}

/* Fournit l'heure comme sélection. */

void selection_handle (GtkWidget *widget,
                       GtkSelectionData *selection_data,
                       gpointer data)
{
  gchar *timestr;
  time_t current_time;

  current_time = time (NULL);
  timestr = asctime (localtime(&current_time));

  /* Lorsqu'on retourne une chaîne, elle ne doit pas se terminer par
   * 0, ce sera fait pour nous. */

  gtk_selection_data_set (selection_data, GDK_SELECTION_TYPE_STRING,
                          8, timestr, strlen(timestr));
}

int main (int argc, char *argv[])
{
  GtkWidget *window;

  GtkWidget *selection_button;

  static int have_selection = FALSE;

  gtk_init (&argc, &argv);

  /* Création de la fenêtre principale. */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Event Box");
  gtk_container_border_width (GTK_CONTAINER (window), 10);

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  /* Création d'un bouton commutateur pour qu'il agisse comme une sélection. */
```

```
  selection_button = gtk_toggle_button_new_with_label ("Demande de sélection");
  gtk_container_add (GTK_CONTAINER (window), selection_button);
  gtk_widget_show (selection_button);

  gtk_signal_connect (GTK_OBJECT(selection_button), "toggled",
                      GTK_SIGNAL_FUNC (selection_toggled), &have_selection);
  gtk_signal_connect (GTK_OBJECT(selection_button), "selection_clear_event",
                      GTK_SIGNAL_FUNC (selection_clear), &have_selection);

  gtk_selection_add_handler (selection_button, GDK_SELECTION_PRIMARY,
                             GDK_SELECTION_TYPE_STRING,
                             selection_handle, NULL, NULL);

  gtk_widget_show (selection_button);
  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

17. glib

La *glib* fournit de nombreuses fonctions et définitions utiles, prêtes à être utilisées lorsqu'on crée des applications GDK et GTK. Je les énumèrerais toutes avec une brève explication. Beaucoup sont des répliques des fonctions standards de la *libc*, et je ne les détaillerais donc pas trop. Ceci doit surtout servir de référence afin de savoir ce qui est disponible pour être utilisé.

17.1 Définitions

Les définitions pour les bornes de la plupart des types standards sont :

```
G_MINFLOAT
G_MAXFLOAT
G_MINDOUBLE
G_MAXDOUBLE
G_MINSHORT
G_MAXSHORT
G_MININT
G_MAXINT
G_MINLONG
G_MAXLONG
```

Voici aussi les redéfinitions de types. Celles qui ne sont pas spécifiées sont configurées dynamiquement selon l'architecture. Évitez surtout de compter sur la taille d'un pointeur si vous voulez un programme portable ! Un pointeur sur un Alpha fait 8 octets, mais il en fait 4 sur un Intel.

```
char   gchar;
short  gshort;
long   glong;
int    gint;
```

```
char   gboolean;

unsigned char   guchar;
unsigned short  gushort;
unsigned long   gulong;
unsigned int    guint;

float   gfloat;
double  gdouble;
long double gldouble;

void* gpointer;

gint8
guint8
gint16
guint16
gint32
guint32
```

17.2 Listes doublement chaînées

Les fonctions suivantes servent à créer, gérer et détruire des listes doublement chaînées. Je suppose que vous savez ce qu'est une liste chaînée car leur explication n'entre pas dans le cadre de ce document. Bien sûr, il n'y a pas besoin de les connaître pour une utilisation générale de GTK, mais c'est bien de savoir comment elles fonctionnent.

```
GList* g_list_alloc       (void);

void   g_list_free        (GList     *list);

void   g_list_free_1      (GList     *list);

GList* g_list_append      (GList     *list,
                           gpointer   data);

GList* g_list_prepend     (GList     *list,
                           gpointer   data);

GList* g_list_insert      (GList     *list,
                           gpointer   data,
                           gint       position);

GList* g_list_remove      (GList     *list,
                           gpointer   data);

GList* g_list_remove_link (GList     *list,
                           GList     *link);

GList* g_list_reverse     (GList     *list);

GList* g_list_nth         (GList     *list,
                           gint       n);

GList* g_list_find        (GList     *list,
                           gpointer   data);

GList* g_list_last        (GList     *list);

GList* g_list_first       (GList     *list);

gint   g_list_length      (GList     *list);
```

```
void   g_list_foreach     (GList     *list,
                           GFunc      func,
                           gpointer   user_data);
```

17.3 Listes simplement chaînées

La plupart des fonctions pour les listes simplement chaînées ci−dessous sont identiques à celles vues plus haut. Voici une liste complète :

```
GSList* g_slist_alloc       (void);

void    g_slist_free        (GSList   *list);

void    g_slist_free_1      (GSList   *list);

GSList* g_slist_append      (GSList   *list,
                             gpointer  data);

GSList* g_slist_prepend     (GSList   *list,
                             gpointer  data);

GSList* g_slist_insert      (GSList   *list,
                             gpointer  data,
                             gint      position);

GSList* g_slist_remove      (GSList   *list,
                             gpointer  data);

GSList* g_slist_remove_link (GSList   *list,
                             GSList   *link);

GSList* g_slist_reverse     (GSList   *list);

GSList* g_slist_nth         (GSList   *list,
                             gint      n);

GSList* g_slist_find        (GSList   *list,
                             gpointer  data);

GSList* g_slist_last        (GSList   *list);

gint    g_slist_length      (GSList   *list);

void    g_slist_foreach     (GSList   *list,
                             GFunc     func,
                             gpointer  user_data);
```

17.4 Gestion de la mémoire

```
gpointer g_malloc      (gulong    size);
```

Remplace *malloc()*. On n'a pas besoin de vérifier la valeur de retour car cela est fait pour nous dans cette fonction.

```
gpointer g_malloc0     (gulong    size);
```

Identique à la précédente, mais initialise la mémoire à zéro avant de retourner un pointeur vers la zone réservée.

```
gpointer g_realloc     (gpointer  mem,
```

```
                                gulong    size);
```

Réalloue *size* octets de mémoire à partir de *mem*. Évidemment, la mémoire doit avoir été allouée auparavant.

```
void     g_free        (gpointer  mem);
```

Libère la mémoire. Facile.

```
void     g_mem_profile (void);
```

Produit un profil de la mémoire utilisée, mais requiert l'ajout de *#define MEM_PROFILE* au début de *glib/gmem.c*, de refaire un *make* et un *make install*.

```
void     g_mem_check   (gpointer  mem);
```

Vérifie qu'un emplacement mémoire est valide. Nécessite que l'on ajoute *#define MEM_CHECK* au début de *gmem.c* que l'on refasse un *make* et un *make install*.

17.5 Timers

Fonctions des timers...

```
GTimer* g_timer_new     (void);

void    g_timer_destroy (GTimer  *timer);

void    g_timer_start   (GTimer  *timer);

void    g_timer_stop    (GTimer  *timer);

void    g_timer_reset   (GTimer  *timer);

gdouble g_timer_elapsed (GTimer  *timer,
                         gulong  *microseconds);
```

17.6 Gestion des chaînes

Un ensemble complet de fonction de gestion des chaînes. Elles semblent toutes très intéressantes et sont sûrement meilleures, à bien des égards, que les fonctions C standards, mais elle nécessitent de la documentation.

```
GString* g_string_new       (gchar   *init);
void     g_string_free      (GString *string,
                             gint     free_segment);

GString* g_string_assign    (GString *lval,
                             gchar   *rval);

GString* g_string_truncate  (GString *string,
                             gint     len);

GString* g_string_append    (GString *string,
                             gchar   *val);

GString* g_string_append_c  (GString *string,
                             gchar    c);

GString* g_string_prepend   (GString *string,
```

```
                                gchar   *val);

GString* g_string_prepend_c (GString *string,
                                gchar    c);

void     g_string_sprintf   (GString *string,
                                gchar   *fmt,
                                ...);

void     g_string_sprintfa  (GString *string,
                                gchar   *fmt,
                                ...);
```

17.7 Utilitaires et fonctions d'erreurs

```
gchar* g_strdup    (const gchar *str);
```

Remplace la fonction *strdup*. Elle copie le contenu de la chaîne d'origine dans la mémoire venant d'être allouée et retourne un pointeur sur cette zone.

```
gchar* g_strerror  (gint errnum);
```

Je recommande de l'utiliser pour tous les messages d'erreur. Elle est beaucoup plus propre et plus portable que *perror()* ou les autres. La sortie est habituellement de la forme :

```
nom du programme:fonction qui a échoué:fichier ou autre descripteur:strerror
```

Voici un exemple d'appel utilisé dans le programme « Bonjour tout le monde ! » :

```
g_print("bonjour_monde:open:%s:%s\n", filename, g_strerror(errno));

void g_error   (gchar *format, ...);
```

Affiche un message d'erreur. Le format est comme *printf*, mais il ajoute « ** ERROR **: » au début du message et sort du programme. À n'utiliser que pour les erreurs fatales.

```
void g_warning (gchar *format, ...);
```

Comme au dessus, mais ajoute « ** WARNING **: », et ne termine pas le programme.

```
void g_message (gchar *format, ...);
```

Affiche « message: » avant la chaîne passée en paramètre.

```
void g_print   (gchar *format, ...);
```

Remplace *printf()*.

Enfin la dernière fonction :

```
gchar* g_strsignal (gint signum);
```

Affiche le nom du signal système Unix correspondant au numéro de signal. Utile pour les fonctions génériques de gestion de signaux.

Tout ce qui est ci−dessus est plus ou moins volé à *glib.h*. Si quelqu'un s'occupe de documenter une fonction, qu'il m'envoit un courrier !

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

18. Fichiers rc de GTK

GTK a sa propre méthode pour gérer les configurations par défaut des applications, en utilisant des fichiers `rc`. Ceux−ci peuvent être utilisés pour configurer les couleurs de presque tous les widgets, et pour mettre des pixmaps sur le fond de certains widgets.

18.1 Fonctions pour les fichiers rc

Au démarrage de votre application, ajoutez un appel à :

```
void gtk_rc_parse (char *filename);
```

en lui passant le nom de votre fichier rc. Ceci forcera GTK à analyser ce fichier et à utiliser les configurations de styles pour les types de widgets qui y sont définis.

Si vous voulez avoir un ensemble particulier de widgets qui prenne le pas sur le style des autres, ou une autre division logique de widgets, utilisez un appel à :

```
void gtk_widget_set_name (GtkWidget *widget,
                          gchar *name);
```

En lui passant comme premier paramètre le widget que vous avez créé, et le nom que vous voulez lui donner comme second paramètre. Ceci vous permettra de changer les attributs de ce widget par son nom dans le fichier rc.

Si vous utilisez un appel comme celui−ci :

```
button = gtk_button_new_with_label ("Bouton Spécial");
gtk_widget_set_name (button, "bouton special");
```

Ce bouton s'appelle « bouton special » et peut être accédé par son nom dans le fichier rc en tant que « bouton special.GtkButton ». [<−−− Vérifiez !]

Le fichier rc ci−dessous configure les propriétés de la fenêtre principale et fait hériter tous les fils de celle−ci du style décrit par « bouton_principal ». Le code utilisé dans l'application est :

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_set_name (window, "fenetre principale");
```

Et le style est défini dans le fichier rc avec :

```
widget "fenetre principale.*GtkButton*" style "bouton_principal"
```

Ce qui configure tous les widgets *GtkButton* de « fenêtre principale » avec le style « bouton_principal » défini dans le fichier rc.

Ainsi que vous pouvez le voir, il s'agit d'un système puissant et flexible. Utilisez votre imagination pour en tirer le meilleur.

18.2 Format des fichiers rc de GTK

Le format du fichier GTK est illustré dans l'exemple suivant. Il s'agit du fichier *testgtkrc* de la distribution GTK mais j'ai ajouté quelques commentaires et autres choses. Vous pouvez inclure cette explication à votre application pour permettre à l'utilisateur de régler finement son application.

Il y a plusieurs directives pour changer les attributs d'un widget.

- fg − configure la couleur de premier plan d'un widget.
- bg − configure la couleur d'arrière plan d'un widget.
- bg_pixmap − configure l'arrière plan d'un widget avec un pixmap.
- font − configure la fonte à utiliser pour un widget.

De plus, un widget peut se trouver dans différents états et l'on peut configurer des couleurs, pixmaps et fontes différentes pour chacun d'eux. Ces états sont :

- NORMAL − L'état normal d'un widget, sans la souris au dessus de lui, non pressé, etc.
- PRELIGHT − Lorsque la souris se trouve au dessus du widget, les couleurs définies pour cet état sont actives.
- ACTIVE − Lorsque le widget est pressé ou cliqué, il devient actif et les attributs associés à cet état sont appliqués.
- INSENSITIVE − Quand un widget est configuré pour être insensible et qu'il ne peut être activé, il prend ces attributs.
- SELECTED − Lorsqu'un objet est choisi, il prend ces attributs.

Lorsqu'on utilise les mots−clés « *fg* » et « *bg* » pour configurer les couleurs des widgets, le format est :

```
fg[<STATE>] = { Red, Green, Blue }
```

Où STATE est l'un des états vus plus haut (PRELIGHT, ACTIVE etc), et où *Red*, *Green* et *Blue* sont des valeurs comprises entre 0 et 1.0. { 1.0, 1.0, 1.0 } représente la couleur blanche. Ces valeurs doivent être de type réel ou elles seront considérées comme valant 0, ainsi un simple « 1 » ne marchera pas, il faut mettre « 1.0 ». Un « 0 » simple convient car ce n'est pas un problème s'il n'est pas reconnu puisque toutes les valeurs non reconnues sont mises à 0.

bg_pixmap est très similaire, sauf que les couleurs sont remplacées par un nom de fichier.

pixmap_path est une liste de chemins séparés par des « : ». Ces chemins seront parcourus pour chaque pixmap que l'on spécifie.

La directive *font* est simplement :

```
font = "<font name>"
```

Où la seule partie difficile est d'arriver à comprendre la chaîne contenant le nom de la fonte. L'utilisation de *xfontsel* ou d'un autre utilitaire semblable peut aider.

« *widget_class* » configure le style d'une classe de widgets. Ces classes sont listées dans la section sur la hiérarchie des widgets.

La directive « *widget* » configure un ensemble spécifique de widgets selon un style donné,

annulant tout style de configuration pour la classe de widget donnée. Ces widgets sont enregistrés dans l'application en utilisant l'appel *gtk_widget_set_name()*. Ceci vous permet de spécifier les attributs d'un widget, widget par widget, au lieu de configurer les attributs d'une classe entière de widgets. Je vous demande instamment de documenter tous ces widgets spéciaux pour que les utilisateurs puisse les adapter à leurs besoins.

Lorsque le mot−clé « *parent* » est utilisé comme attribut, le widget prendra les attributs de son parent dans l'application.

Lorsqu'on définit un style, on peut assigner les attributs d'un style déjà défini à ce nouveau style.

```
style "bouton_principal" = "button"
{
  font = "−adobe−helvetica−medium−r−normal−−*−100−*−*−*−*−*−*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

Cet exemple prend le style "button" et crée un nouveau style "bouton_principal"en changeant simplement la fonte et la couleur de fond pour l'état PRELIGHT.

Bien sûr, un bon nombre de ces attributs ne s'applique pas à tous les widgets. C'est une question de bon sens. Tout ce qui peut s'appliquer s'applique.

18.3 Exemple de fichier rc

```
# pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."
#
pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"
#
# style <name> [= <name>]
# {
#   <option>
# }
#
# widget <widget_set> style <style_name>
# widget_class <widget_class_set> style <style_name>


# Voici une liste des états possibles. Remarquez que certains ne s'appliquent
# pas à certains widgets.
#
# NORMAL − L'état normal d'un widget, sans la souris au dessus de lui,
# non pressé, etc.
#
# PRELIGHT − Lorsque la souris se trouve au dessus du widget, les couleurs
# définies pour cet état sont actives.
#
# ACTIVE − Lorsque le widget est pressé ou cliqué, il devient actif et les
# attributs associés à cet état sont appliqués.
#
# INSENSITIVE − Quand un widget est configuré pour être insensible, et qu'il
# ne peut être activé, il prend ces attributs.
#
# SELECTED − Lorsqu'un objet est choisi, il prend ces attributs.
#
# Avec ces états, on peut configurer les attributs des widgets dans chacun
# de ces états en utilisant les directives suivantes.
#
# fg − configure la couleur de premier plan d'un widget.
# bg − configure la couleur d'arrière plan d'un widget.
# bg_pixmap − configure l'arrière plan d'un widget avec un pixmap.
```

```
# font − configure la fonte à utiliser pour un widget.

# Configuration d'un style appelé "button". Le nom n'est pas important
# car il est assigné aux widgets réels à la fin du fichier.

style "window"
{
  #Configure l'espace autour de la fenêtre avec le pixmap spécifié.
  #bg_pixmap[<STATE>] = "<pixmap filename>"
  bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
{
  #Configure la couleur de premier plan (celle de la fonte) à rouge
  #lorsqu'on est dans l'état "NORMAL".

  fg[NORMAL] = { 1.0, 0, 0 }

  #Configure le pixmap d'arrière plan de ce widget à celui de son parent.
  bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
  # Voici tous les états possibles pour un bouton. Le seul qui ne peut
  # s'appliquer est l'état SELECTED.

  fg[PRELIGHT] = { 0, 1.0, 1.0 }
  bg[PRELIGHT] = { 0, 0, 1.0 }
  bg[ACTIVE] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 0, 1.0, 0 }
  bg[NORMAL] = { 1.0, 1.0, 0 }
  fg[NORMAL] = { .99, 0, .99 }
  bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
  fg[INSENSITIVE] = { 1.0, 0, 1.0 }
}

# Dans cet exemple, on hérite des attributs du style "button" puis on
# écrase la fonte et la couleur de fond pour créer un nouveau style
# "main_button".

style "main_button" = "button"
{
  font = "−adobe−helvetica−medium−r−normal−−*−100−*−*−*−*−*−*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
  fg[NORMAL] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 1.0, 0, 0 }

  # Configure le pixmap de fond du toggle_button à celui de son widget
  # parent (comme défini dans l'application).
  bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
  bg_pixmap[NORMAL] = "marble.xpm"
  fg[NORMAL] = { 1.0, 1.0, 1.0 }
}

style "ruler"
```

```
{
  font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}

# pixmap_path "~/.pixmaps"

# Configuration des types de widget pour utiliser les styles définis
# plus haut.
# Les types de widget sont listés dans la hiérarchie des classes, mais
# peut probablement être listée dans ce document pour que l'utilisateur
# puisse s'y référer.

widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"

# Configure tous les boutons fils de la "main window" avec le style
# main_button. Ceci doit être documenté pour en tirer profit.
widget "main window.*GtkButton*" style "main_button"
```

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

19. Écriture de vos propres widgets

19.1 Vue d'ensemble

Bien que la distribution GTK fournisse de nombreux types de widgets qui devraient couvrir la plupart des besoins de base, il peut arriver un moment où vous aurez besoin de créer votre propre type de widget. Comme GTK utilise l'héritage de widget de façon intensive et qu'il y a déjà un widget ressemblant à celui que vous voulez, il est souvent possible de créer un nouveau type de widget en seulement quelques lignes de code. Mais, avant de travailler sur un nouveau widget, il faut vérifier d'abord que quelqu'un ne l'a pas déjà écrit. Ceci éviter la duplication des efforts et maintient au minimum le nombre de widgets, ce qui permet de garder la cohérence du code et de l'interface des différentes applications. Un effet de bord est que, lorsque l'on a créé un nouveau widget, il faut l'annoncer afin que les autres puissent en bénéficier. Le meilleur endroit pour faire cela est, sans doute, la `gtk-list`.

19.2 Anatomie d'un widget

Afin de créer un nouveau widget, il importe de comprendre comment fonctionnent les objets GTK. Cette section ne se veut être qu'un rapide survol. Consultez la documentation de référence pour plus de détails.

Les widgets sont implantés selon une méthode orientée objet. Cependant, ils sont écrits en C standard. Ceci améliore beaucoup la portabilité et la stabilité, par contre cela signifie que celui qui écrit des widgets doit faire attention à certains détails d'implantation. Les informations

communes à toutes les instances d'une classe de widget (tous les widgets boutons, par exemple) sont stockées dans la *structure de la classe*. Il n'y en a qu'une copie dans laquelle sont stockées les informations sur les signaux de la classe (fonctionnement identique aux fonctions virtuelles en C). Pour permettre l'héritage, le premier champ de la structure de classe doit être une copie de la structure de classe du père. La déclaration de la structure de classe de *GtkButton* ressemble à ceci :

```
struct _GtkButtonClass
{
  GtkContainerClass parent_class;

  void (* pressed)  (GtkButton *button);
  void (* released) (GtkButton *button);
  void (* clicked)  (GtkButton *button);
  void (* enter)    (GtkButton *button);
  void (* leave)    (GtkButton *button);
};
```

Lorsqu'un bouton est traité comme un container (par exemple, lorsqu'il change de taille), sa structure de classe peut être convertie en *GtkContainerClass* et les champs adéquats utilisés pour gérer les signaux.

Il y a aussi une structure pour chaque widget créé sur une base d'instance. Cette structure a des champs pour stocker les informations qui sont différentes pour chaque instance du widget. Nous l'appelerons *structure d'objet*. Pour la classe *Button*, elle ressemble à :

```
struct _GtkButton
{
  GtkContainer container;

  GtkWidget *child;

  guint in_button : 1;
  guint button_down : 1;
};
```

Notez que, comme pour la structure de classe, le premier champ est la structure d'objet de la classe parente, cette structure peut donc être convertie dans la structure d'objet de la classe parente si besoin est.

19.3 Création d'un widget composé

Introduction

Un type de widget qui peut être intéressant à créer est un widget qui est simplement un agrégat d'autres widgets GTK. Ce type de widget ne fait rien qui ne pourrait être fait sans créer de nouveaux widgets, mais offre une méthode pratique pour empaqueter les éléments d'une interface utilisateur afin de la réutiliser facilement. Les widgets *FileSelection* et *ColorSelection* de la distribution standard sont des exemples de ce type de widget.

L'exemple de widget que nous allons créer dans cette section créera un widget *Tictactoe*, un tableau de 3x3 boutons commutateurs qui déclenche un signal lorsque tous les boutons d'une ligne, d'une colonne, ou d'une diagonale sont pressés.

Choix d'une classe parent

La classe parent d'un widget composé est, typiquement, la classe container contenant tous les éléments du widget composé. Par exemple, la classe parent du widget *FileSelection* est la classe *Dialog*. Comme nos boutons seront mis sous la forme d'un tableau, il semble naturel d'utiliser la classe *GtkTable* comme parent. Malheureusement, cela ne peut marcher. La création d'un widget est divisée en deux fonctions −− *WIDGETNAME_new()* que l'utilisateur appelle, et *WIDGETNAME_init()* qui réalise le travail d'initialisation du widget indépendamment des paramètre passés à la fonction `_new()`. Les widgets fils n'appellent que la fonction *_init* de leur widget parent. Mais cette division du travail ne fonctionne pas bien avec les tableaux qui, lorsqu'ils sont créés, ont besoin de connaître leue nombre de lignes et de colonnes. Sauf à dupliquer la plupart des fonctionnalités de *gtk_table_new()* dans notre widget *Tictactoe*, nous ferions mieux d'éviter de le dériver de *GtkTable*. Pour cette raison, nous la dériverons plutôt de *GtkVBox* et nous placerons notre table dans la VBox.

The header file

Chaque classe de widget possède un fichier en−tête qui déclare les structures d'objet et de classe pour ce widget, en plus de fonctions publiques. Quelques caractéristiques méritent d'être indiquées. Afin d'éviter des définitions multiples, on enveloppe le fichier en−tête avec :

```
#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__
.
.
.
#endif /* __TICTACTOE_H__ */
```

Et, pour faire plaisir aux programmes C++ qui incluront ce fichier, on l'enveloppe aussi dans :

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
.
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

En plus des fonctions et structures, nous déclarons trois macros standard, `TICTACTOE(obj)`, `TICTACTOE_CLASS(class)`, et `IS_TICTACTOE(obj)`, qui, respectivement, convertissent un pointeur en un pointeur vers une structure d'objet ou de classe, et vérifient si un objet est un widget Tictactoe.

Voici le fichier en−tête complet :

```
#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__

#include <gdk/gdk.h>
#include <gtk/gtkvbox.h>

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

```
#define TICTACTOE(obj)          GTK_CHECK_CAST (obj, tictactoe_get_type (), Tictact
#define TICTACTOE_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, tictactoe_get_type (),
#define IS_TICTACTOE(obj)       GTK_CHECK_TYPE (obj, tictactoe_get_type ())


typedef struct _Tictactoe       Tictactoe;
typedef struct _TictactoeClass  TictactoeClass;

struct _Tictactoe
{
  GtkVBox vbox;

  GtkWidget *buttons[3][3];
};

struct _TictactoeClass
{
  GtkVBoxClass parent_class;

  void (* tictactoe) (Tictactoe *ttt);
};

guint          tictactoe_get_type        (void);
GtkWidget*     tictactoe_new             (void);
void           tictactoe_clear           (Tictactoe *ttt);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __TICTACTOE_H__ */
```

La fonction `_get_type()`

Continuons maintenant avec l'implantation de notre widget. La fonction centrale pour chaque widget est *WIDGETNAME_get_type()*. Cette fonction, lorsqu'elle est appelée pour la première fois, informe le GTK de la classe et récupère un ID permettant d'identifier celle−ci de façon unique. Lors des appels suivants, elle ne fait que retourner cet ID.

```
guint
tictactoe_get_type ()
{
  static guint ttt_type = 0;

  if (!ttt_type)
    {
      GtkTypeInfo ttt_info =
      {
        "Tictactoe",
        sizeof (Tictactoe),
        sizeof (TictactoeClass),
        (GtkClassInitFunc) tictactoe_class_init,
        (GtkObjectInitFunc) tictactoe_init,
        (GtkArgFunc) NULL,
      };

      ttt_type = gtk_type_unique (gtk_vbox_get_type (), &ttt_info);
    }

  return ttt_type;
}
```

La structure *GtkTypeInfo* est définie de la façon suivante :

```
struct _GtkTypeInfo
{
  gchar *type_name;
  guint object_size;
  guint class_size;
  GtkClassInitFunc class_init_func;
  GtkObjectInitFunc object_init_func;
  GtkArgFunc arg_func;
};
```

Les champs de cette structure s'expliquent d'eux−mêmes. Nous ignorerons le champ *arg_func* ici : il a un rôle important permettant aux options des widgets d'être correctement initialisées à partir des langages interprétés, mais cette fonctionnalité est encore très peu implantée. Lorsque GTK dispose d'une copie correctement remplie de cette structure, il sait comment créer des objets d'un type particulier de widget.

La fonction *_class_init()*

La fonction *WIDGETNAME_class_init()* initialise les champs de la structure de classe du widget et configure tous les signaux de cette classe. Pour notre widget Tictactoe, cet appel est :

```
enum {
  TICTACTOE_SIGNAL,
  LAST_SIGNAL
};

static gint tictactoe_signals[LAST_SIGNAL] = { 0 };

static void
tictactoe_class_init (TictactoeClass *class)
{
  GtkObjectClass *object_class;

  object_class = (GtkObjectClass*) class;

  tictactoe_signals[TICTACTOE_SIGNAL] = gtk_signal_new ("tictactoe",
                                        GTK_RUN_FIRST,
                                        object_class->type,
                                        GTK_SIGNAL_OFFSET (TictactoeClass, tictact
                                        gtk_signal_default_marshaller, GTK_ARG_NON


  gtk_object_class_add_signals (object_class, tictactoe_signals, LAST_SIGNAL);

  class->tictactoe = NULL;
}
```

Notre widget n'a qu'un signal : "tictactoe", invoqué lorsqu'une ligne, une colonne ou une diagonale est complètement remplie. Tous les widgets composés n'ont pas besoin de signaux. Si vous lisez ceci pour la première fois, vous pouvez passer directement à la section suivante car les choses vont se compliquer un peu

La fonction :

```
gint   gtk_signal_new                    (gchar               *name,
                                          GtkSignalRunType     run_type,
                                          gint                 object_type,
                                          gint                 function_offset,
                                          GtkSignalMarshaller  marshaller,
                                          GtkArgType           return_val,
```

```
                                        gint                nparams,
                                        ...);
```

crée un nouveau signal. Les paramètres sont :

- *name* : Le nom du signal signal.
- *run_type* : Indique si le gestionnaire par défaut doit être lancé avant ou après le gestionnaire de l'utilisateur. Le plus souvent, ce sera `GTK_RUN_FIRST`, ou `GTK_RUN_LAST`, bien qu'il y ait d'autres possibilités.
- *object_type* : L'ID de l'objet auquel s'applique ce signal (il s'appliquera aussi au descendants de l'objet).
- *function_offset* : L'offset d'un pointeur vers le gestionnaire par défaut dans la structure de classe.
- *marshaller* : Fonction utilisée pour invoquer le gestionnaire de signal. Pour les gestionnaires de signaux n'ayant pas d'autres paramètres que l'objet émetteur et les données utilisateur, on peut utiliser la fonction prédéfinie *gtk_signal_default_marshaller()*.
- *return_val* : Type de la valeur retournée.
- *nparams* : Nombre de paramètres du gestionnaire de signal (autres que les deux par défaut mentionnés plus haut).
- ... : Types des paramètres.

Lorsque l'on spécifie les types, on utilise l'énumération *GtkArgType* :

```
typedef enum
{
  GTK_ARG_INVALID,
  GTK_ARG_NONE,
  GTK_ARG_CHAR,
  GTK_ARG_SHORT,
  GTK_ARG_INT,
  GTK_ARG_LONG,
  GTK_ARG_POINTER,
  GTK_ARG_OBJECT,
  GTK_ARG_FUNCTION,
  GTK_ARG_SIGNAL
} GtkArgType;
```

gtk_signal_new() retourne un identificateur entier pour le signal, que l'on stocke dans le tableau *tictactoe_signals*, indicé par une énumération (conventionnellement, les éléments de l'énumération sont le nom du signal, en majuscules, mais, ici, il y aurait un conflit avec la macro `TICTACTOE()`, nous l'appellerons donc `TICTACTOE_SIGNAL` à la place.

Après avoir créé nos signaux, nous devons demander à GTK d'associer ceux−ci à la classe Tictactoe. Ceci est fait en appelant *gtk_object_class_add_signals()*. Puis nous configurons le pointeur qui pointe sur le gestionnaire par défaut du signal "tictactoe" à NULL, pour indiquer qu'il n'y a pas d'action par défaut.

La fonction *_init()*

Chaque classe de widget a aussi besoin d'une fonction pour initialiser la structure d'objet. Habituellement, cette fonction a le rôle, plutôt limité, d'initialiser les champs de la structure avec des valeurs par défaut. Cependant, pour les widgets composés, cette fonction crée aussi les widgets composants.

```
static void
tictactoe_init (Tictactoe *ttt)
```

```
{
  GtkWidget *table;
  gint i,j;

  table = gtk_table_new (3, 3, TRUE);
  gtk_container_add (GTK_CONTAINER(ttt), table);
  gtk_widget_show (table);

  for (i=0;i<3; i++)
    for (j=0;j<3; j++)
      {
        ttt->buttons[i][j] = gtk_toggle_button_new ();
        gtk_table_attach_defaults (GTK_TABLE(table), ttt->buttons[i][j],
                                   i, i+1, j, j+1);
        gtk_signal_connect (GTK_OBJECT (ttt->buttons[i][j]), "toggled",
                            GTK_SIGNAL_FUNC (tictactoe_toggle), ttt);
        gtk_widget_set_usize (ttt->buttons[i][j], 20, 20);
        gtk_widget_show (ttt->buttons[i][j]);
      }
}
```

Et le reste...

Il reste une fonction que chaque widget (sauf pour les types widget de base, comme *GtkBin*, qui ne peuvent être instanciés) à besoin d'avoir −− celle que l'utilisateur appelle pour créer un objet de ce type. Elle est conventionnellement appelée *WIDGETNAME_new()*. Pour certains widgets, par pour ceux de Tictactoe, cette fonction prend des paramètres et réalise certaines initialisations dépendantes des paramètres. Les deux autres fonctions sont spécifiques au widget Tictactoe.

tictactoe_clear() est une fonction publique qui remet tous les boutons du widget en position relâchée. Notez l'utilisation de *gtk_signal_handler_block_by_data()* pour empêcher notre gestionnaire de signaux des boutons commutateurs d'être déclenché sans besoin.

tictactoe_toggle() est le gestionnaire de signal invoqué lorsqu'on clique sur un bouton. Il vérifie s'il y a des combinaisons gagnantes concernant le bouton qui vient d'être commuté et, si c'est le cas, émet le signal "tictactoe".

```
GtkWidget*
tictactoe_new ()
{
  return GTK_WIDGET ( gtk_type_new (tictactoe_get_type ()));
}

void
tictactoe_clear (Tictactoe *ttt)
{
  int i,j;

  for (i=0;i<3;i++)
    for (j=0;j<3;j++)
      {
        gtk_signal_handler_block_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
        gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON (ttt->buttons[i][j]),
                                     FALSE);
        gtk_signal_handler_unblock_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
      }
}

static void
tictactoe_toggle (GtkWidget *widget, Tictactoe *ttt)
{
```

```
  int i,k;

  static int rwins[8][3] = { { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                             { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                             { 0, 1, 2 }, { 0, 1, 2 } };
  static int cwins[8][3] = { { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                             { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                             { 0, 1, 2 }, { 2, 1, 0 } };

  int success, found;

  for (k=0; k<8; k++)
    {
      success = TRUE;
      found = FALSE;

      for (i=0;i<3;i++)
        {
          success = success &&
            GTK_TOGGLE_BUTTON(ttt->buttons[rwins[k][i]][cwins[k][i]])->active;
          found = found ||
            ttt->buttons[rwins[k][i]][cwins[k][i]] == widget;
        }

      if (success && found)
        {
          gtk_signal_emit (GTK_OBJECT (ttt),
                           tictactoe_signals[TICTACTOE_SIGNAL]);
          break;
        }
    }
}
```

Enfin, un exemple de programme utilisant notre widget Tictactoe

```
#include <gtk/gtk.h>
#include "tictactoe.h"

/* Invoqué lorsqu'une ligne, une colonne ou une diagonale est complète */

void win (GtkWidget *widget, gpointer data)
{
  g_print ("Ouais !\n");
  tictactoe_clear (TICTACTOE (widget));
}

int main (int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *ttt;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  gtk_container_border_width (GTK_CONTAINER (window), 10);

  /* Création d'un widget Tictactoe */
  ttt = tictactoe_new ();
  gtk_container_add (GTK_CONTAINER (window), ttt);
```

```
  gtk_widget_show (ttt);

  /* On lui attache le signal "tictactoe" */
  gtk_signal_connect (GTK_OBJECT (ttt), "tictactoe",
                      GTK_SIGNAL_FUNC (win), NULL);

  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

19.4 Création d'un widget à partir de zéro

Introduction

Dans cette section, nous en apprendrons plus sur la façon dont les widgets s'affichent eux−mêmes à l'écran et comment ils interagissent avec les événements. Comme exemple, nous créerons un widget d'appel télephonique interactif avec un pointeur que l'utilisateur pourra déplacer pour initialiser la valeur.

Afficher un widget à l'écran

Il y a plusieurs étapes mises en jeu lors de l'affichage. Lorsque le widget est créé par l'appel *WIDGETNAME_new()*, plusieurs autres fonctions supplémentaires sont requises.

- *WIDGETNAME_realize()* s'occupe de créer une fenêtre X pour le widget, s'il en a une.
- *WIDGETNAME_map()* est invoquée après l'appel de *gtk_widget_show()*. Elle s'assure que le widget est bien tracé à l'écran (*mappé*). Dans le cas d'une classe container, elle doit aussi appeler des fonctions *map()*> pour chaque widget fils.
- *WIDGETNAME_draw()* est invoquée lorsque *gtk_widget_draw()* est appelé pour le widget ou l'un de ces ancêtres. Elle réalise les véritables appels aux fonctions de dessin pour tracer le widget à l'écran. Pour les widgets containers, cette fonction doit appeler *gtk_widget_draw()* pour ses widgets fils.
- *WIDGETNAME_expose()* est un gestionnaire pour les événements d'exposition du widget. Il réalise les appels nécessaires aux fonctions de dessin pour tracer la partie exposée à l'écran. Pour les widgets containers, cette fonction doit générer les événements d'exposition pour ses widgets enfants n'ayant pas leurs propres fenêtres (s'ils ont leurs propres fenêtres, X génèrera les événements d'exposition nécessaires).

Vous avez pu noter que les deux dernières fonctions sont assez similaires −− chacune se charge de tracer le widget à l'écran. En fait, de nombreux types de widgets ne se préoccupent pas vraiment de la différence entre les deux. La fonction *draw()* par défaut de la classe widget génère simplement un événement d'exposition synthétique pour la zone à redessiner. Cependant, certains types de widgets peuvent économiser du travail en faisant la différence entre les deux fonctions. Par exemple, si un widget a plusieurs fenêtres X et puisque les événements d'exposition identifient la fenêtre exposée, il peut redessiner seulement la fenêtre concernée, ce qui n'est pas possible avec des appels à *draw()*.

Les widgets container, même s'ils ne se soucient pas eux−mêmes de la différence, ne peuvent pas utiliser simplement la fonction *draw()* car leurs widgets enfants tiennent compte de cette différence. Cependant, ce serait du gaspillage de dupliquer le code de tracé pour les deux fonctions. Conventionnellement, de tels widgets possèdent une fonction nommée *WIDGETNAME_paint()* qui réalise le véritable travail de tracé du widget et qui est appelée par

les fonctions `draw()` et `expose()`.

Dans notre exemple, comme le widget d'appel n'est pas un widget container et n'a qu'une fenêtre, nous pouvons utiliser l'approche la plus simple : utiliser la fonction *draw()* par défaut et n'implanter que la fonction *expose()*.

Origines du widget Dial

Exactement comme les animaux terrestres ne sont que des variantes des premiers amphibiens qui rampèrent hors de la boue, les widgets GTK sont des variantes d'autres widgets, déjà écrits. Ainsi, bien que cette section s'appelle « créer un widget à partir de zéro », le widget Dial commence réellement avec le code source du widget Range. Celui−ci a été pris comme point de départ car ce serait bien que notre Dial ait la même interface que les widgets Scale qui ne sont que des descendants spécialisés du widget Range. Par conséquent, bien que le code source soit présenté ci−dessous sous une forme achevée, cela n'implique pas qu'il a été écrit *deus ex machina*. De plus, si vous ne savez pas comment fonctionnent les widgets Scale du point de vue du programmeur de l'application, il est préférable de les étudier avant de continuer.

Les bases

Un petite partie de notre widget devrait ressembler au widget Tictactoe. Nous avons d'abord le fichier en−tête :

```
/* GTK − The GIMP Toolkit
 * Copyright (C) 1995−1997 Peter Mattis, Spencer Kimball and Josh MacDonald
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Library General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Library General Public License for more details.
 *
 * You should have received a copy of the GNU Library General Public
 * License along with this library; if not, write to the Free
 * Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef __GTK_DIAL_H__
#define __GTK_DIAL_H__

#include <gdk/gdk.h>
#include <gtk/gtkadjustment.h>
#include <gtk/gtkwidget.h>


#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */


#define GTK_DIAL(obj)          GTK_CHECK_CAST (obj, gtk_dial_get_type (), GtkDial)
#define GTK_DIAL_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, gtk_dial_get_type (), G
#define GTK_IS_DIAL(obj)       GTK_CHECK_TYPE (obj, gtk_dial_get_type ())


typedef struct _GtkDial        GtkDial;
```

```
typedef struct _GtkDialClass   GtkDialClass;

struct _GtkDial
{
  GtkWidget widget;

  /* politique de mise à jour
     (GTK_UPDATE_[CONTINUOUS/DELAYED/DISCONTINUOUS]) */

  guint policy : 2;

  /* Le bouton qui est pressé, 0 si aucun */
  guint8 button;

  /* Dimensions des composants de dial */
  gint radius;
  gint pointer_width;

  /* ID du timer de mise à jour, 0 si aucun */
  guint32 timer;

  /* Angle courant*/
  gfloat angle;

  /* Anciennes valeurs d'ajustement stockées. On sait donc quand quelque
     chose change */
  gfloat old_value;
  gfloat old_lower;
  gfloat old_upper;

  /* L'objet ajustment qui stocke les données de cet appel */
  GtkAdjustment *adjustment;
};

struct _GtkDialClass
{
  GtkWidgetClass parent_class;
};


GtkWidget*     gtk_dial_new                    (GtkAdjustment *adjustment);
guint          gtk_dial_get_type               (void);
GtkAdjustment* gtk_dial_get_adjustment         (GtkDial      *dial);
void           gtk_dial_set_update_policy      (GtkDial      *dial,
                                                GtkUpdateType  policy);

void           gtk_dial_set_adjustment         (GtkDial      *dial,
                                                GtkAdjustment *adjustment);
#ifdef __cplusplus
}
#endif /* __cplusplus */


#endif /* __GTK_DIAL_H__ */
```

Comme il y a plus de choses à faire avec ce widget par rapport à l'autre, nous avons plus de champs dans la structure de données, mais à part ça, les choses sont plutôt similaires.

Puis, après avoir inclus les fichiers en−tête et déclaré quelques constantes, nous devons fournir quelques fonctions pour donner des informations sur le widget et pour l'initialiser :

```
#include <math.h>
#include <stdio.h>
#include <gtk/gtkmain.h>
```

```
#include <gtk/gtksignal.h>

#include "gtkdial.h"

#define SCROLL_DELAY_LENGTH  300
#define DIAL_DEFAULT_SIZE 100

/* Déclararations des prototypes */

[ omis pour gagner de la place ]

/* Données locales */

static GtkWidgetClass *parent_class = NULL;

guint
gtk_dial_get_type ()
{
  static guint dial_type = 0;

  if (!dial_type)
    {
      GtkTypeInfo dial_info =
      {
        "GtkDial",
        sizeof (GtkDial),
        sizeof (GtkDialClass),
        (GtkClassInitFunc) gtk_dial_class_init,
        (GtkObjectInitFunc) gtk_dial_init,
        (GtkArgFunc) NULL,
      };

      dial_type = gtk_type_unique (gtk_widget_get_type (), &dial_info);
    }

  return dial_type;
}

static void
gtk_dial_class_init (GtkDialClass *class)
{
  GtkObjectClass *object_class;
  GtkWidgetClass *widget_class;

  object_class = (GtkObjectClass*) class;
  widget_class = (GtkWidgetClass*) class;

  parent_class = gtk_type_class (gtk_widget_get_type ());

  object_class->destroy = gtk_dial_destroy;

  widget_class->realize = gtk_dial_realize;
  widget_class->expose_event = gtk_dial_expose;
  widget_class->size_request = gtk_dial_size_request;
  widget_class->size_allocate = gtk_dial_size_allocate;
  widget_class->button_press_event = gtk_dial_button_press;
  widget_class->button_release_event = gtk_dial_button_release;
  widget_class->motion_notify_event = gtk_dial_motion_notify;
}

static void
gtk_dial_init (GtkDial *dial)
{
  dial->button = 0;
  dial->policy = GTK_UPDATE_CONTINUOUS;
```

```
  dial->timer = 0;
  dial->radius = 0;
  dial->pointer_width = 0;
  dial->angle = 0.0;
  dial->old_value = 0.0;
  dial->old_lower = 0.0;
  dial->old_upper = 0.0;
  dial->adjustment = NULL;
}

GtkWidget*
gtk_dial_new (GtkAdjustment *adjustment)
{
  GtkDial *dial;

  dial = gtk_type_new (gtk_dial_get_type ());

  if (!adjustment)
    adjustment = (GtkAdjustment*) gtk_adjustment_new (0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

  gtk_dial_set_adjustment (dial, adjustment);

  return GTK_WIDGET (dial);
}

static void
gtk_dial_destroy (GtkObject *object)
{
  GtkDial *dial;

  g_return_if_fail (object != NULL);
  g_return_if_fail (GTK_IS_DIAL (object));

  dial = GTK_DIAL (object);

  if (dial->adjustment)
    gtk_object_unref (GTK_OBJECT (dial->adjustment));

  if (GTK_OBJECT_CLASS (parent_class)->destroy)
    (* GTK_OBJECT_CLASS (parent_class)->destroy) (object);
}
```

Notez que cette fonction *init()* fait moins de choses que pour le widget Tictactoe car ce n'est pas un widget composé et que la fonction *new()* en fait plus car elle a maintenant un paramètre. Notez aussi que lorsque nous stockons un pointeur vers l'objet Adjustement, nous incrémentons son nombre de références (et nous le décrémentons lorsque nous ne l'utilisons plus) afin que GTK puisse savoir quand il pourra être détruit sans danger.

Il y a aussi quelques fonctions pour manipuler les options du widget :

```
GtkAdjustment*
gtk_dial_get_adjustment (GtkDial *dial)
{
  g_return_val_if_fail (dial != NULL, NULL);
  g_return_val_if_fail (GTK_IS_DIAL (dial), NULL);

  return dial->adjustment;
}

void
gtk_dial_set_update_policy (GtkDial      *dial,
                            GtkUpdateType  policy)
{
```

```
  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  dial->policy = policy;
}

void
gtk_dial_set_adjustment (GtkDial      *dial,
                         GtkAdjustment *adjustment)
{
  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  if (dial->adjustment)
    {
      gtk_signal_disconnect_by_data (GTK_OBJECT (dial->adjustment), (gpointer) dial
      gtk_object_unref (GTK_OBJECT (dial->adjustment));
    }

  dial->adjustment = adjustment;
  gtk_object_ref (GTK_OBJECT (dial->adjustment));

  gtk_signal_connect (GTK_OBJECT (adjustment), "changed",
                      (GtkSignalFunc) gtk_dial_adjustment_changed,
                      (gpointer) dial);
  gtk_signal_connect (GTK_OBJECT (adjustment), "value_changed",
                      (GtkSignalFunc) gtk_dial_adjustment_value_changed,
                      (gpointer) dial);

  dial->old_value = adjustment->value;
  dial->old_lower = adjustment->lower;
  dial->old_upper = adjustment->upper;

  gtk_dial_update (dial);
}
```

gtk_dial_realize()

Nous arrivons maintenant à quelques nouveaux types de fonctions. D'abord, nous avons une fonction qui réalise la création de la fenêtre X. Notez que l'on passe un masque à la fonction *gdk_window_new()* pour spécifier quels sont les champs de la structure GdkWindowAttr qui contiennent des données (les autres recevront des valeurs par défaut). Notez aussi la façon dont est créé le masque d'événement du widget. On appelle *gtk_widget_get_events()* pour récupérer le masque d'événement que l'utilisateur a spécifié pour ce widget (avec *gtk_widget_set_events()*) et ajouter les événements qui nous intéressent.

Après avoir créé la fenêtre, nous configurons son style et son fond et mettons un pointeur vers le widget dans le champ user de la GdkWindow. Cette dernière étape permet à GTK de distribuer les événements pour cette fenêtre au widget correct.

```
static void
gtk_dial_realize (GtkWidget *widget)
{
  GtkDial *dial;
  GdkWindowAttr attributes;
  gint attributes_mask;

  g_return_if_fail (widget != NULL);
  g_return_if_fail (GTK_IS_DIAL (widget));

  GTK_WIDGET_SET_FLAGS (widget, GTK_REALIZED);
  dial = GTK_DIAL (widget);
```

```
  attributes.x = widget->allocation.x;
  attributes.y = widget->allocation.y;
  attributes.width = widget->allocation.width;
  attributes.height = widget->allocation.height;
  attributes.wclass = GDK_INPUT_OUTPUT;
  attributes.window_type = GDK_WINDOW_CHILD;
  attributes.event_mask = gtk_widget_get_events (widget) |
    GDK_EXPOSURE_MASK | GDK_BUTTON_PRESS_MASK |
    GDK_BUTTON_RELEASE_MASK | GDK_POINTER_MOTION_MASK |
    GDK_POINTER_MOTION_HINT_MASK;
  attributes.visual = gtk_widget_get_visual (widget);
  attributes.colormap = gtk_widget_get_colormap (widget);

  attributes_mask = GDK_WA_X | GDK_WA_Y | GDK_WA_VISUAL | GDK_WA_COLORMAP;
  widget->window = gdk_window_new (widget->parent->window, &attributes, attributes_

  widget->style = gtk_style_attach (widget->style, widget->window);

  gdk_window_set_user_data (widget->window, widget);

  gtk_style_set_background (widget->style, widget->window, GTK_STATE_ACTIVE);
}
```

Négotiation de la taille

Avant le premier affichage de la fenêtre contenant un widget et à chaque fois que la forme de la fenêtre change, GTK demande à chaque widget fils la taille qu'il désire avoir. Cette requête est gérée par la fonction *gtk_dial_size_request()*. Comme notre widget n'est pas un widget container, et n'a pas de contraintes réelles sur sa taille, nous ne faisons que retourner une valeur raisonnable par défaut.

```
static void
gtk_dial_size_request (GtkWidget      *widget,
                       GtkRequisition *requisition)
{
  requisition->width = DIAL_DEFAULT_SIZE;
  requisition->height = DIAL_DEFAULT_SIZE;
}
```

Lorsque tous les widgets on demandé une taille idéale, le forme de la fenêtre est calculée et chaque widget fils est averti de sa taille. Habituellement, ce sera autant que la taille requise, mais si, par exemple, l'utilisateur a redimensionné la fenêtre, cette taille peut occasionnellement être plus petite que la taille requise. La notification de la taille est gérée par la fonction *gtk_dial_size_allocate()*. Notez qu'en même temps qu'elle calcule les tailles de certains composants pour une utilisation future, cette routine fait aussi le travail de base consistant à déplacer les widgets X Window dans leur nouvelles positions et tailles.

```
static void
gtk_dial_size_allocate (GtkWidget     *widget,
                        GtkAllocation *allocation)
{
  GtkDial *dial;

  g_return_if_fail (widget != NULL);
  g_return_if_fail (GTK_IS_DIAL (widget));
  g_return_if_fail (allocation != NULL);

  widget->allocation = *allocation;
  if (GTK_WIDGET_REALIZED (widget))
    {
```

```
        dial = GTK_DIAL (widget);

        gdk_window_move_resize (widget->window,
                                allocation->x, allocation->y,
                                allocation->width, allocation->height);

        dial->radius = MAX(allocation->width,allocation->height) * 0.45;
        dial->pointer_width = dial->radius / 5;
      }
  }
```

.

gtk_dial_expose()

Comme cela est mentionné plus haut, tout le dessin de ce widget est réalisé dans le gestionnaire pour les événements d'exposition. Il n'y a pas grand chose de plus à dire là dessus, sauf constater l'utilisation de la fonction *gtk_draw_polygon* pour dessiner le pointeur avec une forme en trois dimensions selon les couleurs stockées dans le style du widget. style.

```
static gint
gtk_dial_expose (GtkWidget      *widget,
                 GdkEventExpose *event)
{
  GtkDial *dial;
  GdkPoint points[3];
  gdouble s,c;
  gdouble theta;
  gint xc, yc;
  gint tick_length;
  gint i;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  if (event->count > 0)
    return FALSE;

  dial = GTK_DIAL (widget);

  gdk_window_clear_area (widget->window,
                         0, 0,
                         widget->allocation.width,
                         widget->allocation.height);

  xc = widget->allocation.width/2;
  yc = widget->allocation.height/2;

  /* Draw ticks */

  for (i=0; i<25; i++)
    {
      theta = (i*M_PI/18. - M_PI/6.);
      s = sin(theta);
      c = cos(theta);

      tick_length = (i%6 == 0) ? dial->pointer_width : dial->pointer_width/2;

      gdk_draw_line (widget->window,
                     widget->style->fg_gc[widget->state],
                     xc + c*(dial->radius - tick_length),
                     yc - s*(dial->radius - tick_length),
```

```
                    xc + c*dial->radius,
                    yc - s*dial->radius);
    }

  /* Draw pointer */

  s = sin(dial->angle);
  c = cos(dial->angle);


  points[0].x = xc + s*dial->pointer_width/2;
  points[0].y = yc + c*dial->pointer_width/2;
  points[1].x = xc + c*dial->radius;
  points[1].y = yc - s*dial->radius;
  points[2].x = xc - s*dial->pointer_width/2;
  points[2].y = yc - c*dial->pointer_width/2;

  gtk_draw_polygon (widget->style,
                    widget->window,
                    GTK_STATE_NORMAL,
                    GTK_SHADOW_OUT,
                    points, 3,
                    TRUE);

  return FALSE;
}
```

Gestion des événements

Le reste du code du widget gère différents types d'événements et n'est pas trop différent de ce que l'on trouve dans la plupart des applications GTK. Deux types d'événements peuvent survenir -- l'utilisateur peut cliquer sur le widget avec la souris et faire glisser pour déplacer le pointeur, ou bien la valeur de l'objet Adjustment peut changer à cause d'une circonstance extérieure.

Lorsque l'utilisateur clique sur le widget, on vérifie si le clic s'est bien passé près du pointeur et si c'est le cas, on stocke alors le bouton avec lequel l'utilisateur a cliqué dans le champ *button* de la structure du widget et on récupère tous les événements souris avec un appel à *gtk_grab_add()*. Un déplacement ultérieur de la souris provoque le recalcul de la valeur de contrôle (par la fonction *gtk_dial_update_mouse*). Selon la politique qui a été choisie, les événements "value_changed" sont, soit générés instantanément (*GTK_UPDATE_CONTINUOUS*), après un délai ajouté au timer avec *gtk_timeout_add()* (*GTK_UPDATE_DELAYED*), ou seulement lorsque le bouton est relâché (*GTK_UPDATE_DISCONTINUOUS*).

```
static gint
gtk_dial_button_press (GtkWidget      *widget,
                       GdkEventButton *event)
{
  GtkDial *dial;
  gint dx, dy;
  double s, c;
  double d_parallel;
  double d_perpendicular;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  /* Détermine si le bouton pressé est dans la région du pointeur.
     On fait cela en calculant les distances parallèle et perpendiculaire
     du point où la souris a été pressée par rapport à la ligne passant
```

```
     par le pointeur */

  dx = event->x - widget->allocation.width / 2;
  dy = widget->allocation.height / 2 - event->y;

  s = sin(dial->angle);
  c = cos(dial->angle);

  d_parallel = s*dy + c*dx;
  d_perpendicular = fabs(s*dx - c*dy);

  if (!dial->button &&
      (d_perpendicular < dial->pointer_width/2) &&
      (d_parallel > - dial->pointer_width))
    {
      gtk_grab_add (widget);

      dial->button = event->button;

      gtk_dial_update_mouse (dial, event->x, event->y);
    }

  return FALSE;
}

static gint
gtk_dial_button_release (GtkWidget      *widget,
                         GdkEventButton *event)
{
  GtkDial *dial;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  if (dial->button == event->button)
    {
      gtk_grab_remove (widget);

      dial->button = 0;

      if (dial->policy == GTK_UPDATE_DELAYED)
        gtk_timeout_remove (dial->timer);

      if ((dial->policy != GTK_UPDATE_CONTINUOUS) &&
          (dial->old_value != dial->adjustment->value))
        gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
    }

  return FALSE;
}

static gint
gtk_dial_motion_notify (GtkWidget      *widget,
                        GdkEventMotion *event)
{
  GtkDial *dial;
  GdkModifierType mods;
  gint x, y, mask;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);
```

```
  dial = GTK_DIAL (widget);

  if (dial->button != 0)
    {
      x = event->x;
      y = event->y;

      if (event->is_hint || (event->window != widget->window))
        gdk_window_get_pointer (widget->window, &x, &y, &mods);

      switch (dial->button)
        {
        case 1:
          mask = GDK_BUTTON1_MASK;
          break;
        case 2:
          mask = GDK_BUTTON2_MASK;
          break;
        case 3:
          mask = GDK_BUTTON3_MASK;
          break;
        default:
          mask = 0;
          break;
        }

      if (mods & mask)
        gtk_dial_update_mouse (dial, x,y);
    }

  return FALSE;
}

static gint
gtk_dial_timer (GtkDial *dial)
{
  g_return_val_if_fail (dial != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (dial), FALSE);

  if (dial->policy == GTK_UPDATE_DELAYED)
    gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");

  return FALSE;
}

static void
gtk_dial_update_mouse (GtkDial *dial, gint x, gint y)
{
  gint xc, yc;
  gfloat old_value;

  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  xc = GTK_WIDGET(dial)->allocation.width / 2;
  yc = GTK_WIDGET(dial)->allocation.height / 2;

  old_value = dial->adjustment->value;
  dial->angle = atan2(yc-y, x-xc);

  if (dial->angle < -M_PI/2.)
    dial->angle += 2*M_PI;

  if (dial->angle < -M_PI/6)
```

```
    dial->angle = -M_PI/6;

  if (dial->angle > 7.*M_PI/6.)
    dial->angle = 7.*M_PI/6.;

  dial->adjustment->value = dial->adjustment->lower + (7.*M_PI/6 - dial->angle) *
    (dial->adjustment->upper - dial->adjustment->lower) / (4.*M_PI/3.);

  if (dial->adjustment->value != old_value)
    {
      if (dial->policy == GTK_UPDATE_CONTINUOUS)
        {
          gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
        }
      else
        {
          gtk_widget_draw (GTK_WIDGET(dial), NULL);

          if (dial->policy == GTK_UPDATE_DELAYED)
            {
              if (dial->timer)
                gtk_timeout_remove (dial->timer);

              dial->timer = gtk_timeout_add (SCROLL_DELAY_LENGTH,
                                             (GtkFunction) gtk_dial_timer,
                                             (gpointer) dial);
            }
        }
    }
}
```

Les changements de l'Adjustement par des moyens extérieurs sont communiqués à notre widget par les signaux "changed" et "value_changed". Les gestionnaires pour ces fonctions appellent *gtk_dial_update()* pour valider les paramètres, calculer le nouvel angle du pointeur et redessiner le widget (en appelant *gtk_widget_draw()*).

```
static void
gtk_dial_update (GtkDial *dial)
{
  gfloat new_value;

  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  new_value = dial->adjustment->value;

  if (new_value < dial->adjustment->lower)
    new_value = dial->adjustment->lower;

  if (new_value > dial->adjustment->upper)
    new_value = dial->adjustment->upper;

  if (new_value != dial->adjustment->value)
    {
      dial->adjustment->value = new_value;
      gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
    }

  dial->angle = 7.*M_PI/6. - (new_value - dial->adjustment->lower) * 4.*M_PI/3. /
    (dial->adjustment->upper - dial->adjustment->lower);

  gtk_widget_draw (GTK_WIDGET(dial), NULL);
}
```

```
static void
gtk_dial_adjustment_changed (GtkAdjustment *adjustment,
                             gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if ((dial->old_value != adjustment->value) ||
      (dial->old_lower != adjustment->lower) ||
      (dial->old_upper != adjustment->upper))
    {
      gtk_dial_update (dial);

      dial->old_value = adjustment->value;
      dial->old_lower = adjustment->lower;
      dial->old_upper = adjustment->upper;
    }
}

static void
gtk_dial_adjustment_value_changed (GtkAdjustment *adjustment,
                                   gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if (dial->old_value != adjustment->value)
    {
      gtk_dial_update (dial);

      dial->old_value = adjustment->value;
    }
}
```

Améliorations possibles

Le widget Dial décrit jusqu'à maintenant exécute à peu près 670 lignes de code. Bien que cela puisse sembler beaucoup, nous en avons vraiment fait beaucoup avec ce code, notamment parce que la majeure partie de cette longueur est due aux en−têtes et à la préparation. Cependant, certaines améliorations peuvent être apportées à ce widget :

- Si vous testez ce widget, vous vous apercevrez qu'il y a un peu de scintillement lorsque le pointeur est déplacé. Ceci est dû au fait que le widget entier est effacé, puis redessiné à chaque mouvement du pointeur. Souvent, la meilleure façon de gérer ce problème est de dessiner sur un pixmap non affiché, puis de copier le résultat final sur l'écran en une seule étape (le widget *ProgressBar* se dessine de cette façon).
- L'utilisateur devrait pouvoir utiliser les flèches du curseur vers le haut et vers le bas pour incrémenter et décrémenter la valeur.
- Ce serait bien si le widget avait des boutons pour augmenter et diminuer la valeur dans de petites ou de grosses proportions. Bien qu'il serait possible d'utiliser les widgets *Button* pour cela, nous voudrions aussi que les boutons s'auto−répètent lorsqu'ils sont maintenus appuyés, comme font les flèches d'une barre de défilement. La majeure partie du code pour implanter ce type de comportement peut se trouver dans le widget

GtkRange.

- Le widget Dial pourrait être fait dans un widget container avec un seul widget fils positionnée en bas, entre les boutons mentionnés ci−dessus. L'utilisateur pourrait alors ajouter au choix, un widget label ou entrée pour afficher la valeur courante de l'appel.

19.5 En savoir plus

Seule une petite partie des nombreux détails de la création des widgets a pu être décrite. Si vous désirez écrire vos propres widgets, la meilleure source d'exemples est le source de GTK lui−même. Posez−vous quelques questions sur les widgets que vous voulez écrire : est−ce un widget container ? possède−t−il sa propre fenêtre ? est−ce une modification d'un widget existant ? Puis, trouvez un widget identique et commencez à faire les modifications. Bonne chance !

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

20. Scribble, un programme simple de dessin

20.1 Présentation

Dans cette section, nous construirons un programme simple de dessin. Ce faisant, nous examinerons comment gérer les événements souris, comment dessiner dans une fenêtre, et comment mieux dessiner en utilisant un pixmap en arrière plan. Après avoir créé ce programme, nous l'étendrons en lui ajoutant le support des périphériques *Xinput*, comme les tables de tracé. GTK dispose de routines de support qui facilitent beaucoup l'obtention des informations étendues (comme la pression et l'inclinaison du stylet) à partir de tels périphériques.

20.2 Gestion d'événement

Les signaux GTK que nous avons déjà vus concernent les actions de haut niveau, comme la sélection d'un choix d'un menu. Cependant, il est quelques fois utile de connaître les cas de bas niveau, comme le déplacement de la souris, ou la pression d'une touche. Il existe aussi des signaux GTK correspondant à ces *événements* bas niveau. Les gestionnaires de ces signaux ont un paramètre supplémentaire qui est un pointeur vers une structure contenant des informations sur l'événement. Par exemple, les gestionnaires des événements de déplacement recoivent un paramètre vers une structure *GdkEventMotion* qui ressemble (en partie) à ceci :

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
  gdouble x;
  gdouble y;
  ...
  guint state;
  ...
};
```

type sera initialisé avec le type de l'événement, ici *GDK_MOTION_NOTIFY*, *window* est la fenêtre dans laquelle l'événement est survenu. *x* et *y* donnent les coordonnées de l'événement et

state spécifie l'état du modificateur lorsque l'événement s'est produit (c'est−à−dire quelles sont les touches de modification et les boutons souris qui ont été pressés). Il s'agit d'un OU bit à bit de l'une des valeurs suivantes :

```
GDK_SHIFT_MASK
GDK_LOCK_MASK
GDK_CONTROL_MASK
GDK_MOD1_MASK
GDK_MOD2_MASK
GDK_MOD3_MASK
GDK_MOD4_MASK
GDK_MOD5_MASK
GDK_BUTTON1_MASK
GDK_BUTTON2_MASK
GDK_BUTTON3_MASK
GDK_BUTTON4_MASK
GDK_BUTTON5_MASK
```

Comme pour les autres signaux, on appelle *gtk_signal_connect()* pour déterminer ce qui se passe lorsqu'un événement survient. Mais nous devons aussi faire en sorte que GTK sache de quels événements nous voulons être avertis. Pour ce faire, on appelle la fonction :

```
void       gtk_widget_set_events          (GtkWidget           *widget,
                                           gint                 events);
```

Le deuxième champ spécifie les événements qui nous intéressent. Il s'agit d'un OU bit à bit de constantes qui indiquent différent types d'événements. Pour référence ultérieure, les types d'événements sont :

```
GDK_EXPOSURE_MASK
GDK_POINTER_MOTION_MASK
GDK_POINTER_MOTION_HINT_MASK
GDK_BUTTON_MOTION_MASK
GDK_BUTTON1_MOTION_MASK
GDK_BUTTON2_MOTION_MASK
GDK_BUTTON3_MOTION_MASK
GDK_BUTTON_PRESS_MASK
GDK_BUTTON_RELEASE_MASK
GDK_KEY_PRESS_MASK
GDK_KEY_RELEASE_MASK
GDK_ENTER_NOTIFY_MASK
GDK_LEAVE_NOTIFY_MASK
GDK_FOCUS_CHANGE_MASK
GDK_STRUCTURE_MASK
GDK_PROPERTY_CHANGE_MASK
GDK_PROXIMITY_IN_MASK
GDK_PROXIMITY_OUT_MASK
```

Il y a quelques points subtils qui doivent être observés lorsqu'on appelle *gtk_widget_set_events()*. D'abord, elle doit être appelée avant que la fenêtre X d'un widget GTK soit créée. En pratique, cela signifie que l'on doit l'appeler immédiatement après avoir créé le widget. Ensuite, le widget doit avoir une fenêtre X associée. Pour des raisons d'efficacité, de nombreux types de widgets n'ont pas de fenêtre propre, mais se dessinent dans la fenêtre de leur parent. Ces widgets sont :

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
```

```
GtkLabel
GtkPaned
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkViewport
GtkAspectFrame
GtkFrame
GtkVPaned
GtkHPaned
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

Pour capturer les événements pour ces widgets, on doit utiliser un widget *EventBox*. Voir la section sur Le widget EventBox pour plus de détails.

Pour notre programme de dessin, on veut savoir quand le bouton de la souris est pressé et quand la souris est déplacée, nous indiquons donc *GDK_POINTER_MOTION_MASK* et *GDK_BUTTON_PRESS_MASK*. On veut aussi savoir quand il est nécessaire de redessiner notre fenêtre, on indique donc *GDK_EXPOSURE_MASK*. Bien que nous voulions être avertis via un événement *Configure* lorsque la taille de notre fenêtre change, on n'a pas besoin de préciser le flag *GDK_STRUCTURE_MASK* correspondant car il est automatiquement spécifié pour chaque fenêtre.

Il arrive cependant qu'il puisse y avoir un problème en indiquant seulement *GDK_POINTER_MOTION_MASK*. Cela fera que le serveur ajoutera un nouvel événement de déplacement à la file des événements à chaque fois que l'utilisateur déplace la souris. Si cela nous prend 0,1 seconde pour gérer un événement de déplacement, si le serveur X n'ajoute un nouvel événement de déplacement dans la queue que toutes les 0,05 secondes, nous serons vite à la traîne de l'utilisateur. Si l'utilisateur dessine pendant 5 secondes, cela nous prendra 5 secondes de plus pour le traiter après qu'il ait relâché le bouton de la souris ! Ce que l'on voudrait, c'est ne récupérer qu'un événement de déplacement pour chaque événement que l'on traite. Pour cela, il faut préciser *GDK_POINTER_MOTION_HINT_MASK*.

Avec *GDK_POINTER_MOTION_HINT_MASK*, le serveur nous envoit un événement de déplacement la première fois que la pointeur se déplace après être entré dans la fenêtre, ou après un événement d'appui ou de relâchement d'un bouton. Les événements de déplacement suivants seront supprimés jusqu'à ce que l'on demande explicitement la position du pointeur en utilisant la fonction :

```
GdkWindow*    gdk_window_get_pointer     (GdkWindow       *window,
                                          gint            *x,
                                          gint            *y,
                                          GdkModifierType *mask);
```

(Il existe une autre fonction, *gtk_widget_get_pointer()* qui possède une interface simple, mais n'est pas très utile car elle ne fait que récupérer la position de la souris et ne se préoccupe pas de savoir si les boutons sont pressés).

Le code pour configurer les événements pour notre fenêtre ressemble alors à :

```
gtk_signal_connect (GTK_OBJECT (drawing_area), "expose_event",
                    (GtkSignalFunc) expose_event, NULL);
gtk_signal_connect (GTK_OBJECT(drawing_area),"configure_event",
                    (GtkSignalFunc) configure_event, NULL);
```

```
  gtk_signal_connect (GTK_OBJECT (drawing_area), "motion_notify_event",
                      (GtkSignalFunc) motion_notify_event, NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area), "button_press_event",
                      (GtkSignalFunc) button_press_event, NULL);

  gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK
                         | GDK_LEAVE_NOTIFY_MASK
                         | GDK_BUTTON_PRESS_MASK
                         | GDK_POINTER_MOTION_MASK
                         | GDK_POINTER_MOTION_HINT_MASK);
```

Nous garderons les gestionnaires de "expose_event" et "configure_event" pour plus tard. Les gestionnaires de "motion_notify_event" et "button_press_event" sont très simples :

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  if (event->button == 1 && pixmap != NULL)
      draw_brush (widget, event->x, event->y);

  return TRUE;
}

static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  int x, y;
  GdkModifierType state;

  if (event->is_hint)
    gdk_window_get_pointer (event->window, &x, &y, &state);
  else
    {
      x = event->x;
      y = event->y;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, x, y);

  return TRUE;
}
```

20.3 Le widget DrawingArea et le dessin

Revenons au processus de dessin sur l'écran. Le widget que l'on utilise pour ceci est le widget *DrawingArea*. Un tel widget est essentiellement une fenêtre X et rien de plus. Il s'agit d'une toile vide sur laquelle nous pouvons dessiner ce que nous voulons.

Une zone de dessin est créée avec l'appel :

```
GtkWidget* gtk_drawing_area_new        (void);
```

Une taille par défaut peut être donnée au widget par l'appel :

```
void       gtk_drawing_area_size       (GtkDrawingArea      *darea,
                                        gint                 width,
                                        gint                 height);
```

Cette taille par défaut peu être surchargée en appelant *gtk_widget_set_usize()* et celle−ci, à son tour, peut être surchargée si l'utilisateur modifie manuellement la taille de la fenêtre contenant la zone de dessin.

Il faut noter que lorsque l'on crée un widget *DrawingArea*, nous sommes *complètement* responsable du dessin du contenu. Si notre fenêtre est cachée puis redécouverte, nous recevrons un événement d'exposition et devrons redessiner ce qui a été caché auparavant.

Devoir se rappeler tout ce qui a été dessiné à l'écran pour pouvoir correctement le redessiner peut s'avérer, c'est le moins que l'on puisse dire, pénible. De plus, cela peut être visible si des portions de la fenêtre sont effacées puis redessinées étape par étape. La solution à ce problème est d'utiliser un *pixmap d'arrière−plan* qui n'est pas sur l'écran. Au lieu de dessiner directement à l'écran, on dessine sur une image stockée dans la mémoire du serveur et qui n'est pas affichée, puis, lorsque l'image change ou lorsque de nouvelles portions de l'image sont affichées, on copie les parties adéquates sur l'écran.

Pour créer un pixmap mémoire, on appelle la fonction :

```
GdkPixmap* gdk_pixmap_new               (GdkWindow  *window,
                                         gint        width,
                                         gint        height,
                                         gint        depth);
```

Le paramètre *windows* indique une fenêtre GTK de laquelle ce pixmap tire certaines de ses propriétés. *width* et *height* précisent la taille du pixmap. *depth* précise la *profondeur de couleur*, c'est−à−dire le nombre de bits par pixel, de la nouvelle fenêtre. Si cette profondeur vaut *−1*, elle correspondra à celle de *window*.

Nous créons le pixmap dans notre gestionnaire "configure_event". Cet événement est généré à chaque fois que la fenêtre change de taille, y compris lorsqu'elle initialement créée.

```
/* Pixmap d'arrière−plan pour la zone de dessin */
static GdkPixmap *pixmap = NULL;

/* Création d'un nouveau pixmap d'arrière−plan de la taille voulue */
static gint
configure_event (GtkWidget *widget, GdkEventConfigure *event)
{
  if (pixmap)
    {
      gdk_pixmap_destroy(pixmap);
    }
  pixmap = gdk_pixmap_new(widget−>window,
                          widget−>allocation.width,
                          widget−>allocation.height,
                          −1);
  gdk_draw_rectangle (pixmap,
                      widget−>style−>white_gc,
                      TRUE,
                      0, 0,
                      widget−>allocation.width,
                      widget−>allocation.height);

  return TRUE;
}
```

L'appel à *gdk_draw_rectangle()* remet le pixmap à blanc. Nous en dirons un peu plus dans un moment.

Notre gestionnaire d'événement d'exposition copie alors simplement la partie concernées du pixmap sur l'écran (on détermine la zone qu'il faut redessiner en utilisant le champ *event−>area* de l'événement d'exposition) :

```
/* Remplit l'écran à partir du pixmap d'arrière−plan */
static gint
expose_event (GtkWidget *widget, GdkEventExpose *event)
{
  gdk_draw_pixmap(widget−>window,
                  widget−>style−>fg_gc[GTK_WIDGET_STATE (widget)],
                  pixmap,
                  event−>area.x, event−>area.y,
                  event−>area.x, event−>area.y,
                  event−>area.width, event−>area.height);

  return FALSE;
}
```

Nous avons vu comment garder l'écran à jour avec notre pixmap, mais comment dessine−t−on réellement ce que l'on veut dans le pixmap ? Il existe un grand nombre d'appels dans la bibliothèque GDK de GTK pour dessiner sur des *dessinables*. Un dessinable est simplement quelque chose sur lequel on peut dessiner. Cela peut être une fenêtre, un pixmap, ou un bitmap (une image en noir et blanc). Nous avons déjà vu plus haut deux de ces appels, *gdk_draw_rectangle()* et *gdk_draw_pixmap()*. La liste complète est :

```
gdk_draw_line ()
gdk_draw_rectangle ()
gdk_draw_arc ()
gdk_draw_polygon ()
gdk_draw_string ()
gdk_draw_text ()
gdk_draw_pixmap ()
gdk_draw_bitmap ()
gdk_draw_image ()
gdk_draw_points ()
gdk_draw_segments ()
```

Consultez la documentation de référence ou le fichier en−tête <*gdk*gdk.h>/ pour plus de détails sur ces fonctions. Celles−ci partagent toutes les mêmes deux paramêtres. Le premier est le dessinable sur lequel dessiner, le second est un *contexte graphique* (GC).

Un contexte graphique encapsule l'information sur des choses comme les couleurs de premier et d'arrière plan et la largeur de ligne. GDK possède un ensemble complet de fonctions pour créer et manipuler les contextes graphiques, mais, pour simplifier, nous n'utiliserons que les contextes graphiques prédéfinis. Chaque widget a un style associé (qui peut être modifié dans un fichier gtkrc, voir la section sur les fichiers rc de GTK). Celui−ci, entre autres choses, stocke plusieurs contextes graphiques. Quelques exemples d'accès à ces contextes sont :

```
widget−>style−>white_gc
widget−>style−>black_gc
widget−>style−>fg_gc[GTK_STATE_NORMAL]
widget−>style−>bg_gc[GTK_WIDGET_STATE(widget)]
```

Les champs *fg_gc*, *bg_gc*, *dark_gc* et *light_gc* sont indexés par un paramètre de type *GtkStateType* qui peut prendre les valeurs :

```
GTK_STATE_NORMAL,
GTK_STATE_ACTIVE,
GTK_STATE_PRELIGHT,
GTK_STATE_SELECTED,
```

```
GTK_STATE_INSENSITIVE
```

Par exemple, pour *GTK_STATE_SELECTED*, la couleur de premier plan par défaut est blanc et la couleur d'arrière plan par défaut est bleu foncé.

Notre fonction *draw_brush()*, qui réalise le dessin à l'écran est alors :

```
/* Dessine un rectangle à l'écran */
static void
draw_brush (GtkWidget *widget, gdouble x, gdouble y)
{
  GdkRectangle update_rect;

  update_rect.x = x - 5;
  update_rect.y = y - 5;
  update_rect.width = 10;
  update_rect.height = 10;
  gdk_draw_rectangle (pixmap,
                      widget->style->black_gc,
                      TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

Après avoir dessiné le rectangle représentant la brosse sur le pixmap, nous appelons la fonction :

```
void       gtk_widget_draw               (GtkWidget           *widget,
                                          GdkRectangle        *area);
```

qui indique à X que la zone donnée par le paramètre *area* a besoin d'être mise à jour. X génèrera éventuellement un événement d'exposition (en combinant peut-être les zones passés dans plusieurs appels à *gtk_widget_draw()*) ce qui forcera notre gestionnaire d'événement d'exposition à copier les parties adéquates à l'écran.

Nous avons maintenant couvert entièrement le programme de dessin, sauf quelques détails banals comme la création de la fenêtre principale. Le code source complet est disponible à l'endroit où vous avez obtenu ce didacticiel.

20.4 Ajouter le support XInput

Il est maintenant possible d'acheter des périphériques bon marché, comme les tablettes graphiques qui permettent d'exprimer beaucoup plus facilement son talent qu'avec une souris. La façon la plus simple pour utiliser de tels périphériques est simplement de le faire comme un remplacement de la souris, mais cela ne tire pas partie des nombreux avantages de ces périphériques, comme :

- Sensibilité à la pression ;
- rapport d'inclinaison ;
- positionnement au dessous du pixel ;
- entrées multiples (par exemple, un stylet avec pointe et gomme).

Pour des informations sur l'extension XInput, voir XInput-HOWTO.

Si l'on examine la définition complète de, par exemple, la structure *GdkEventMotion*, on voit qu'elle possède des champs pour supporter des informations étendues sur les périphériques.

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  gint16 is_hint;
  GdkInputSource source;
  guint32 deviceid;
};
```

pressure indique la pression comme un nombre réel compris entre 0 et 1. *xtilt* et *ytilt* peuvent prendre des valeurs entre −1 et 1, correspondant au degré d'inclinaison dans chaque direction, *source* et *deviceid* précisent de deux façons différentes le périphérique pour lequel l'événement est survenus. *source* donne une simple information sur le type du périphérique. Il peut prendre l'une des valeurs suivantes :

```
GDK_SOURCE_MOUSE
GDK_SOURCE_PEN
GDK_SOURCE_ERASER
GDK_SOURCE_CURSOR
```

`deviceid` précise un ID numérique unique pour le périphérique. Il peut être utilisé pour trouver des informations supplémentaires sur le périphérique en utilisant l'appel *gdk_input_list_devices()* (voir ci−dessous). La valeur spéciale *GDK_CORE_POINTER* sert à désigner le périphérique de pointage principal (habituellement la souris).

Valider l'information supplémentaire sur un périphérique

Pour indiquer à GTK que l'on désire obtenir des informations supplémentaires sur le périphérique, on a simplement besoin d'ajouter une ligne à nos programmes.

```
gtk_widget_set_extension_events (drawing_area, GDK_EXTENSION_EVENTS_CURSOR);
```

En donnant la valeur *GDK_EXTENSION_EVENTS_CURSOR*, on indique que nous désirons les événements d'extension, mais seulement si l'on ne doit pas dessiner notre propre curseur. Voir la section Sophistications supplémentaires ci−dessous pour des plus de détails sur le dessin du curseur. Nous pourrions aussi donner les valeurs *GDK_EXTENSION_EVENTS_ALL* si nous voulons dessiner notre propre curseur, ou `GDK_EXTENSION_EVENTS_NONE` pour revenir à la situation par défaut.

Toutefois, nous ne sommes pas complètement à la fin de l'histoire. Par défaut, aucun périphérique d'extension n'est autorisé. Nous avons besoin d'un mécanisme pour que les utilisateurs puissent autoriser et configurer leur extensions. GTK dispose du widget *InputDialog* pour automatiser cette tâche. La procédure suivante gère un widget InputDialog. Elle crée le dialogue s'il n'est pas présent et le place au premier plan sinon.

```
void
input_dialog_destroy (GtkWidget *w, gpointer data)
{
  *((GtkWidget **)data) = NULL;
}

void
```

```
create_input_dialog ()
{
  static GtkWidget *inputd = NULL;

  if (!inputd)
    {
      inputd = gtk_input_dialog_new();

      gtk_signal_connect (GTK_OBJECT(inputd), "destroy",
                          (GtkSignalFunc)input_dialog_destroy, &inputd);
      gtk_signal_connect_object (GTK_OBJECT(GTK_INPUT_DIALOG(inputd)->close_button)
                                 "clicked",
                                 (GtkSignalFunc)gtk_widget_hide,
                                 GTK_OBJECT(inputd));
      gtk_widget_hide ( GTK_INPUT_DIALOG(inputd)->save_button);

      gtk_widget_show (inputd);
    }
  else
    {
      if (!GTK_WIDGET_MAPPED(inputd))
        gtk_widget_show(inputd);
      else
        gdk_window_raise(inputd->window);
    }
}
```

(vous pouvez remarquer la façon dont nous gérons ce dialogue. En le connectant au signal "destroy", nous nous assurons que nous ne garderons pas un pointeur sur le dialogue après l'avoir détruit −− cela pourrait provoquer une erreur de segmentation).

InputDialog a deux boutons "Close" et "Save", qui n'ont pas d'actions qui leur sont assignées par défaut. Dans la fonction ci−dessus, nous associons à "Close" le masquage du dialogue et nous cachons le bouton "Save" car nous n'implantons pas la sauvegarde des options XInput dans ce programme.

Utiliser l'information supplémentaire d'un périphérique

Lorsque l'on a validé le périphérique, on peut simplement utiliser l'information supplémentaire des champs des structures d'événement. En fait, il est toujours prident d'utiliser ces informations car ces champs auront des valeurs par défaut judicieuses même lorsque les événements supplémentaires ne sont pas autorisés.

La seule modification consiste à appeler *gdk_input_window_get_pointer()* au lieu de *gdk_window_get_pointer*. Cela est nécessaire car *gdk_window_get_pointer* ne retourne pas l'information supplémentaire.

```
void gdk_input_window_get_pointer     (GdkWindow       *window,
                                       guint32         deviceid,
                                       gdouble         *x,
                                       gdouble         *y,
                                       gdouble         *pressure,
                                       gdouble         *xtilt,
                                       gdouble         *ytilt,
                                       GdkModifierType *mask);
```

Lorsque l'on appelle cette fonction, on doit préciser l'ID du périphérique ainsi que la fenêtre. Habituellement, on aura obtenu cet ID par le champ *deviceid* d'une structure d'événement. Cette fonction retournera une valeur cohérente lorsque les événements ne sont pas autorisés (dans ce cas, *event−>deviceid* aura la valeur *GDK_CORE_POINTER*).

La structure de base des gestionnaires d'événements de déplacement et de bouton pressé ne change donc pas trop −− il faut juste ajouter le code permettant de traiter l'information supplémentaire.

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  print_button_press (event->deviceid);

  if (event->button == 1 && pixmap != NULL)
    draw_brush (widget, event->source, event->x, event->y, event->pressure);

  return TRUE;
}

static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  gdouble x, y;
  gdouble pressure;
  GdkModifierType state;

  if (event->is_hint)
    gdk_input_window_get_pointer (event->window, event->deviceid,
                                  &x, &y, &pressure, NULL, NULL, &state);
  else
    {
      x = event->x;
      y = event->y;
      pressure = event->pressure;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, event->source, x, y, pressure);

  return TRUE;
}
```

On doit aussi faire quelquechose de cette nouvelle information. Notre nouvelle fonction *draw_brush()* dessine avec une couleur différente pour chaque *event->source* et change la taille du pinceau selon la pression.

```
/* Dessine un rectangle à l'écran, la taille dépend de la pression,
   et la couleur dépend du type de périphérique */
static void
draw_brush (GtkWidget *widget, GdkInputSource source,
            gdouble x, gdouble y, gdouble pressure)
{
  GdkGC *gc;
  GdkRectangle update_rect;

  switch (source)
    {
    case GDK_SOURCE_MOUSE:
      gc = widget->style->dark_gc[GTK_WIDGET_STATE (widget)];
      break;
    case GDK_SOURCE_PEN:
      gc = widget->style->black_gc;
      break;
    case GDK_SOURCE_ERASER:
      gc = widget->style->white_gc;
      break;
```

```
    default:
      gc = widget->style->light_gc[GTK_WIDGET_STATE (widget)];
    }

  update_rect.x = x - 10 * pressure;
  update_rect.y = y - 10 * pressure;
  update_rect.width = 20 * pressure;
  update_rect.height = 20 * pressure;
  gdk_draw_rectangle (pixmap, gc, TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

En savoir plus sur un périphérique

Notre programme affichera le nom du périphérique qui a généré chaque appui de bouton. Pour trouver le nom d'un périphérique, nous appelons la fonction :

```
GList *gdk_input_list_devices               (void);
```

qui retourne une GList (un type de liste chaînée de la bibliothèque glib) de structures GdkDeviceInfo. La structure GdkDeviceInfo est définie de la façon suivante :

```
struct _GdkDeviceInfo
{
  guint32 deviceid;
  gchar *name;
  GdkInputSource source;
  GdkInputMode mode;
  gint has_cursor;
  gint num_axes;
  GdkAxisUse *axes;
  gint num_keys;
  GdkDeviceKey *keys;
};
```

La plupart de ces champs sont des informations de configuration que l'on peut ignorer sauf si l'on implante la sauvegarde de la configuration XInput. Celui qui nous intéresse est *name* qui est, tout simplement, le nom que X donne au périphérique. L'autre champ, qui n'est pas une information de configuration, est *has_cursor*. Si *has_cursor* est faux, on doit dessiner notre propre curseur, mais puisque nous avons précisé *GDK_EXTENSION_EVENTS_CURSOR*, nous n'avons pas à nous en préoccuper.

Notre fonction *print_button_press()* ne fait parcourir la liste retournée jusqu'à trouver une correspondance, puis affiche le nom du périphérique.

```
static void
print_button_press (guint32 deviceid)
{
  GList *tmp_list;

  /* gdk_input_list_devices retourne une liste interne, nous ne devons donc
    pas la libérer après */
  tmp_list = gdk_input_list_devices();

  while (tmp_list)
    {
      GdkDeviceInfo *info = (GdkDeviceInfo *)tmp_list->data;

      if (info->deviceid == deviceid)
```

```
            {
              printf("Bouton pressé sur le périphérique '%s'\n", info->name);
              return;
            }

          tmp_list = tmp_list->next;
        }
    }
```

Ceci termine les modifications de notre programme « XInputize ». Comme pour la première version, le code complet est disponible à l'endroit où vous avez obtenu ce didacticiel.

Sophistications supplémentaires

Bien que notre programme supporte maintenant XInput, il y manque des caractéristiques que l'on souhaite trouver dans les applications complètes. D'abord, l'utilisateur ne veut probablement pas avoir à configurer ses périphériques à chaque fois qu'il lance le programme et nous devons donc lui permettre de sauvegarder la configuration du périphérique. Ceci est réalisé en parcourant ce que retourne *gdk_input_list_devices()* et en écrivant la configuration dans un fichier.

Pour restaurer un état au prochain démarrage du programme, GDK dispose de fonctions pour changer la configuration des périphériques :

```
gdk_input_set_extension_events()
gdk_input_set_source()
gdk_input_set_mode()
gdk_input_set_axes()
gdk_input_set_key()
```

(La liste retournée par *gdk_input_list_devices()* ne doit pas être modifiée directement). Un exemple est donné dans le programme de dessin *gsumi* (disponible à l'adresse http://www.msc.cornell.edu/~otaylor/gsumi/). De plus, ce serait pratique d'avoir une méthode standard pour faire cela pour toutes les applications. Ceci appartient probablement à un niveau légèrement supérieur à GTK, peut−être dans la bibliothèque GNOME.

Une autre grosse omission que nous avons mentionnée plus haut est l'absence de dessin du curseur. Les plates−formes autres qu'XFree86 n'autorisent pas encore l'utilisation simultanée d'un périphérique comme pointeur de base et comme pointeur d'une application. Lisez le XInput−HOWTO pour plus d'informations là−dessus. Ceci signifie que les applications qui veulent atteindre le plus de monde possible doivent dessiner leur propre curseur.

Une application qui dessine son propre curseur doit faire deux choses : déterminer si le périphérique courant a besoin ou non d'un curseur dessiné et déterminer si le périphérique courant est à proximité (si celui−ci est une tablette de dessin, il est pratique de faire disparaître le curseur lorsque le stylet est en dehors de la tablette. Lorsque le périphérique voit le stylet, on dit qu'il est « à proximité »). La première vérification est faite en recherchant dans la liste des périphériques, comme nous l'avons fait pour obtenir le nom du périphérique. La deuxième est réalisée en choisissant des événements "proximity_out". Une exemple de dessin d'un curseur personnel est donné dans le programme *testinput* de la distribution GTK.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

21. Conseils pour l'écriture d'applications GTK

Cette section est simplement un regroupement de lignes de conduites générales et sages, ainsi que d'astuces pour créer des applications GTK correctes. Elle est totalement inutile pour l'instant car il ne s'agit que d'une phrase :)

Utilisez *autoconf* et *automake* de GNU ! Ce sont vos amis :) J'ai en projet de les présenter brièvement ici.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

22. Contributions

Ce document, comme beaucoup d'autres beaux logiciels, a été créé (NdT : et traduit) par des volontaires bénévoles. Si vous vous y connaissez sur certains aspects de GTK qui ne sont pas encore documentés, soyez gentils de contribuer à ce document.

Si vous décidez de contribuer, envoyez−moi (Ian Main) votre texte à `slow@intergate.bc.ca`. La totalité de ce document est libre et tout ajout que vous pourriez y faire doit l'être également. Ainsi, tout le monde peut utiliser n'importe quelle partie de vos exemples dans les programmes, les copies de ce document peuvent être distribuées à volonté, etc.

Merci.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

23. Remerciements

Je voudrai remercier les personnes suivantes pour leurs contributions à ce texte :

- Bawer Dagdeviren, `_chamele0n@geocities.com` pour le didacticiel sur les menus.
- Raph Levien, `_raph@acm.org` pour *bonjour tout le monde* à la GTK, le placement des widgets et pour sa sagesse. Il a aussi généreusement donné un abri à ce didacticiel.
- Peter Mattis, `_petm@xcf.berkeley.edu` pour le programme GTK le plus simple et pour sa capacité à le faire :)
- Werner Koch `_werner.koch@guug.de` pour la conversion du texte original en SGML, et pour la hiérarchie des classes de widget.
- Mark Crichton `_crichton@expert.cc.purdue.edu` pour le code de l'usine à menus et pour le didacticiel de placement des tables.
- Owen Taylor `_owt1@cornell.edu` pour la section sur le widget EventBox (et le

patch de la distribution). Il est aussi responsable des sections sur l'écriture de vos propres widgets GTK et de l'application exemple. Merci beaucoup à Owen pour toute son aide !

- Mark VanderBoom _mvboom42@calvin.edu pour son merveilleux travail sur les widgets Notebook, Progress Bar, Dialog et File selection. Merci beaucoup, Mark ! Ton aide a été très précieuse.
- Tim Janik _timj@psynet.net pour son beau travail sur le widget Lists. Merci Tim :)
- Rajat Datta _rajat@ix.netcom.com pour son excellent travail sur le didacticiel sur les Pixmaps.
- Michael K. Johnson _johnsonm@redhat.com pour ses infos et le code pour les menus.

Et à tous ceux d'entre vous qui ont commenté et aidé à améliorer ce document.

Merci.

Page suivante Page précédente Table des matières

Page suivante Page précédente Table des matières

24. Copyright

Éric Jacoboni. Toute remarque sur cette adaptation sera la bienvenue.

Merci à Joël Bernier pour avoir initié cette adaptation, et à Vincent Renardias pour sa relecture.

Page suivante Page précédente Table des matières