

# Interfaces Humain Machine



Introduction à la programmation d'IHM

# Les interfaces WIMP

---

Window, Icon, Mouse, Pull down menu

Popularisées par le Macintosh d'Apple, reprises par Windows, X Window, ...

## Histoire

Stanford (années 60)

PARC XEROX (années 70)

Apple (années 80)



# Les produits d'IHM

---

**Développer une IHM ex nihilo est économiquement insupportable**

**Cohérence et intégration nécessitent des outils de base communs**

- ▷ boîtes à outils de composants d'IHM (UI Toolbox)
  - ▷ écrites en C ou en langage non objet
- ▷ librairies de classes de composants d'IHM (OO UI Framework ou Application Framework)
  - ▷ écrites en langages objet (C++, java, C#, Objective-C, Swift, ...)
  - ▷ utilisent la boîte à outils du système d'exploitation

**Un composant d'IHM est aussi appelé :**

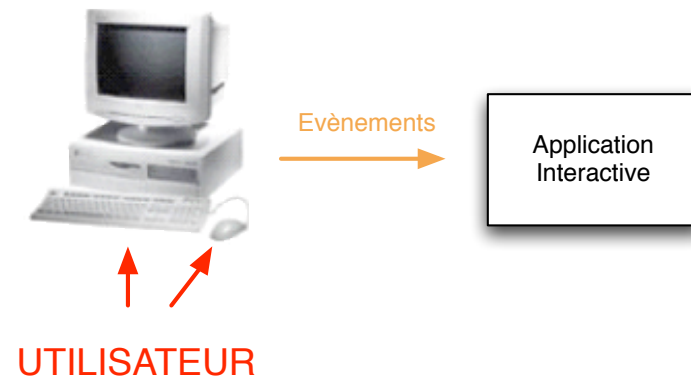
- ▷ Widget (Window Gadget)
- ▷ Contrôle
- ▷ Interacteur
- ▷ ...

# Aspects logiciels fondamentaux

---

# Programmation par événement

Un événement est un message envoyé par le système graphique à l'application à chaque action élémentaire de l'utilisateur



## Exemples (X Window)

- ▷ ButtonPress, ButtonRelease
  - ▷ Appui/relâchement d'un bouton de la souris
- ▷ KeyPress, KeyRelease
  - ▷ Appui/relâchement d'une touche du clavier
- ▷ MotionNotify
  - ▷ Déplacement de la souris bouton enfoncé
- ▷ Expose
  - ▷ Rafraîchissement de la fenêtre

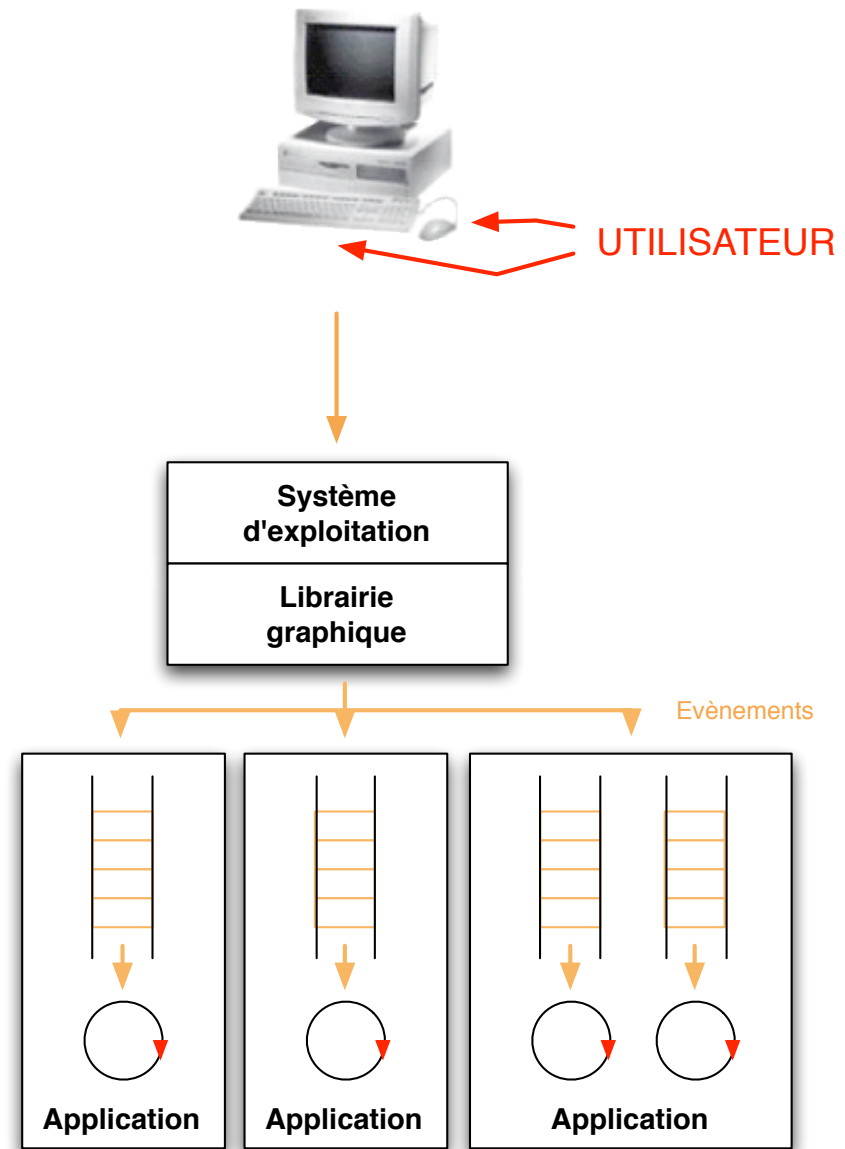
# Gestion des événements

## 1 - Création des composants

- ▷ Création des boutons, menus, fenêtres, ...

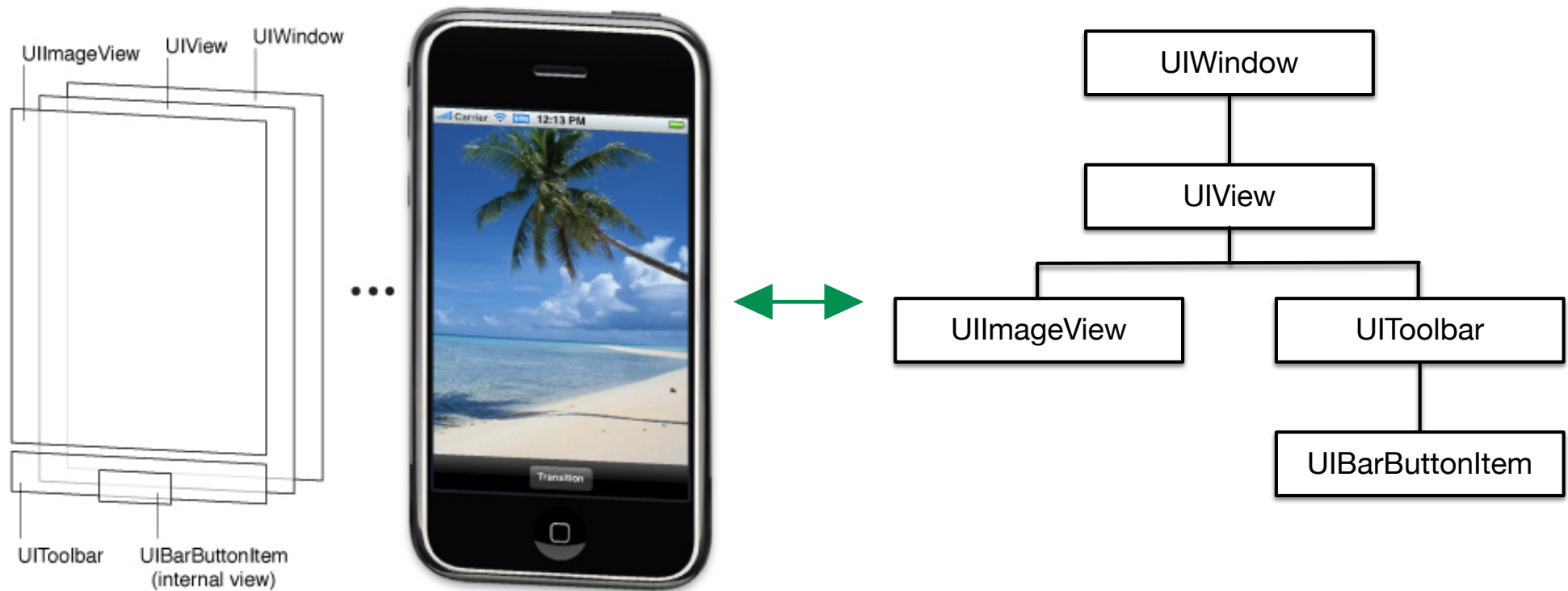
## 2 - Application mise en attente d'événements

- ▷ L'application devient « esclave » de l'utilisateur
- ▷ Elle comporte une **boucle de gestion des événements (Event Main Loop)**
- ▷ Boucle infinie qui :
  - ▷ Récupère les événements
  - ▷ Appelle les fonctions du programme



# Représentation logicielle d'une IHM

Une IHM est représentée de façon logicielle sous la forme d'un **arbre** de composants



référence : Apple

# Introduction succincte à GTK



La boîte à outils de composants d'IHM que vous allez utiliser



# GTK

---

## GIMP Toolkit

Boîte à outils multi-plateformes (Linux, Windows, MacOS, ...) comportant un ensemble complet de widgets pour la création d'interfaces utilisateur graphiques

Documentation et logiciel : [www.gtk.org](http://www.gtk.org)

# Installation de GTK+ 3

---

## Installer *pkg-config*

▷ *sudo apt install pkg-config*

## Vérifier que gtk+-3.0 est installé

▷ *dpkg -l libgtk\* | grep -e '^i' | grep -e 'libgtk-\*[0-9]'*

▷ *libgtk-3...*

## Sinon installer gtk+-3.0, voir le site :

▷ <https://developer.gnome.org/gtk3/stable/gtk-building.html>

## Installer la version développeur de gtk+-3.0

▷ *sudo apt-get install build-essential libgtk-3-dev*

## Vérifier l'installation

▷ *pkg-config --cflags gtk+-3.0*

▷ *-pthread -I/usr/include/gtk-3.0 ...*

# Créer une application GTK+

## Inclure le fichier d'entête GTK+

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

## Initialiser GTK+ dans la fonction *main* avec :

```
void gtk_init(int *argc, char **argv);
```

## Le programme principal, écrit dans un fichier *example.c*, est :

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

int main(int argc, char **argv)
{
    /* Initialisation of GTK+ */
    gtk_init(&argc, &argv);

    return EXIT_SUCCESS;
}
```

## Générer le programme

▷ `gcc `pkg-config --cflags gtk+-3.0` -o example example.c `pkg-config --libs gtk+-3.0``

# Widgets et notion d'héritage

Un composant GTK est nommé **Widget**

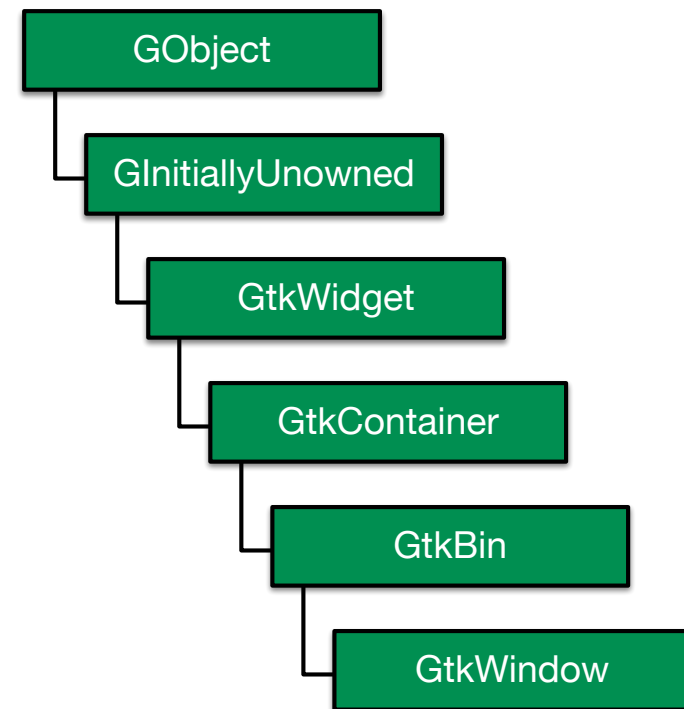
Un Widget possède des **propriétés** et un **comportement** donné par un ensemble de fonctions

Bien qu'écrit en langage C, GTK met en œuvre la notion d'**héritage** entre Widget

- ▷ un widget enfant dans l'arbre de spécialisation hérite des propriétés et du comportement du widget parent

**GtkWindow** est le widget qui définit une fenêtre

- ▷ il a ses propriétés et son comportement propre
- ▷ mais hérite des propriétés et du comportement de GtkWidget, ...



# Créer et afficher une fenêtre

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

int main(int argc, char **argv)
{
    /* Widget declaration */
    GtkWidget *window;
    /* Initialisation of GTK+ */
    gtk_init(&argc, &argv);

    /* Creation of main window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* Make window visible */
    gtk_widget_show(window);
    /* Window destruction */
    gtk_widget_destroy(window);

    return EXIT_SUCCESS;
}
```


tous les pointeurs vers des widgets quelles que soient leurs natures sont de type GtkWidget \*

le composant est créé en mémoire mais pas visible à l'écran

GTK\_WINDOW\_TOPLEVEL: fenêtre complète  
GTK\_WINDOW\_POPUP : fenêtre sans décoration (palette d'outils, ...)

la fenêtre apparaît à l'écran

le widget est détruit et la mémoire qu'il occupait récupérée. La fenêtre disparaît de l'écran

- ▷ Remarque : lorsque le programme est lancé, rien n'apparaît à l'écran. La fenêtre est détruite juste après sa création sans qu'aucun événement n'ait été traité.
- ▷  il faut traiter les événements

# Événements et signaux

---

Une action de l'utilisateur sur les périphériques d'entrée génère un **événement**

Suite à cet événement, le widget concerné émet un **signal**

▷ exemple : "destroy" lorsque l'utilisateur ferme la fenêtre

Chaque widget peut **émettre un ou plusieurs signaux** de nature différente

Pour traiter un signal, une **fonction callback** peut être associée à un signal et exécutée par la boucle de gestion des événements

La **signature** de la **fonction callback** est en général :

```
void fonction_de_traitement(GtkWidget *widget, gpointer data)
// widget : composant qui a émis le signal
// data : donnée supplémentaire
```

Pour **connecter** un signal à une fonction callback :

```
gulong g_signal_connect(gpointer *object, const gchar *name, GCallback func, gpointer
func_data);
// object : widget qui émet le signal (transtypé avec la macro G_OBJECT)
// name : nom du signal
// func : fonction callback associée au signal (transtypé avec la macro G_CALLBACK)
// func_data : donnée supplémentaire transmise à la fonction callback
```

# Événements

---

## Pour lancer la boucle de gestion des événements

```
void gtk_main(void);
```

- ▷ la boucle s'exécute alors à l'infini

## Pour arrêter la boucle de gestion des événements

- ▷ et permettre à l'application de se terminer

```
void gtk_main_quit(void);
```

**Il est possible de demander le traitement des événements en attente durant un calcul (hors de la boucle de gestion des événements)**

- ▷ voir la documentation sur Gtk

# Traitement de l'action de l'utilisateur sur la case de fermeture d'une fenêtre

---

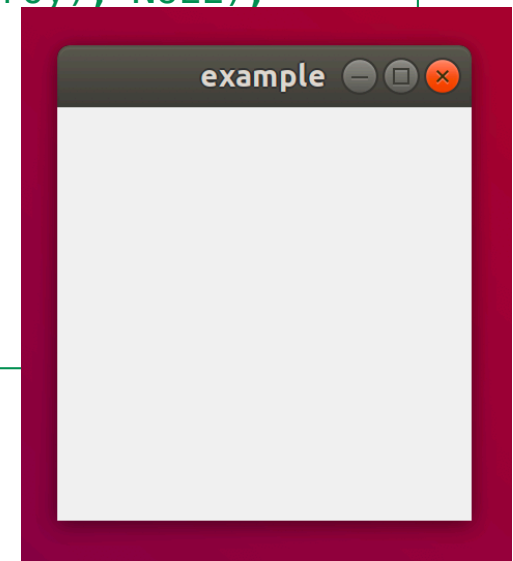
```
#include <stdlib.h>
#include <gtk/gtk.h>

void onDestroy(GtkWidget *widget, gpointer data)
{
    /* Halt main event loop */
    gtk_main_quit();
}

int main(int argc, char **argv)
{
    GtkWidget *window;
    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* Connection of signal named "destroy" */
    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(onDestroy), NULL);
    gtk_widget_show(window);
    /* Start main event loop */
    gtk_main();

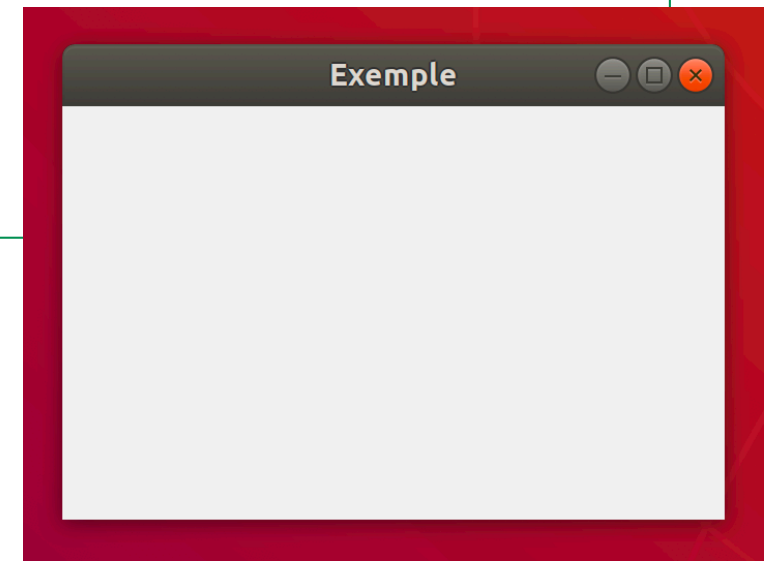
    return EXIT_SUCCESS;
}
```





# Modification de propriétés

```
...  
  
int main(int argc, char **argv)  
{  
    GtkWidget *window;  
    gtk_init(&argc, &argv);  
  
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);  
    /* Center window on screen */  
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);  
    /* Default window size */  
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);  
    /* Set window title */  
    gtk_window_set_title(GTK_WINDOW(window), "Exemple");  
    /* Connection of signal named "destroy" */  
    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(onDestroy), NULL);  
    gtk_widget_show(window);  
    gtk_main();  
  
    return EXIT_SUCCESS;  
}
```



# Ajout d'un label

```
...
int main(int argc, char **argv)
{
    GtkWidget *window;
    GtkWidget* label;
    /* Initialisation of GTK+ */
    gtk_init(&argc, &argv);

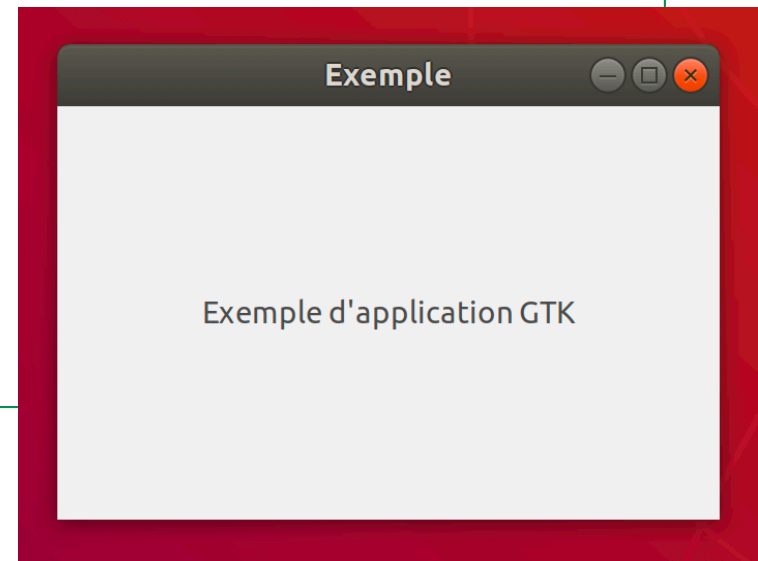
    /* Creation of main window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* Center window on screen */
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    /* Default window size */
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);
    /* Set window title */
    gtk_window_set_title(GTK_WINDOW(window), "Exemple");

    /* Creation of the label */
    label = gtk_label_new("Exemple d'application GTK");
    /* Insert the label in main window */
    gtk_container_add(GTK_CONTAINER(window), label);

    /* Connection of signal named "destroy" */
    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(onDestroy),
        NULL);

    /* Make window and all its content visible */
    gtk_widget_show_all(window);
    /* Start main event loop */
    gtk_main();

    return EXIT_SUCCESS;
}
...
```



# Label et attributs graphiques

GTK utilise un encodage UTF-8 et des balises semblables à celles de HTML pour prendre en compte du texte riche

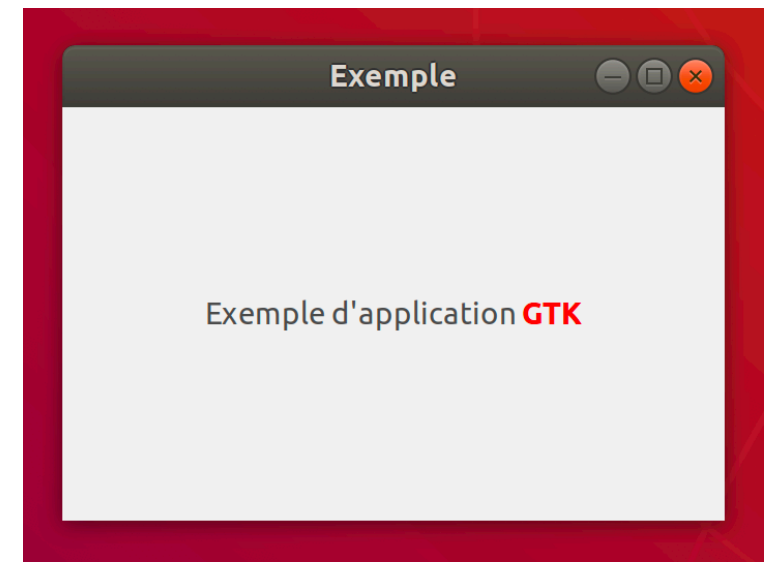
```
...  
/* Creation of the label */  
label = gtk_label_new(NULL);  
gchar* str = g_locale_to_utf8("Exemple d'application <span foreground=\"#FF0000\"><b>GTK</b></span>",  
                             -1, NULL, NULL, NULL);  
gtk_label_set_markup(GTK_LABEL(label), str);  
g_free(str);  
/* Insert the label in main window */  
gtk_container_add(GTK_CONTAINER(window), label);  
...
```

conversion de l'encodage de caractères du système en UTF-8

balises de formatage

affectation du texte balisé au label

libération de la mémoire allouée à la chaîne de caractères



# Disposition des composants enfants

---

Pour ajouter plusieurs composants enfants à la fenêtre, il est possible d'imbriquer des widgets de type **GtkBox**

- ▷ **GtkHBox** dispose les widgets enfants horizontalement
- ▷ **GtkVBox** dispose les widgets enfants verticalement

La fonction de création est :

```
GtkWidget* gtk_box_new(GtkOrientation orientation, gint spacing);  
// orientation horizontale si orientation = GTK_ORIENTATION_HORIZONTAL  
// orientation verticale si orientation = GTK_ORIENTATION_VERTICAL  
// spacing définit l'espace entre les widgets enfants
```

Les fonctions d'ajout de widgets enfants sont :

```
void gtk_box_pack_start(GtkBox* box, GtkWidget* child, gboolean expand, gboolean  
fill, guint padding);  
    // ajout de haut en bas ou de gauche à droite  
void gtk_box_pack_end(GtkBox* box, GtkWidget* child, gboolean expand, gboolean fill,  
guint padding);  
    // ajout de bas en haut ou de droite à gauche  
// les enfants insérés avec expand à vrai se partagent tout l'espace libre de la  
GtkBox  
// si fill est vrai, l'enfant occupe tout l'espace qui lui est réservé  
// padding définit espace autour de l'enfant
```

# Exemple de disposition verticale (début)

---

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

void onDestroy(GtkWidget *widget, gpointer data);

int main(int argc, char **argv)
{
    /* Widget declaration */
    GtkWidget *window;
    GtkWidget *vBox;
    GtkWidget* label;
    GtkWidget* message;
    /* Initialisation of GTK+ */
    gtk_init(&argc, &argv);

    /* Creation of main window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* Center window on screen */
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    /* Default window size */
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);
    /* Set window title */
    gtk_window_set_title(GTK_WINDOW(window), "Exemple");

    /* Creation of a vertical box */
    vBox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);
    /* Insertion of the box in the window */
    gtk_container_add(GTK_CONTAINER(window), vBox);
```

# Exemple de disposition verticale (fin)

```
/* Creation of the label */
label = gtk_label_new(NULL);
gchar* str = g_locale_to_utf8("Exemple d'application <span foreground=\"#FF0000\"><b>GTK</b></span>",
                             -1, NULL, NULL, NULL);
gtk_label_set_markup(GTK_LABEL(label), str);
g_free(str);

/* Insertion of the label in the box */
gtk_box_pack_start(GTK_BOX(vBox), label, FALSE, FALSE, 0);

message = gtk_label_new(NULL);
str = g_locale_to_utf8("<span foreground=\"#202020\">Un message</span>", -1, NULL, NULL, NULL);
gtk_label_set_markup(GTK_LABEL(message), str);
g_free(str);

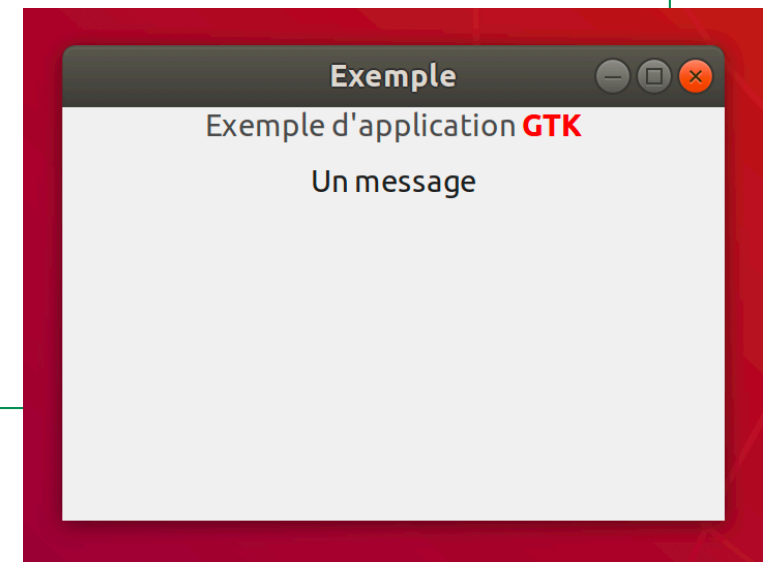
/* Insertion of the message in the box */
gtk_box_pack_start(GTK_BOX(vBox), message, FALSE, FALSE, 0);

/* Connection of signal named "destroy" */
g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(onDestroy), NULL);

/* Make window and all its content visible */
gtk_widget_show_all(window);
/* Start main event loop */
gtk_main();

return EXIT_SUCCESS;
}

void onDestroy(GtkWidget *widget, gpointer data)
{
    /* Halt main event loop */
    gtk_main_quit();
}
```



# Ajout d'un bouton de commande

```
...
GtkWidget* quitButton;
...
gtk_box_pack_start(GTK_BOX(vBox), message, FALSE, FALSE, 0);

GtkWidget *hBox = gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 0);

/* Creation of a button */
quitButton = gtk_button_new_with_label("Quitter");
/* Connection of signal named "clicked" */
g_signal_connect(G_OBJECT(quitButton), "clicked", G_CALLBACK(gtk_main_quit), NULL);
/* Insertion of the button in the box */
gtk_box_pack_start(GTK_BOX(hBox), quitButton, TRUE, FALSE, 0);
/* Insertion of hBox in vBox */
gtk_box_pack_start(GTK_BOX(vBox), hBox, FALSE, FALSE, 0);

/* Connection of signal named "destroy" */
g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(onDestroy), NULL);

/* Make window and all its content visible */
gtk_widget_show_all(window);
/* Start main event loop */
gtk_main();

return EXIT_SUCCESS;
}
...
```

le bouton est mis dans une boîte horizontale pour que ses dimensions idéales soient conservées

création d'un bouton (autres variantes possibles)

fonction callback (par exemple)

la boîte horizontale est imbriquée dans la boîte verticale



# Zone de texte éditable

Les fonctions de création sont :

```
GtkWidget* gtk_entry_new(void);  
GtkWidget* gtk_entry_new_with_max_length(gint max);  
    // max est le nombre maximale de caractères qui peuvent être saisis
```

Les fonctions de lecture ou de modification du texte contenu :

```
G_CONST_RETURN gchar* gtk_entry_get_text(GtkEntry *entry);  
void gtk_entry_set_text(GtkEntry *entry, const gchar *text);
```

▷ rq : la chaîne de caractères retournée ne doit pas être désallouée

La zone de texte émet un signal "**activate**" lorsque l'utilisateur appuie sur la touche **Entrée**

▷ la fonction callback a cette signature :

```
void user_function(GtkEntry *entry, gpointer user_data);
```



# Ajout d'une zone de texte éditable

```
...
/* Insertion of the message in the box */
gtk_box_pack_start(GTK_BOX(vBox), message, FALSE, FALSE, 0);

/* Creation of a text zone */
GtkWidget* entry = gtk_entry_new();
/* Connection to the signal named "activate" */
g_signal_connect(G_OBJECT(entry), "activate", G_CALLBACK(onActivateEntry), NULL);
/* Insertion in vBox */
gtk_box_pack_start(GTK_BOX(vBox), entry, FALSE, TRUE, 0);
...
}
...
void onActivateEntry(GtkEntry *entry, gpointer data)
{
    const gchar *text;

    /* Recuperation du texte contenu dans le GtkEntry */
    text = gtk_entry_get_text(GTK_ENTRY(entry));
}
```



# Dessin personnalisé dans GTK



# Dessin personnalisé

Le composant *GtkDrawingArea* permet le dessin personnalisé

La fonction de création est

```
GtkWidget *gtk_drawing_area_new()
```

La fonction de dessin est associée au signal "*draw*" de ce composant

Le dessin ne se fait **pas directement à la demande** du développeur, il faut indiquer à GTK qu'il est nécessaire de **rafraichir la surface** ou une partie de la surface du composant

▷ principe fondamental de la plupart des boîtes à outils/framework d'IU

La fonction *gtk\_widget\_queue\_draw()* ou ses variantes demande à GTK de rafraichir la vue d'un composant

Pour réaliser le dessin, GTK utilise la librairie graphique *Cairo*

Une surface de dessin est représentée par une variable de type *cairo\_surface\_t \**

# Cairo

Le dessin nécessite un **contexte** qui maintient des informations telles que la couleur, le tracé en cours, ...

Un **contexte de dessin** est représenté par une variable de type *cairo\_t* \*

Pour créer un contexte de dessin

```
cairo_t *cairo_create(cairo_surface_t *target)
```




Il faut détruire (déréférencer) le contexte lorsque le dessin a été réalisé pour libérer la mémoire

```
void cairo_destroy(cairo_t *cr)
```

Pour définir la **couleur** dans laquelle vont s'effectuer les tracés

```
void cairo_set_source_rgb(cairo_t *cr, double red, double green, double blue)
```

▷ red, green, blue : composantes couleur dans l'intervalle [0, 1]

▷ exemples : noir (0, 0, 0) , blanc (1, 1, 1) , rouge saturé (1, 0, 0) 

# Cairo

Le dessin s'effectue en utilisant des opérateurs de construction de **chemin** (tracé) puis en **remplissant** ce chemin ou en **traçant** son contour

**Pour créer un nouveau chemin**

```
void cairo_new_path(cairo_t *cr)
```

**Pour créer un sous-chemin et définir le point courant**

```
void cairo_move_to(cairo_t *cr, double x, double y)
```

**Pour ajouter un ligne au sous-chemin à partir du point courant**

```
void cairo_line_to(cairo_t *cr, double x, double y)
```

▷ le nouveau point courant est (x, y)

**Pour refermer le sous-chemin courant**

```
void cairo_close_path(cairo_t *cr);
```

▷ trace une ligne depuis le point courant jusqu'au premier point du sous-chemin

# Cairo

## Pour ajouter un rectangle au chemin

```
void cairo_rectangle(cairo_t *cr, double x, double y, double width, double height)
```

- ▷ (x, y) : coordonnées du coin haut-gauche du rectangle
- ▷ width : largeur du rectangle
- ▷ height : hauteur du rectangle

## Pour ajouter un arc de cercle au chemin

```
void cairo_arc(cairo_t *cr, double xc, double yc, double radius, double angle1,  
double angle2)
```

- ▷ (xc, yc) : coordonnées du centre du cercle
- ▷ radius : rayon du cercle
- ▷ angle1 : angle de départ en radians
- ▷ angle2 : angle d'arrivée en radians

# Cairo

## Pour ajouter un texte au sous-chemin

```
void cairo_show_text(cairo_t *cr, const char *utf8)
```

- ▷ utf8 : chaîne de caractères encodée en UTF-8
- ▷ le texte est dessiné à partir du point courant

## Pour définir la police de caractères

```
void cairo_select_font_face(cairo_t *cr, const char *family, cairo_font_slant_t  
slant, cairo_font_weight_t weight)
```

## Pour changer la taille de la police de caractères

```
void cairo_set_font_size(cairo_t *cr, double size)
```

# Cairo

Pour **remplir** le chemin courant avec la couleur courante de l'environnement

```
void cairo_fill(cairo_t *cr)
```

▷ après le dessin, le chemin est réinitialisé

Pour **tracer** le contour du chemin courant avec la couleur courante

```
void cairo_stroke(cairo_t *cr)
```

▷ après le dessin, le chemin est réinitialisé

Pour changer la **largeur** du contour (avant le tracé)

```
void cairo_set_line_width(cairo_t *cr, double width)
```

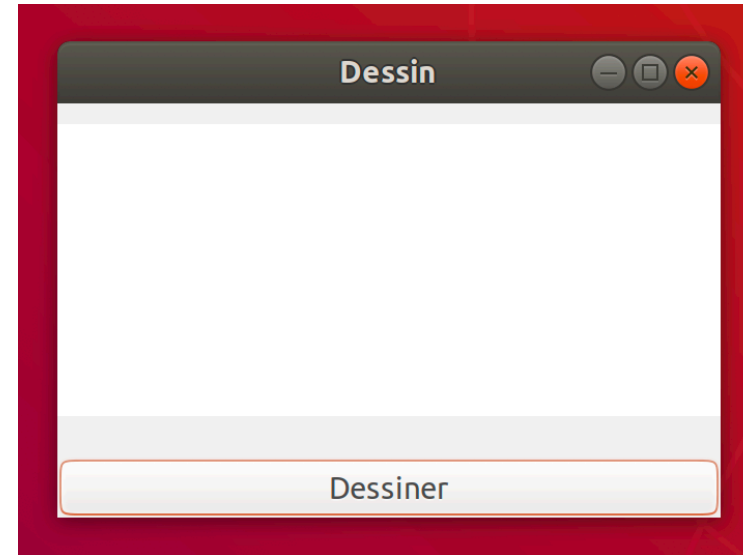
Pour **remplir toute la surface** avec la source courante (par ex, couleur)

```
void cairo_paint(cairo_t *cr)
```

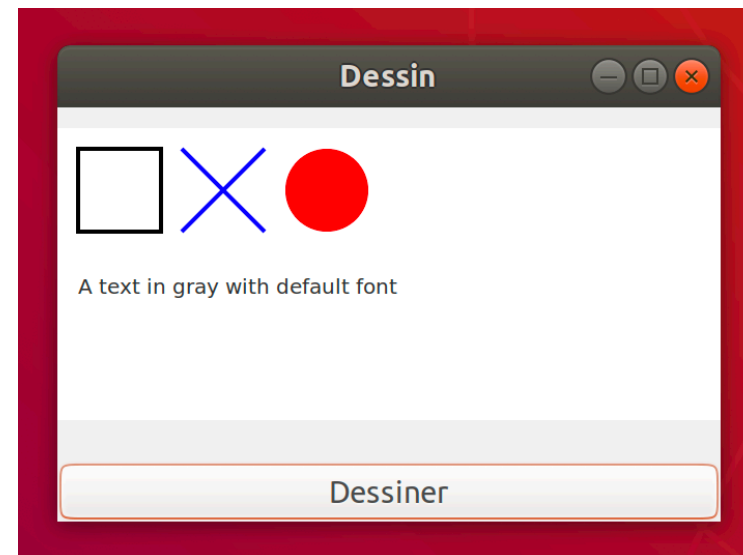


# Exemple de programme avec dessin personnalisé

Au lancement



Après appui sur le bouton



# Exemple de programme avec dessin personnalisé (1/5)

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

void on_destroy(GtkWidget *widget, gpointer data);
void button_cb(GtkWidget *widget, gpointer data);
gboolean draw_cb(GtkWidget *widget, cairo_t *cr, gpointer data);

int button_clicked;
int main(int argc, char **argv)
{
    /* Widget declaration */
    GtkWidget *window;
    GtkWidget *vBox;
    GtkWidget *button;
    GtkWidget *drawing_area;

    button_clicked = FALSE;

    /* Initialisation of GTK+ */
    gtk_init(&argc, &argv);

    /* Creation of main window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* Center window on screen */
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    /* Default window size */
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);
    /* Set window title */
    gtk_window_set_title(GTK_WINDOW(window), "Dessin");
```

← état de l'application ; l'utilisateur a-t-il cliqué sur le bouton ?

← le widget qui représente la zone de dessin

## Exemple de programme avec dessin personnalisé (2/5)

```
/* Creation of a vertical box */  
vBox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);  
/* Insertion of the box in the window */  
gtk_container_add(GTK_CONTAINER(window), vBox);
```

```
/* Creation of the drawing area */  
drawing_area = gtk_drawing_area_new();
```

← création du widget zone de dessin

```
/* Set a minimum size */
```

```
gtk_widget_set_size_request(drawing_area, 100, 100);
```

← demande d'une taille minimale

```
/* Insertion of the drawing area in the box */
```

```
gtk_box_pack_start(GTK_BOX(vBox), drawing_area, TRUE, TRUE, 10);
```

```
/* Creation of a button */
```

```
button = gtk_button_new_with_label("Dessiner");
```

```
/* Connection of signal named "clicked" */
```

```
g_signal_connect(G_OBJECT(button), "clicked", G_CALLBACK(button_cb), drawing_area);
```

```
/* Insertion of the button in the box */
```

```
gtk_box_pack_start(GTK_BOX(vBox), button, FALSE, FALSE, 0);
```

```
/* Signals used to draw the content of the widget */
```

```
g_signal_connect(drawing_area, "draw", G_CALLBACK(draw_cb), drawing_area);
```

```
/* Connection of signal named "destroy" */
```

```
g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(on_destroy), NULL);
```

```
/* Make window and all its content visible */
```

```
gtk_widget_show_all(window);
```

```
/* Start main event loop */
```

```
gtk_main();
```

```
return EXIT_SUCCESS;
```

```
}
```

↑ connexion des signaux du widget zone de dessin à des fonctions callback

# Exemple de programme avec dessin personnalisé (3/5)

```
void on_destroy(GtkWidget *widget, gpointer data)
{
    /* Halt main event loop */
    gtk_main_quit();
}

void button_cb(GtkWidget *widget, gpointer data)
{
    GtkWidget *drawing_area = data;
    button_clicked = TRUE;

    /* Now invalidate the affected region of the drawing area. */
    gtk_widget_queue_draw(drawing_area);
}

void clear(cairo_t *cr)
{
    /* Save current context */
    cairo_save(cr);

    cairo_set_source_rgb(cr, 1, 1, 1);
    cairo_paint(cr);

    /* Restore context */
    cairo_restore(cr);
}
```

fonction callback du bouton

le widget est passé via le paramètre data

l'état de l'application est modifié

on demande à la zone de dessin de se rafraichir

fonction qui efface le contenu de la zone

sauvegarde du contexte courant

la source est une couleur blanche

la source est dessinée dans tout l'intérieur de la zone

restauration du contexte sauvegardé

# Exemple de programme avec dessin personnalisé (4/5)

```
void draw(cairo_t *cr)
{
    /* Save current context */
    cairo_save(cr);

    /* Draw a rectangle */
    cairo_rectangle(cr, 10, 10, 40, 40);
    cairo_stroke(cr);

    /* Draw a cross in blue */
    cairo_move_to(cr, 60, 10);
    cairo_line_to(cr, 100, 50);
    cairo_move_to(cr, 100, 10);
    cairo_line_to(cr, 60, 50);
    cairo_set_source_rgb(cr, 0, 0, 1);
    cairo_stroke(cr);

    /* Fill a circle in red */
    cairo_arc(cr, 130, 30, 20, 0, 2 * M_PI);
    cairo_set_source_rgb(cr, 1, 0, 0);
    cairo_fill(cr);

    cairo_move_to(cr, 10, 80);
    cairo_set_source_rgb(cr, 0.2, 0.2, 0.2);
    cairo_show_text(cr, "A text in gray with default font");

    /* Restore context */
    cairo_restore(cr);
}
```

fonction qui dessine le contenu de la zone

sauvegarde du contexte courant

ajout d'un rectangle au chemin courant

tracé du contour du chemin avec la couleur par défaut (noir) et la largeur par défaut (2.0)

création de 2 sous-chemins

affectation de la couleur bleue

tracé des 2 sous-chemins

création d'un sous-chemin arc de cercle

affectation de la couleur rouge

remplissage du cercle

positionnement du texte

dessin du texte

restauration du contexte sauvegardé

# Exemple de programme avec dessin personnalisé (5/5)

```
/* Redraw the widget. Note that the ::draw
 * signal receives a ready-to-be-used cairo_t that is already
 * clipped to only draw the exposed areas of the widget
 */
gboolean draw_cb(GtkWidget *widget, cairo_t *cr, gpointer data) ← fonction callback appelée lors de la
{                                                                    création du widget zone de dessin
    GtkWidget *drawing_area = data;

    clear(cr); ← effacement de la zone de dessin
    if (button_clicked)
        draw(cr); ← dessin des formes si le bouton a
                    été cliqué

    return FALSE;
}
```

# Le sujet

---

# Le sujet

---

Créer un programme en **langage C** et fondé sur la bibliothèque **GTK** qui met en œuvre le jeu de dames

## Modularité

- ▷ Les fonctionnalités du jeu de dames (vérification des mouvements, détermination du mouvement, ...) doivent être séparées du code de l'Interface Humain-Machine



# Livrable

---

**Attention !!!**

**Ne cassez pas vos autres projets, réécrivez votre application dans un dossier séparé du reste.**

**Votre projet en GTK sur svn dans un dossier bien identifié et différencié du reste (dont le nom comporte gtk par exemple)**