TP1 et 1' : GDK et Glib

I. Premier programme

La librairie GDK permet de créer des fenêtres, gérer le graphisme et les évènements que recevront ces fenêtres durant leur affichage. Un programme utilisant cette librairie doit inclure `<gdk/gdk.h>` puis appeler la fonction `gdk_init` de la façon suivante :

```
#include <stdio.h>
#include <stdlib.h>
#include <gdk/gdk.h>

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

    printf ("Init gdk réussie\n");
    return 0;
}
```

Lors de l'appel de `gdk_init`, on lui passe les arguments de la ligne de commande ; les arguments reconnus sont retirés de la liste.

La fonction `gdk_init` réalise ensuite l'initialisation du mode graphique. Cette initialisation peut échouer ; dans ce cas, `gdk_init` affiche un message d'erreur et termine le programme.

1) Recopier le programme ci-dessus dans un fichier `a1-init.c`. Pour le compiler, taper la ligne suivante dans un terminal :

```
gcc -Wall a1-init.c -o a1-init `pkg-config gdk-2.0 --cflags --libs`
```

La commande `pkg-config` placée entre backquotes (``) permet de fournir les chemins propres à l'installation locale de GDK. Vous pouvez voir son résultat en tapant simplement dans le terminal :

```
pkg-config gdk-2.0 --cflags --libs
```

Vous pouvez aussi connaître le numéro de version de la librairie installée :

```
pkg-config gdk-2.0 --modversion
```

Si vous obtenez un message d'erreur à la compilation, il vous manque probablement des paquets. Pour les installer sur Ubuntu, il vous suffit de taper :

```
sudo apt-get install libc6-dev libgtk2.0-dev libgtk2.0-doc
```

Si vous êtes sous MacOS ou Windows, lire

```
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ez-draw-gtk/README
```

2) Tester ensuite le programme : `./a1-init` doit afficher un message de succès, tandis que l'appel `./a1-init --display toto:0` doit afficher une erreur (parfois au bout de quelques secondes) : cet argument permet de demander à GDK de se connecter à une machine distante (ici `toto`, qui n'existe pas) pour y réaliser l'affichage (ici sur le display numéro 0).

II. Makefile

Il est bon de prendre dès le départ l'habitude d'utiliser un `Makefile` pour compiler les programmes écrits en GDK. Recopier le `Makefile` suivant (sans oublier les tabulations et les ``) :

```
SHELL  = /bin/bash
CC     = gcc
RM     = rm -f
CFLAGS = `pkg-config gdk-2.0 --cflags` -Wall
LIBS   = `pkg-config gdk-2.0 --libs`
```

```
EXECS  = a1-init

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

all :: $(EXECS)

$(EXECS) : % : %.o
    $(CC) -o $@ $^ $(LIBS)

clean ::
    $(RM) *.o $(EXECS)
```

À l'utilisation, `make` ou `make all` doit tout compiler, tandis que `make clean` doit effacer les fichiers produits. Pour forcer à tout recompiler, taper `make clean all`.

Quelques explications : on récupère les `CFLAGS` et les `LIBS` par des appels à `pkg-config`. La macro `EXECS` contient la liste des exécutables que l'on veut produire, et que l'on complétera au fil des questions. Enfin, la triple dépendance `$(EXECS) : % : %.o` est une syntaxe propre à *GNU make*, elle permet de factoriser des dépendances de la façon suivante :

```
EXECS  = prog1 prog2 ... progn

$(EXECS) : % : %.o
    $(CC) -o $@ $^ $(LIBS)
```

est équivalent à

```
EXECS  = prog1 prog2 ... progn

prog1 : prog1.o
prog2 : prog2.o
...
progn : progn.o

$(EXECS) :
    $(CC) -o $@ $^ $(LIBS)
```

ou encore à

```
prog1 : prog1.o
    $(CC) -o prog1 prog1.o $(LIBS)

prog2 : prog2.o
    $(CC) -o prog2 prog2.o $(LIBS)
...

progn : progn.o
    $(CC) -o progn progn.o $(LIBS)
```

III. Première fenêtre

Plusieurs ingrédients sont nécessaires pour créer et afficher une fenêtre. On se donne les fonctions suivantes pour y parvenir : `creer_fenetre` et `boucle_principale`.

1) Commençons par recopier le programme précédent en `a2-win.c` Au début du `main`, déclarer la variable `GdkWindow *win1`. Elle permettra de mémoriser tout ce qui concerne la fenêtre.

Après l'initialisation de GDK, on crée une fenêtre (ici de taille 400 par 300 pixels) en appelant

```
win1 = creer_fenetre ("Titre de la fenetre", 400, 300);
```

2) À la fin du `main`, on appelle `boucle_principale ();`. Cette fonction est la "boucle de vie" de l'interface graphique, qui affiche les fenêtres, et distribue les évènements les concernant. On ne sortira de cette fonction qu'en faisant un `exit` lors d'un évènement choisi. (Remarque : lors de son initialisation, GDK a mis en place une fonction `atexit`, chargée de fermer proprement le programme ; on ne s'occupe donc de rien lors de la terminaison).

La fonction `boucle_principale` utilise des fonctions de la Glib, voir ci-dessous. On crée une variable `loop` pour mémoriser l'état de la boucle de vie (les paramètres de `g_main_loop_new` sont sans intérêt), puis on entre dans la boucle de vie proprement dite :

```
void boucle_principale ()
{
    GMainLoop *loop = g_main_loop_new (NULL, FALSE);
    g_main_loop_run (loop);
}
```

3) Il reste à écrire la fonction

`GdkWindow *creer_fenetre (char *titre, int largeur, int hauteur);`

Dans cette fonction, déclarer une variable `GdkWindowAttr attr`. C'est un `struct`, dont on initialise le champ `width` à `largeur`, `height` à `hauteur`, `window_type` à la constante `GDK_WINDOW_TOPLEVEL` (on veut créer une fenêtre principale) et enfin `wclass` à la constante `GDK_INPUT_OUTPUT` (c'est une fenêtre normale).

On peut maintenant créer la fenêtre `GdkWindow *win` en invoquant

`win = gdk_window_new (NULL, &attr, 0);`

Le premier paramètre signifie que son parent est la fenêtre racine (c'est-à-dire l'écran) ; le deuxième est la liste des attributs que l'on vient de renseigner ; le troisième signifie que l'on n'a pas fourni d'attribut optionnel.

On fixe ensuite le titre de la fenêtre par `gdk_window_set_title (win, titre);` puis on rend la fenêtre visible (par défaut elle est cachée) grâce à `gdk_window_show (win);` et enfin on renvoie `win`.

4) Le programme est maintenant achevé. Pour le compiler, rajouter `a2-win` à la fin de `EXECS = ...` dans le `Makefile` puis taper `make`. L'exécution doit ouvrir une fenêtre à l'écran avec le titre choisi. Observez que la fenêtre peut être déplacée et retaillée, mais que le bouton "fermeture" est sans effet pour le moment.

IV. Affichage des évènements

L'interface graphique que l'on est en train de construire doit être capable de réagir aux évènements provoqués par l'utilisateur (clic de souris, touche du clavier, etc) ainsi qu'aux évènements automatiques du gestionnaire de fenêtre (redessin de la fenêtre, etc).

1) Dans ce but, on recopie le programme précédent en `a3-event.c` puis on rajoute la fonction :

```
void on_event (GdkEvent *ev, gpointer data)
{
    printf ("Évènement %d reçu\n", ev->type);
}
```

On signale ensuite à GDK que notre fonction doit être appelée par `g_main_loop_run` lors de chaque évènement. Il suffit de rajouter dans `boucle_principale` avant `g_main_loop_run` ceci :

`gdk_event_handler_set (on_event, NULL, NULL);`

La fonction donnée en premier paramètre s'appelle un *event handler*, en français un *gestionnaire d'évènements*. Les deux derniers paramètres seront vus plus tard.

Enfin, on rajoute `gdk_window_set_events (win, GDK_ALL_EVENTS_MASK);` dans `creer_fenetre` pour accepter tous les évènements.

2) Testons maintenant le programme (comme d'habitude, on rajoute ce qu'il faut dans le `Makefile`). On obtient une suite de lignes `Évènement .. reçu` avec un numéro d'évènement dans la console, d'une part au démarrage puis lors du survol ou de clics souris, de frappe du clavier, ou encore si l'on retaille la fenêtre ou l'on tente de la fermer.

À quoi correspondent ces numéros d'évènements ? À des constantes tels que `GDK_BUTTON_PRESS`, `GDK_EXPOSE`, etc, déclarées par le type énuméré `GdkEventType` dans le fichier `gdkevents.h`. Cherchez

le chemin de ce fichier à partir des informations affichées par

```
pkg-config gdk-2.0 --cflags
```

puis éditez le fichier, et lisez les commentaires situés avant la déclaration de `GdkEventType` : ils décrivent la signification de chaque évènement.

3) Dans notre fonction `on_event` nous allons afficher le nom de chaque évènement. On remplace le `printf` actuel par `printf ("%s\n", get_event_name (ev));` puis on écrit la fonction :

```
char *get_event_name (GdkEvent *ev)
{
    if (ev == NULL) return "NULL event";

    switch (ev->type) {
        case GDK_NOTHING : return "GDK_NOTHING";
        /* puis un case par évènement */

        default : return "unknown event";
    }
}
```

Pour compléter la liste des case, on se simplifie le travail en recopiant simplement cette commande bash dans le terminal :

```
ok=false;
while read a b ; do
    if [ "$a" = "GDK_NOTHING" ]; then ok=true
    elif [ "$b" = "GdkEventType;" ]; then break; fi
    if $ok ; then echo "        case $a : return \"$a\";"; fi
done < /usr/include/gtk-2.0/gdk/gdkevents.h
```

Remplacez le chemin de `gdkevents.h` si celui-ci n'est pas le bon. Insérez la liste des `case` dans votre programme. Enfin, recompilez le programme et observez le déluge d'évènements qu'il reçoit : c'est déjà plus clair !

V. Filtrage et traitement des évènements

En général dans un programme on n'est pas intéressé par tous les évènements ; c'est pourquoi on les filtre à l'aide d'un *masque d'évènements*. Les évènements n'appartenant pas au masque (il y a des exceptions) ne sont pas transmis au gestionnaire d'évènement.

1) Recopier le précédent programme en `a4-decode.c`. Le masque est donné à l'aide de la fonction `gdk_window_set_events` dans `creer_fenetre`. Un masque est un OU binaire (opérateur `|`) entre des constantes du type énuméré `GdkEventMask` (voir dans le fichier `gdkevents.h`). Actuellement, le masque est `GDK_ALL_EVENTS_MASK`, ce qui signifie "accepter tous les évènements". Observez les évènements que l'on peut obtenir avec le masque :

```
GDK_EXPOSURE_MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK |
GDK_KEY_PRESS_MASK | GDK_KEY_RELEASE_MASK
```

On constate que certains évènements ne correspondant pas au masque apparaissent quand même (`GDK_CONFIGURE`, `GDK_DELETE`, etc), car ils ne sont pas filtrables.

2) On va maintenant traiter certains évènements, c'est-à-dire réagir lors de leur réception. Dans le gestionnaire d'évènement `on_event`, faire un `switch` sur `ev->type`, et repousser le `printf` actuel dans le cas par défaut : ainsi, seul les évènements non traités dans le `switch` seront affichés.

Vous aurez noté que le bouton fermeture de la fenêtre provoque l'évènement `GDK_DELETE`. Pour que cela ce traduise par la fermeture du programme, inserer dans le `switch` le cas `GDK_DELETE` et faire `exit(0);`. Tester.

3) Dans chaque évènement, un certain nombre d'informations sont enregistrées, informations qui dépendent du type d'évènement. C'est pourquoi le type `GdkEvent` est un type *union de structs* (dans `gdkevents.h`, chercher `typedef GdkEvent` puis `union _GdkEvent`).

Tous les sous-structures de cette union ont le champ `type` en premier, ce qui permet de consulter `ev->type` quelque soit l'évènement. En fonction de l'évènement, on doit utiliser le champ correspondant. Ainsi, pour l'évènement `GDK_BUTTON_PRESS`, on utilise le champ `button` de `ev`.

Pour l'évènement `GDK_BUTTON_PRESS` (bouton souris enfoncé), afficher le nom de l'évènement, suivi des coordonnées de la souris au moment du clic (champs `x` et `y` de `ev->button` ; les caster en `int`) puis du numéro du bouton (champ `button`). Tester, faire de même avec `GDK_BUTTON_RELEASE`.

4) L'évènement `GDK_KEY_PRESS` (touche clavier enfoncée) est un brin plus compliqué à décoder ; il faut consulter cette fois les champs de `ev->key`. Afficher le nom de l'évènement, puis le champ `string` (il contient la chaîne de caractères générée par la touche, en général une seule lettre), le champ `length` (la longueur de la chaîne, elle peut être 0), enfin le symbole de la touche.

Q'est-ce que le symbole d'une touche ? C'est une constante définie dans `<gdk/gdkkeysyms.h>` (inclure ce fichier et l'étudier un peu). Par exemple `GDK_q` est le symbole de la touche `'q'`, `GDK_Escape` celui de la touche échappement. La valeur entière du symbole est dans `ev->key.keyval`. La fonction `gdk_keyval_name` prend cette valeur en paramètre et renvoie le nom de la constante, privée du préfixe `GDK_`. Utiliser ceci pour que les noms des symboles de touches soit affichés dans le terminal lorsque des touches sont enfoncées.

Pour finir le traitement de `GDK_KEY_PRESS`, on fait un `switch` sur `ev->key.keyval`, dans lequel, pour les symboles `GDK_q` et `GDK_Escape`, on termine le programme.

VI. Premiers dessins

L'évènement `GDK_EXPOSE` signifie que le programme doit dessiner ou redessiner tout l'intérieur de la fenêtre. GDK propose de nombreuses fonctions de dessin. Les coordonnées utilisées sont les mêmes coordonnées que pour les évènements souris : le coin en haut à gauche de l'intérieur de la fenêtre est l'origine, les x sont vers la droite, les y vers le bas.

1) Recopier le programme précédent en `a5-maison.c` ; déclarer une fonction `void redessiner_win (GdkWindow *win)` dans laquelle on affiche `"redessiner_win"`. Appeler cette fonction pour l'évènement `GDK_EXPOSE`, en lui passant `ev->expose.window`. Vérifier que la fonction est bien appelée lors du démarrage, puis chaque fois que l'on déplace une autre fenêtre devant celle de notre programme.

2) Toutes les fonctions de dessin utilisent un certain nombre de paramètres (couleur du trait, épaisseur, pointillé, etc) qui sont stockés dans une variable appelée *contexte graphique*.

Dans `redessiner_win`, déclarer `GdkGC *gc;` puis créer un nouveau contexte graphique en faisant `gc = gdk_gc_new (win);`. À la fin de la fonction, on le libère par `g_object_unref (gc);`.

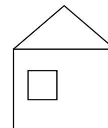
3) Les couleurs sont spécifiées en GDK par un triplet de couleurs primaires (rouge, vert, bleu) chacune sur 16 bits. Elles sont stockées dans un struct `GdkColor`, qui possède un quatrième champ `pixel` dans lequel est stocké la valeur (sur 24 ou 32 bits) qui sera envoyée à la carte graphique (c'est GDK qui la calculera pour nous). Pour changer la couleur courante de dessin on utilisera :

```
void set_color (GdkGC *gc, int r, int g, int b)  /* r,g,b entre 0 et 255 */
{
    GdkColor c;
    c.pixel = 0; c.red = r << 8; c.green = g << 8; c.blue = b << 8;
    gdk_gc_set_rgb_fg_color (gc, &c);
}
```

4) Pour dessiner un segment dans la fenêtre, on appelle
`gdk_draw_line (win, gc, x1, y1, x2, y2);`
où les quatre derniers paramètres sont les coordonnées du premier et dernier point, respectivement.

Dessiner un segment en rouge dans la fenêtre. Essayer ensuite avec une coordonnée en dehors de la fenêtre : vous constaterez que le dessin est *coupé* par la fenêtre, de façon à ne pas dépasser sur les autres fenêtres de l'écran. Ce *coupage automatique* est un principe fondamental de la notion de fenêtre, qui simplifie énormément la programmation.

5) Dessiner une maison rouge en vous inspirant du dessin ci-contre. Essayez d'autres couleurs. Faites changer de couleur parmi une liste chaque fois que l'on appuie sur la barre d'espace.



VII. Un peu de ménage

Le gestionnaire d'évènement `on_event` est actuellement tout d'un bloc, et il risque de devenir très long, rendant le programme assez lourd à lire. C'est pourquoi on va le subdiviser avec une fonction par évènement.

1) Recopier le précédent programme en `a6-subev.c`, puis remplacer le corps de `on_event` par

```
void on_event (GdkEvent *ev, gpointer data)
{
    switch (ev->type)   /* On fait un aiguillage selon le type d'évènement */
    {
        case GDK_EXPOSE       : on_expose       (&ev->expose, data); return;
        case GDK_DELETE       : on_delete       (&ev->any   , data); return;
        case GDK_BUTTON_PRESS : on_button_press (&ev->button, data); return;
        case GDK_KEY_PRESS    : on_key_press    (&ev->key   , data); return;
        default : printf ("%s\n", get_event_name (ev));
    }
}
```

2) Ensuite, réaliser chacune des fonctions d'évènement :

```
void on_expose (GdkEventExpose *e, gpointer data)
{
    redessiner_win (e->window);
}

void on_delete (GdkEventAny *e, gpointer data)
{ ... }

void on_button_press (GdkEventButton *e, gpointer data)
{ ... }

void on_key_press (GdkEventKey *e, gpointer data)
{ ... }
```

Le corps de la première fonction est déjà donné ci-dessus ; déplacer les blocs de code de votre ancienne fonction `on_event` dans chacune des nouvelles fonctions. Vous constaterez qu'il n'y a plus besoin d'écrire `ev->expose.window` mais `e->window`, plus simplement. N'oubliez pas de faire la même simplification sur le code déplacé.

Où trouver de la documentation ?

Vous pouvez approfondir ce que nous avons vu avec le manuel de référence de GDK :
`http://library.gnome.org/devel/gdk/stable/`

Les sections qui nous intéressent sont : *Windows, Events, Event structures, Drawing Primitives.*

TP2 et 2' : Bézier en GDK

Dans cet énoncé, on va apprendre à gérer une liste de sommets à la souris, puis dessiner une courbe de Bézier cubique définie par ces sommets, et enfin une courbe de Bézier par morceaux (encore appelée courbe B-spline cubique uniforme).

I. Préparation

1) On part de la fin du TP1 : créer un nouveau répertoire (par exemple `TP2`), recopier le dernier programme du TP 1 en `b1-info.c` ainsi que le `Makefile` dans `TP2`, modifier la macro `EXECS`.

Créons un module appelé `ig-util` (`ig` pour `interface graphique`), dans lequel on rangera nos fonctions utilitaires. Dans le fichier `ig-util.h` on met une *garde* :

```
#ifndef IG_UTIL__H
#define IG_UTIL__H

#endif /* IG_UTIL__H */
```

puis on déclare le prototype `char *ig_get_event_name (GdkEvent *ev);` à l'intérieur.

Dans le fichier `ig-util.c` on inclut `<gdk/gdk.h>` et `"ig-util.h"`, puis on déplace la fonction `get_event_name` de `b1-info.c` vers `ig-util.c` en la préfixant par `ig_`.

Maintenant dans `b1-info.c`, on inclut `"ig-util.h"`, et on met à jour le nom de la fonction là où elle est appelée. Enfin, on rajoute `ig-util.o` dans le `Makefile` de façon à avoir :

```
$(EXECS) : % : %.o ig-util.o
```

Compiler et vérifier l'absence de warnings.

2) Déclarer un struct `Info` avec pour le moment un champ `int dummy`. Ce type va nous permettre de mémoriser toutes les données du programme (coordonnées des sommets, etc) sans faire usage de variables globales.

Écrire la fonction `void info_init (Info *info)`, qui affiche `info_init` dans la console, puis initialise les champs de `info`. C'est ici que l'on fera des `malloc` si besoin ; pour le moment, initialiser le champ `dummy` à 123.

Écrire la fonction `void info_destroy (Info *info)`, qui affiche `info_destroy` dans la console. C'est là que l'on fera des `free` par la suite (un `free` pour un `malloc`, bien entendu).

Écrire la fonction `void quitter (Info *info)`, appelant `info_destroy (info)` puis `exit(0)`. Remplacer les `exit(0)` par `quitter (data)` dans `on_delete` et `on_key_press`. On peut déjà tester ici : lors de la fermeture de la fenêtre, on doit voir la trace `info_destroy` dans la console.

Maintenant dans le `main`, déclarer `Info info`. Après `gdk_init`, initialiser `&info`. Enfin, passer `&info` en paramètre de `boucle_principale`.

Il faut bien sûr adapter `boucle_principale` : on lui rajoute le paramètre `gpointer data`, puis on passe `data` en paramètre numéro deux de `gdk_event_handler_set` à la place de `NULL`. Quel est le type `gpointer` ? C'est un `void*`, type compatible avec toutes les adresses, sans avoir besoin de caster d'après la norme ANSI. À quoi sert `data` ? À mémoriser une donnée du programme (on dit souvent un *client data*) pour éviter d'avoir à la mettre en global.

Comment les fonctions du programme peuvent-elles recevoir `data` ? Le mécanisme est déjà en place : `data` est reçu en second paramètre du handler d'évènements `on_event`, puis transmis aux handlers spécialisés (`on_button_press`, etc) dans le `switch`.

Il nous reste à tester que tout fonctionne bien : au début du handler `on_button_press`, déclarer `Info *info = data;` (à partir de l'adresse générique `data`, on redonne le type `Info` pour pouvoir utiliser les champs), puis afficher la valeur du champ `dummy`. Si à l'exécution, un clic de souris fait afficher 123, le mécanisme fonctionne !

II. Polygone de contrôle

Le polygone de contrôle est une liste de sommets où chaque couple consécutif dans la liste est relié par une arête (c'est-à-dire un segment de droite).

1) Structures de données

Déclarer une constante `PMAX` valant 1000, et dans le struct `Info`, remplacer `dummy` par les champs entiers `*Px, *Py, Pn, Pmax, clic`. Les champs `Px` et `Py` seront des tableaux de taille `Pmax`. Les coordonnées des sommets seront mémorisées dans `Px[i]` et `Py[i]`, avec $0 \leq$ `i` $<$ `Pn` $\leq$ `Pmax`. Enfin, `clic` est le numéro du sommet sélectionné, par défaut -1.

Dans `info_init`, initialiser tous les champs : `Pmax` à `PMAX`, `Pn` à 3, `clic` à -1. Allouer les deux tableaux, et tester si l'allocation a réussi. Enfin, donner des coordonnées pour les 3 premiers points (ceci nous permettra de tester l'affichage).

Dans `info_destroy`, libérer les tableaux. Tester la compilation.

2) Affichage

Écrire la fonction `void dessin_sommet (GdkWindow *win, GdkGC *gc, int x, int y)` qui dessine un carré centré en `x,y` et de rayon 3 pixels. La documentation de `gdk_draw_rectangle` est dans `http://library.gnome.org/devel/gdk/stable/reference.html`, section *Drawing Primitives*.

Dans `redessiner_win`, supprimer le dessin actuel (la maison), puis commencer par mettre un fond blanc à la fenêtre. Pour cela, on récupère la taille actuelle de la fenêtre avec `gdk_drawable_get_size` (documentée aussi dans la section *Drawing Primitives*), on fixe la couleur à blanc puis on dessine un rectangle plein, de la taille de la fenêtre.

Dessiner ensuite les arêtes du polygone de contrôle en vert ; tester. Puis dessiner les sommets en bleu ; tester. Enfin, tester si le numéro de sommet sélectionné est valide (c'est-à-dire entre 0 et `Pn-1`) et si oui, dessiner le sommet sélectionné en rouge ; pour tester, initialiser `info->clic` par exemple à 1.

3) Sélection d'un sommet

Écrire la fonction `int chercher_clic (Info *info, int x, int y)` qui renvoie l'indice du premier sommet "proche" du point `x,y`, sinon -1. On dit que deux points sont proches si l'écart absolu pour chaque coordonnée est inférieur ou égal à 4.

Écrire la fonction `void reafficher (GdkWindow *win)` qui appelle

```
gdk_window_invalidate_rect (win, NULL, FALSE);
```

Cet appel aura pour effet d'envoyer un évènement `GDK_EXPOSE` à la fenêtre, et donc, de provoquer un redessin de la fenêtre "au bon moment" dans la boucle d'évènement.

Modifier `on_button_press` ainsi : si le bouton enfoncé est le numéro 1, on cherche le sommet proche du point cliqué et on le mémorise dans le champ `clic`, puis on demande le réaffichage en appelant `reafficher`.

Vous pouvez maintenant tester : cliquez sur chacun des sommets pour voir s'il s'affiche bien en rouge.

4) Déplacement d'un sommet

Il faut commencer par accepter les évènements `GDK_MOTION_NOTIFY` : dans `gdk_window_set_events`, rajouter `GDK_BUTTON_MOTION_MASK` au masque courant ; tester que les déplacements de la souris dans la fenêtre provoquent bien un déluge d'évènements `GDK_MOTION_NOTIFY`.

Écrire la fonction `void deplacer_sommet (Info *info, int i, int x, int y)` qui teste si `i` est un indice valide puis change en `x,y` les coordonnées du sommet numéro `i`.

Traiter l'évènement `GDK_MOTION_NOTIFY` dans `on_event` en branchant un handler

```
void on_motion (GdkEventMotion *e, gpointer data);
```

Dans `on_motion`, si un sommet est sélectionné, le déplacer puis demander le réaffichage. Tester.

5) Insertion d'un sommet

Écrire la fonction `int inserer_sommet (Info *info, int x, int y)` qui insère un nouveau sommet de coordonnées `x,y` à la fin de la liste des sommets. La fonction met à jour le nombre de sommets, puis renvoie l'indice du sommet inséré, ou -1 si la liste était pleine et que le sommet n'a pas pu être inséré.

Dans `on_button_press`, après la recherche du sommet sélectionné, si aucun sommet n'est sélectionné, insérer un nouveau sommet dont les coordonnées sont celles de la souris dans cet évènement.

Tester que un clic sur un sommet l'affiche en rouge, et que un clic ailleurs insère un nouveau sommet (affiché en rouge), relié à l'ancien dernier sommet.

6) Suppression de sommets

Écrire la fonction `void vider_sommets (Info *info)` qui met le champ `Pn` à 0 et `clic` à -1. Dans `on_key_press`, si la touche `v` est pressée, vider les sommets puis demander le réaffichage. Tester.

On se donne aussi la possibilité de supprimer un sommet donné. Écrire la fonction `void supprimer_sommet (Info *info, int i)` qui supprime le sommet d'indice `i` (si `i` est un indice valide) de la liste en tassant le tableau. La fonction met à jour le nombre de sommets et désélectionne tout sommet.

Écrire la fonction `void supprimer_sommet_ext (Info *info, int i, int larg, int haut)` qui supprime de la liste le sommet d'indice `i` (si `i` est un indice valide) si ce sommet est situé à l'extérieur du rectangle dont le coin supérieur gauche est `0,0` et le coin inférieur droit est `larg-1,haut-1`.

Traiter l'évènement `GDK_BUTTON_RELEASE` dans `on_event` en branchant un handler

```
void on_button_release (GdkEventButton *e, gpointer data);
```

Dans `on_button_release`, si le bouton numéro 1 est relaché, récupérer la largeur et la hauteur de la fenêtre, puis supprimer le sommet s'il est à l'extérieur de la fenêtre ; enfin, désélectionner tout sommet et provoquer le réaffichage.

Tester que si on tire un sommet hors de la fenêtre, il soit bien supprimé lorsqu'on lache le bouton.

III. Courbe de Bézier cubique

1) Écrire la fonction `double poly_bezier (int *B, double t)` qui reçoit un tableau `B` de (au moins) quatre entiers et un réel `t` entre 0 et 1. Le polynôme cubique de Bézier pour les coefficients B_i est $f(t) = (1-t)^3 B_0 + 3(1-t)^2 t B_1 + 3(1-t)t^2 B_2 + t^3 B_3$. La fonction calcule $f(t)$ et renvoie le résultat.

Pour calculer un point `x,y` de la courbe de Bézier pour le paramètre `t` et les sommets de contrôle (`Bx[0]`,`By[0]`), . . . , (`Bx[3]`,`By[3]`), l'appelant devra appeler deux fois cette fonction, en faisant :

`x = poly_bezier (Bx, t);` et `y = poly_bezier (By, t);` .

2) Écrire la fonction `void dessin_bezier (GdkWindow *win, GdkGC *gc, int *Bx, int *By, double pas)` qui trace la courbe de Bézier cubique pour les sommets de coordonnées (`Bx[0]`,`By[0]`), . . . , (`Bx[3]`,`By[3]`), en faisant un échantillonage de `pas` donné, c'est-à-dire pour `t = 0`, `t = pas`, `t = 2*pas`, . . . , `1`, et en reliant les points calculés par des segments.

Cela revient à calculer x_1 (pour $t = 0$), y_1 (pour $t = 0$), x_2 (pour $t =$ `pas`), y_2 (pour $t =$ `pas`), puis dessiner le segment (x_1, y_1, x_2, y_2) ; ensuite sauver x_2, y_2 dans x_1, y_1, puis calculer x_2 (pour $t = 2*$ `pas`), y_2 (pour $t = 2*$ `pas`) puis tracer le segment, etc, tant que $t < 1$.

Attention, à la fin de la boucle en t on tombe rarement exactement sur $t = 1$, c'est pourquoi après la boucle, on calcule le point $t = 1$ et on le relie au précédent point calculé.

3) Dans `redessiner_win`, s'il y a au moins 4 sommets, tracer en magenta la courbe de Bézier cubique pour les 4 premiers sommets avec un pas de $0,05$. Tester, et déplacer les sommets pour voir évoluer la courbe. Vérifier qu'elle est bien tangente à la première et troisième arête du polygone de contrôle, en respectivement le premier et quatrième sommet.

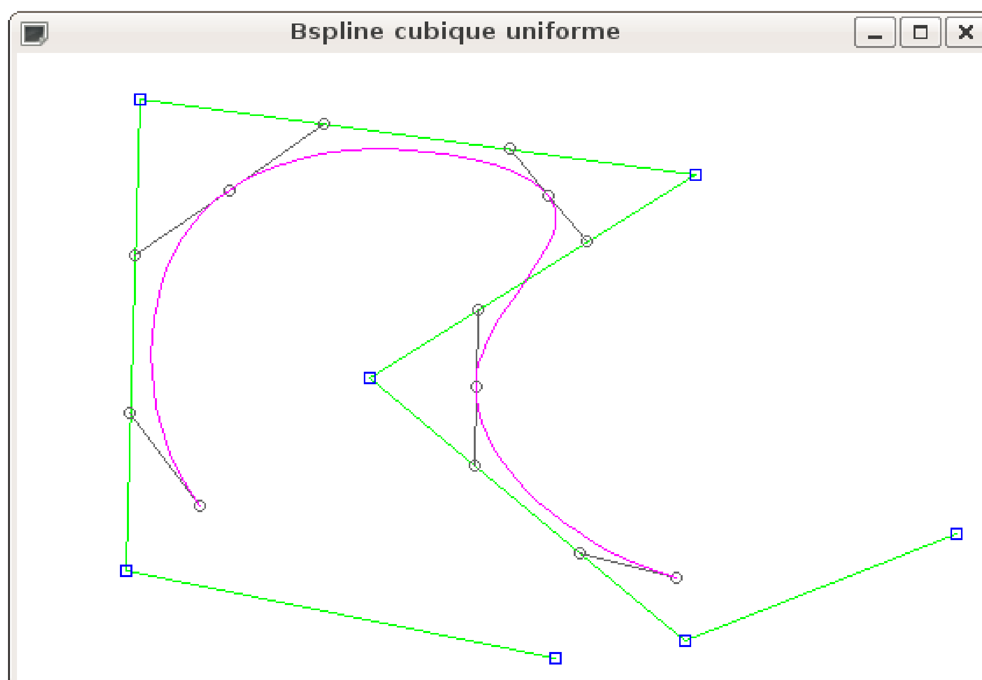
IV. B-spline cubique uniforme

Une B-spline cubique uniforme est un cas particulier des B-splines, car elle peut être décomposée en une suite de courbes de Bézier cubiques.

1) Écrire la fonction `void convertir_coords (int *P, int i, int *B)` qui convertit les coordonnées des sommets de contrôle de B-spline `P[i]`, `P[i+1]`, `P[i+2]`, `P[i+3]`, en coordonnées de sommets de Bézier `B[0]`, `B[1]`, `B[2]`, `B[3]`, en appliquant les formules : $B_0 = (P_i + 4P_{i+1} + P_{i+2})/6$, $B_1 = (2P_{i+1} + P_{i+2})/3$, $B_2 = (P_{i+1} + 2P_{i+2})/3$, $B_3 = (P_{i+1} + 4P_{i+2} + P_{i+3})/6$.

2) Écrire la fonction `void dessin_bspline (GdkWindow *win, GdkGC *gc, int *Px, int *Py, int Pn, double pas)` qui, pour les sommets de contrôle d'indice $0 \ldots 3$, puis $1 \ldots 4$, et ainsi de suite jusqu'à `Pn` $-4 \ldots$ `Pn` -1 :
- ▷ convertit les coordonnées des sommets de contrôle en coordonnées de sommets de Bézier (appeler `convertir_coords` pour les `Px` et encore une fois pour les `Py`) ;
- ▷ relie par un segment gris clair le premier et deuxième sommet de Bezier, idem avec le troisième et quatrième ;
- ▷ dans la même couleur, dessine en chaque sommet de Bezier un cercle centré sur lui et de rayon 3 (utiliser `gdk_draw_arc` avec un angle de départ de 0 et un angle final de $360 * 64$) ;
- ▷ enfin, dessine en magenta la courbe de Bézier.

3) Dans `redessiner_win`, remplacer le dessin de la courbe de Bézier par le dessin de la B-spline. Voici un exemple de ce que l'on devrait obtenir :



TP5 et 5' : Pong en GTK+

On se propose de réaliser un jeu de Pong pour expérimenter les animations en GTK+ et GDK.

I. Préparation

1) Récupérer sur internet le *tarball* suivant :
`http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/igra/tp5-v1.tgz`

2) Ouvrir un terminal et se placer dans le répertoire des TPs d'interface graphique, puis taper la commande : `tar xvfz` *chemin*`/tp5-v1.tgz`
en remplaçant *chemin* par le chemin correct (probablement `~/Bureau` ou `~/Téléchargements`).

3) Examinons la structure des répertoires générés. Le répertoire `TP5-pong` contient un `Makefile` général pour toutes les futures versions du programme, un sous-répertoire `V1` contenant la version V1 du programme, et un sous-répertoire `V6` contenant des fichiers qui seront utiles pour la dernière question.
Le sous-répertoire `V1` contient un `Makefile` permettant de calculer les dépendances (`make depend`) dans le fichier caché `.depend`, de tout compiler (`make all`) et de tout nettoyer (`make clean`).

4) Les sources du sous-répertoire `V1` sont découpés en plusieurs fichiers `.c` et `.h`, avec des noms de fonctions et des types proches de ceux employés dans la planche du TP4 (à l'exception des menus, des *frames* et des *listeners*, que l'on pourra rajouter plus tard si besoin).

Éditez et examinez ces fichiers par exemple en tapant : `gedit Makefile *.c *.h &`

5) Compilez le programme et testez-le. La zone de dessin (en jaune très pâle) est capable de recevoir les évènements du clavier, mais encore faut-il lui donner le *focus* clavier. Par défaut c'est le premier bouton (Quit) qui le possède ; il suffit de taper sur la touche [Tab] pour passer le focus au widget suivant, qui est la zone de dessin. Presser ensuite des touches et constatez la réception des évènements clavier. Par la suite on automatisera l'acquisition du focus clavier.

6) Créer une version V2 du programme. Pour ce faire, dans le terminal, aller dans le répertoire `TP5-pong` puis taper : `cp -a V1 V2 && cd V2`
puis fermer votre éditeur de texte, enfin rouvrez-le par exemple en tapant :
`gedit Makefile *.c *.h &`
On créera ainsi une nouvelle version du programme pour chaque section de cette planche. Par la suite, lorsque l'on voudra faire une sauvegarde générale, il suffira d'aller dans le répertoire `TP5-pong`, de taper `make clean` (ce qui déclenchera le nettoyage dans chaque sous-répertoire de version), puis d'aller dans le répertoire parent (`cd ..`), puis de taper : `tar cvfz tp5-v`*xy*`.tgz TP5-pong`
et enfin d'archiver ce *tarball* sur votre clé USB.

II. Gestion du temps

Pour animer la balle, les raquettes et l'affichage, nous allons utiliser des *timeout*, au moyen de la fonction `g_timeout_add` documentée dans la section "The Main Event Loop" du manuel de GLib.

1) Rajouter un champ entier `anim` (valeurs possibles : 0 pour animation stoppée, 1 pour animation en cours) dans le type `Info` et l'initialiser à 0 dans `info_init_default`.

Rajouter un bouton "Start" dans la *top box* de la fenêtre, et relier le signal `"clicked"` au *handler* `on_start_clicked`.

Dans ce *handler*, récupérer `Gui *g = gui_check (data);` puis `Info *info = g->info`. Faire la négation de `info->anim`, puis selon la nouvelle valeur (1 ou 0), remplacer le nom du bouton par "Stop" (respectivement "Start") ; tester.

2) Déclarer un champ `gint timeout1` dans le struct `Info` ; initialiser ce champ à 0 dans `info_init_default` (0 signifie : pas de *timeout* en cours ; $k > 0$: *timeout* en cours, de numéro (ou encore *handle*) k).

Dans le fichier `info.c`, section `/*-- Animation --*/`, écrire la fonction `void info_anim_start (Info *info, gpointer data)` (ne pas oublier de déclarer cette fonction dans `info.h`). Dans cette fonction, déclencher un *timeout* en appelant `g_timeout_add` et en mémorisant le *handle* dans `info->timeout1`. Par sécurité, ne pas déclencher ce *timeout* si `info->timeout1` est différent de 0 (car cela signifie qu'un *timeout* est déjà en cours). La durée du *timeout* est une constante `TIMEOUT1` initialisée à 500 ms, la *callback* est `on_timeout1`, et on transmet aussi `data`.

Écrire `void info_anim_stop (Info *info)`. Dans cette fonction, supprimer le *timeout* en cours à l'aide de `g_source_remove` en lui passant `info->timeout1`, puis en mettant `info->timeout1` à 0. Par sécurité, ne pas supprimer le *timeout* si `info->timeout1` vaut déjà 0.

Dans la *callback* `gboolean on_timeout1 (gpointer data)`, afficher un message puis renvoyer `TRUE` (ceci réarme le *timeout* avec le même délai).

Dans `on_start_clicked`, si `info->anim` vaut 1, appeler la fonction `info_anim_start`, sinon appeler `info_anim_stop`.

3) Tester : le bouton Start déclenche l'affichage d'un message toutes les demi-secondes, le bouton Stop arrête le mécanisme. Si en cliquant plusieurs fois très vite sur le bouton, vous voyez apparaître plusieurs messages à la fois toutes les demi-secondes, cela voudra dire que plusieurs *timeout* sont à l'œuvre simultanément, et donc que vous avez mal géré `info->timeout1` dans `on_start_clicked`.

III. Mouvements de la balle

1) Faire une version V3 du programme, quitter et relancer l'éditeur de texte dans le répertoire.

Déclarer dans `info.h` un struct `Ball` dont les champs sont les entiers `x`, `y` (coordonnées du centre de la balle), `dx`, `dy` (déplacement élémentaire de la balle lors d'une étape), et `r` (rayon de la balle). Rajouter un champ `Ball ball` dans `Info`.

Rajouter une section `/*-- Ball --*/` dans `info.c`. Écrire la fonction `void ball_init (Info *info)` qui initialise les champs de `info->ball` de telle sorte que la balle fasse 15 pixels de rayon et qu'elle soit située dans le coin en haut à gauche du *drawing area*. Les déplacements élémentaires `dx` et `dy` sont initialisés à 2 pixels. Pour simplifier l'écriture, on peut déclarer `Ball *b = &info->ball;` au début de la fonction puis travailler avec `b`. Enfin, appeler `ball_init` dans `info_init_default`.

Écrire `void ball_draw (GdkWindow *win, GdkGC *gc, Info *info)` qui dessine en rouge un cercle représentant la balle. Appeler `ball_draw` dans `on_area_expose` juste après le dessin du fond. Vérifier que la balle s'affiche au bon endroit.

2) Rajouter dans `Info` les champs entiers `area_width` et `area_height` pour mémoriser la taille du *drawing area*. Les initialiser à 0 dans `info_init_default`. Rajouter `GDK_EXPOSURE_MASK` dans le masque des évènements du *drawing area*. Connecter le signal `"configure_event"` au *handler* `gboolean on_area_configure (GtkWidget *area, GdkEventConfigure *e, gpointer data)`. Dans ce dernier, afficher dans la console la nouvelle taille `e->width` et `e->height` du *drawing area* et la mémoriser dans `info`. Vérifier en déformant la fenêtre que la taille s'affiche.

3) Écrire la fonction `void ball_next_step (Info *info)` qui calcule les nouvelles coordonnées de la balle. Il suffit d'incrémenter `x` de `dx` et `y` de `dy`. Pour donner l'illusion d'un rebond contre les

bords du *drawing area*, il faut tester si la balle sort en x (respectivement en y), puis si c'est le cas, placer la balle exactement au bord du *drawing area* (en x ou en y) et enfin inverser le signe de dx (respectivement de dy). Tester séparément tout en x puis tout en y simplifie beaucoup l'écriture !

4) On va procéder maintenant à l'animation générale. Implémenter un second *timeout* sur le modèle du premier en dupliquant tout le code. Changer le délai TIMEOUT1 à 30 ms et le délai TIMEOUT2 à 10 ms. Vérifier que dans la console, la *callback* on_timeout2 affiche en moyenne 3 fois plus souvent que on_timeout1.

Dans util.c, écrire une fonction void area_redraw (GtkWidget *area) qui appelle gdk_window_invalidate_rect sur area->window dans le but de lui faire parvenir un évènement GDK_EXPOSE.

Dans on_timeout1, remplacer le printf par un appel à area_redraw sur g->area ; ceci provoquera environ 35 rafraîchissements par seconde, ce qui devrait être suffisant pour avoir une animation fluide. Enfin dans on_timeout2, remplacer le printf par un appel à ball_next_step. Vous devriez voir la balle rebondir contre les bords du *drawing area*, y compris si vous retaillez la fenêtre.

IV. Gestion des raquettes

1) Faire une version V4 du programme, quitter et relancer l'éditeur de texte.

Déclarer dans info.h un struct Racquet comprenant les champs entiers x, y (coordonnées du centre de la raquette) et rx, ry (rayon en x et en y). Rajouter les champs Racquet rac_left, rac_right dans Info.

Dans le fichier info.c rajouter une section /*-- Racquet --*/. Écrire la fonction void racquet_init (Racquet *rac) qui initialise les champs de rac avec x et y à 100, rx à 4 et ry à 35. Appeler racquet_init dans info_init_default pour chaque raquette.

Écrire la fonction void racquet_draw (GdkWindow *win, GdkGC *gc, Racquet *rac) qui dessine en bleu un rectangle non plein représentant la raquette ; l'appeler dans on_area_expose pour chaque raquette (à ce stade on n'en verra qu'une seule car elles possèdent les mêmes coordonnées).

2) Écrire la fonction void racquet_adapt_left (Racquet *rac, int width, int height) qui reçoit en paramètre la largeur et la hauteur du *drawing area* et modifie éventuellement les coordonnées de la raquette de telle façon qu'elle soit positionnée exactement à gauche du *drawing area*, et qu'elle n'en sorte pas ni en haut ni en bas. Faire de la même manière une fonction racquet_adapt_right.

Appeler ces deux fonctions dans area_on_configure pour les raquettes ; vérifier que dès le départ, les raquettes soit placées à gauche et à droite, et que si l'on retaille la fenêtre, elles restent bien au bord.

3) Procédons maintenant à l'animation des raquettes au moyen des touches du clavier. Il faut commencer par demander le *focus* clavier, c'est-à-dire demander à ce que le *drawing area* reçoive les évènements clavier (par défaut c'est le dernier bouton cliqué qui les reçoit). Il devrait être suffisant d'appeler gtk_widget_grab_focus sur le *drawing area* dans on_start_clicked pour le cas "Start", et aussi dans on_area_button_press.

Rajouter des champs entiers key_LT, key_LB, key_RT, key_RB dans Info (L,R,T,B signifient respectivement Left, Right, Top, Bottom). Les initialiser à 0 dans info_init_default. Lorsque la touche 'a' est enfoncée, mettre key_LT à 1 ; le remettre à 0 lorsque cette touche est relâchée. Procéder de même pour les touches 'q', 'p' et 'l' avec key_LB, key_RT, key_RB.

On connaît ainsi à tout moment l'état des touches 'a', 'q', 'p' et 'l' (sans avoir à se battre avec l'*auto-repeat* du clavier). Tester avec des printf pour connaître l'état des variables key_*.

4) Nous y somme presque. Écrire `void racket_next_step (Racquet *rac, int area_height, int key_T, int key_B);` qui calcule la prochaine position de la raquette. Si aucune touche n'est enfoncée on ne fait rien, de même si les deux touches sont dans l'état enfoncé. Sinon, on déplace la raquette dans le sens indiqué, par exemple de 6 pixels, et on vérifie que la raquette ne sort pas du *drawing area*.

Enfin, on appelle `racquet_next_step` dans `on_timeout2` pour chaque raquette. Vérifier que les raquettes se déplacent sans sortir (y compris lorsqu'on retaille la fenêtre), que le mouvement est immédiat, fluide, et suffisamment rapide par rapport à la balle.

V. Jeu de pong

Dans cette section nous rajoutons une interaction entre la balle et les raquettes, ainsi qu'un état pour gérer la situation du jeu. Faire une version V5 du programme, quitter et relancer l'éditeur de texte.

1) Déclarer dans `info.h` un énuméré avec les valeurs `S_BEGIN`, `S_REBOUND`, `S_LOST`, `S_WON` ; rajouter un champ `state` dans `Info`, initialisé à `S_BEGIN` dans `info_init_default`.

Dans le fichier `info.c` rajouter une section `/*-- State --*/`. Pour faire connaître l'état du jeu, on se donne une fonction `void state_show (Info *info, GtkWidget *statusbar)` qui, selon `info->state`, affiche dans le *status bar* respectivement : "Appuyez sur espace puis sur les touches a, q, p, l ...", "Jeu en cours ...", "C'est perdu !! Appuyez sur espace ...", "C'est gagné !! Appuyez sur espace ...". La fonction `void state_set (Info *info, int state, GtkWidget *statusbar)` est chargée de mémoriser le nouvel état dans `info->state` et de mettre à jour le message du *status bar*. (Remarque : on pourrait éviter de passer `statusbar` à ces fonctions au moyen de *listeners* ...)

Le bouton Start/Stop n'est pas sensé changer l'état du jeu ; pour autant, il doit mettre à jour le message du *status bar*. Pour être cohérent, on modifie `state_show` de telle sorte que le message affiché soit vide lorsque l'animation est suspendue.

2) Lorsque la touche espace est pressée : si l'état est `S_BEGIN` il devient `S_REBOUND` ; s'il est `S_LOST` ou `S_WON`, il devient `S_BEGIN`.

Écrire `void ball_prepare_service (Info *info)`. Cette fonction place la balle exactement au milieu du *drawing area*.

Dans `on_timeout2` on remanie le code de la façon suivante : on appelle `racquet_next_step` pour chaque raquette, puis : si l'état est `S_BEGIN` on appelle `ball_prepare_service` ; si l'état est `S_REBOUND` on appelle `ball_next_step`.

Vérifiez qu'au lancement du programme, la balle soit en haut à gauche ; dès que le bouton Start est pressé, la balle vient se placer au milieu du *drawing area* ; une fois la barre d'espace pressée, le jeu démarre et la balle se met à bouger.

3) On s'attaque maintenant au jeu proprement dit, et cela se passe dans `ball_next_step`. Les seul cas à modifier sont lorsque la balle arrive sur un bord horizontal : si la balle arrive sur une raquette, la balle rebondit ; sinon, l'état du jeu passe à `S_LOST` (la balle s'arrête alors, et il faut appuyer sur la barre d'espace pour recommencer ; ceci est déjà géré). Penser à rajouter `statusbar` en paramètre à `ball_next_step`.

VI. Décoration avec des images

1) Une fois le jeu fonctionnel, vous pourrez améliorer l'aspect général en calculant le score et en l'affichant en grand au milieu du *drawing area* (voir fichiers `V6/font.[ch]` du tarball).

2) Une autre amélioration souhaitable serait d'afficher une image de fond et des morceaux d'images à la place de la balle et des raquettes. GDK sait lire et afficher les images aux formats classiques ; en particulier, il sait gérer la transparence dans le format PNG.
Vous aurez besoin des fonctions suivantes, à placer dans votre module `util.c` :

```
#include <gdk-pixbuf/gdk-pixbuf.h>

void pixbuf_get_size (GdkPixbuf *pix, int *w, int *h)
{
    *w = gdk_pixbuf_get_width (pix); *h = gdk_pixbuf_get_height (pix);
}

GdkPixbuf pixbuf_load (const char *filename)
{
    GError *error = NULL; GdkPixbuf *pix; int w, h;

    printf ("Loading %s ... ", filename); fflush (stdout);
    pix = gdk_pixbuf_new_from_file (filename, &error);
    if (pix == NULL) {
        printf ("error\n");
        fprintf (stderr, "Unable to read file: %s\n", error->message);
        g_error_free (error); return NULL;
    }
    pixbuf_get_size (pix, &w, &h); printf ("%dx%d\n", w, h);
    return pix;
}

void pixbuf_draw (GdkWindow *win, GdkGC *gc, GdkPixbuf *pix, int x, int y)
{
    gdk_draw_pixbuf (win, gc, pix, 0, 0, x, y, -1, -1, GDK_RGB_DITHER_NONE, 0, 0);
}
```

3) Déclarer dans `Info` des champs pour mémoriser les images ; charger les images dans une fonction `void pix_load_all (Info *info)`, elle-même appelée au début de `info_init_default`. Ainsi, les fonctions d'initialisation de la balle et des raquettes doivent pouvoir s'adapter à la taille de leur image ; il en ira de même pour le *drawing area*, qui doit pouvoir s'adapter à la taille de l'image de fond (utiliser `gtk_widget_set_size_request`). L'idéal est que l'absence d'une image au démarrage du programme n'affecte pas son fonctionnement, mais revienne à l'affichage et la taille par défaut pour cet élément. Voici un exemple du jeu sans et avec images :