TP1 : premiers pas en GTK+

I. Préparation

1) Téléchargez depuis http://j.mp/optigra les transparents du cours, de manière à pouvoir copier-coller le code. Idem pour cette planche.

2) Compilez et exécutez les exemples, jusqu'à celui du bouton Quit.

3) Changez le titre de la fenêtre en "TP1 en GTK". Rajoutez des champs `win_width` et `win_height` dans le struct `Mydata` ; initialisez-les dans le `main` et utilisez-les dans `on_app_activate` pour donner la taille de la fenêtre.

4) Créez une fonction d'initialisation `void init_mydata (Mydata *my)`, que vous appelerez au début du `main` ; déplacez les affectations des champs de `my` figurant actuellement dans le `main` vers cette nouvelle fonction. Dorénavant toutes les initialisations de `my` iront dans cette fonction.

5) Il vous est conseillé à chaque question de copier le fichier de la question précédente et de le renommer, afin de ne pas perdre votre travail. Vous pouvez par exemple nommer `tp01-1.c` le fichier pour la question I., etc. Compilez et testez systématiquement.

II. Un peu de magie

Une des principale source de bugs est la signature des callbacks : vous verrez que selon le signal, les callbacks ont des nombres de paramètres assez différents. Or, la variable `my`, cruciale pour notre programme, est souvent placée en dernier paramètre : une erreur sur le nombre de paramètres ne fait pas recevoir la bonne adresse de `my` et vous expose à un plantage du programme, souvent difficile à trouver. Voici une méthode pour prévenir les erreurs.

1) Rajoutez au début du struct `Mydata` le champ `unsigned int magic` ; déclarez la constante `#define MYDATA_MAGIC 0x46EA7E05` ; au début de `init_mydata`, affectez `magic` à `MYDATA_MAGIC`.

2) Recopiez la fonction suivante (téléchargez le pdf puis copiez-collez) :

```
// Cette fonction permet de tester si le data que l'on a recuperé dans
// une callback contient bien my ; sinon, cela veut dire que :
//  - soit on a oublié de transmettre my dans g_signal_connect,
//  - soit on s'est trompé dans le nombre de paramètres de la callback.

Mydata *get_mydata (gpointer data)
{
    if (data == NULL) {
        fprintf (stderr, "get_mydata: NULL data\n"); return NULL;
    }
    if (((Mydata *)data)->magic != MYDATA_MAGIC) {
        fprintf (stderr, "get_mydata: bad magic number\n"); return NULL;
    }
    return data;
}
```

3) Au début de chaque callback, remplacez `Mydata *my = data;` (ou `user_data`) par : `Mydata *my = get_mydata(data);` y compris pour `on_app_activate`

4) Testez que tout marche bien. Testez en remplaçant un `&my` par `NULL` dans la connection du signal `clicked` pour un bouton (vous devriez voir le message `"NULL data"` avant que le programme ne plante). Testez en rajoutant dans la signature de la callback d'un bouton les paramètres `int x, int y, int z` *avant* la déclaration de `data` (vous verrez peut-être `"bad magic number"` avant que le programme ne plante). Revenez à la normale.

Dorénavant il faudra systématiquement récupérer `my` en utilisant `get_mydata`.

III. Conteneurs

Nous allons changer de conteneur pour les boutons. Le conteneur `gtk_button_box` que nous avons utilisé jusqu'à présent est trop rigide, et n'adapte pas la taille des boutons à la place disponible.

1) Lisez la description des paramètres de `gtk_box_new` sur `Devhelp` (si installé sur votre ordinateur) ou sur le site https://developer.gnome.org/gtk3/stable/ en tapant le nom de la fonction en haut dans la barre de recherche. Dans la suite on dira "voir l'aide" pour renvoyer à la description d'une fonction.

2) Remplacer : `button_box = gtk_button_box_new (GTK_ORIENTATION_HORIZONTAL);`
par `button_box = gtk_box_new (GTK_ORIENTATION_HORIZONTAL, 4);`
Pour chaque bouton, remplacer `gtk_container_add (GTK_CONTAINER (button_box), le_bouton);`
par `gtk_box_pack_start (GTK_BOX (button_box), le_bouton, TRUE, TRUE, 0);`
Compilez, puis déformez la fenêtre à la souris. Vous constaterez que les boutons prennent toute la place en hauteur et en largeur, en laissant 4 pixels entre les deux boutons.

3) Les combinaisons possibles pour `gtk_box_pack_start` (voir l'aide) sont :
`FALSE, FALSE` ou `TRUE, FALSE` ou encore `TRUE,TRUE` .
Essayez les trois combinaisons pour le `button1` et déformez la fenêtre.

IV. Conteneurs dans des conteneurs

Nous allons créer davantage de boutons et les organiser dans la fenêtre. Certains boutons seront ensuite remplacés par d'autres widgets, lorsque le placement sera au point.

1) On va remanier complètement `on_app_activate`, dans le but d'obtenir la disposition à droite. Supprimez le `button_box` ; remplacez-le par un widget `vbox1`, créé à l'aide de `gtk_box_new` dans le sens vertical ; attachez-le à la fenêtre à l'aide de `gtk_container_add`.

2) Créez un widget `hbox1`, créé à l'aide de `gtk_box_new` dans le sens horizontal ; attachez-le à `vbox1` à l'aide de `gtk_box_pack_start` de façon à minimiser la place occupée.

3) Attachez `button1` et `b_quit` à `hbox1` à l'aide de `gtk_box_pack_start`, de façon à ce que `button1` maximise et `b_quit` minimise la place.

4) Créez deux boutons supplémentaires `b_center` et `b_bottom` (inutile de leur connecter une call-back). Attachez-les à `vbox1` à l'aide de `gtk_box_pack_start`, de façon à ce que `b_center` maximise et `b_bottom` minimise la place.

5) Testez en déformant la fenêtre.

V. Affichage d'une image

Nous allons voir comment afficher une image.

1) Allez dans l'application `DevHelp` et dans le champs de recherche tapez `gallery`, ou encore allez sur site https://developer.gnome.org/gtk3/stable/ et descendez dans la fenêtre jusqu'à "Widget Gallery".

Cliquez ensuite sur le widget "Image", puis sur la fonction `gtk_image_new_from_file` : en lisant la description vous allez voir que cette fonction lit une image (au format `png`, `jpg`, `gif`, etc) et créé un widget `image` capable de l'afficher.

2) Recopiez dans votre répertoire local le fichier `tux2.gif` que vous récupérerez soit dans `/usr/local/EZ-Draw-1.2/`, soit sur la page http://j.mp/ez-draw dans le `.tar.gz`, répertoire `images/`.

3) Dans `on_app_activate`, créez un widget `image1` à l'aide de la fonction `gtk_image_new_from_file` avec le fichier `tux2.gif`. Attachez-le à la place de `b_center`, puis supprimez `b_center`. Vous devriez obtenir la fenêtre à droite (dimensions mises à part).

4) Changez le label de `button1` en `"Load image"`. Déplacez la déclaration de `image1` dans `my`, puis remplacez `image1` par `my->image1` dans `on_app_activate`. Testez.

5) Dans `on_app_activate`, remplacez `gtk_image_new_from_file` par `gtk_image_new` (voir la doc). Dans la callback de `button1`, appelez `gtk_image_set_from_file` de façon à afficher le fichier `tux2.gif` dans `image1`. Vérifiez qu'au lancement il n'y a pas d'image, mais qu'elle apparaît bien lorsque le bouton `"Load image"` est cliqué.

VI. Sélection de fichier

Nous allons doter notre programme d'un sélecteur de fichier. Ce sélecteur apparaîtra lorsque l'on cliquera sur le bouton `"Load Image"`.

1) Dans la "Widget Gallery", cliquez sur le widget `GtkFileChooserDialog` et lisez la section Description ; la partie qui nous intéresse spécialement est le premier cas de la sous-section "Typical usage" (action `OPEN`).

2) Tout va maintenant se passer dans la callback du `button1`. Commentez l'appel actuel à la fonction `gtk_image_set_from_file` et recopiez en dessous le code du "Typical usage" pour l'action `OPEN`, en y faisant les adaptations suivantes :

- ▷ changez le titre de la boîte de dialogue, et donnez comme parent le `window` stocké dans `my` ;
- ▷ remplacez `_("_Cancel")` par `"Cancel"` et de même pour `"Open"` : la fonction `_()` est une opération pour l'internationalisation de l'interface, dont nous allons nous passer ;
- ▷ remplacez `open_file(filename)` par l'appel à `gtk_image_set_from_file`, en lui donnant `filename` en paramètre.

La boîte de dialogue devrait maintenant être fonctionnelle. Testez sur plusieurs images (par exemple les monstres dans le répertoire `images-doodle/` de EZ-Draw). Si le fichier n'est pas une image, une petite icone apparaitra pour signaler l'erreur.

3) Problème : à chaque appel du sélecteur de fichier, on revient dans le répertoire local et on perd l'emplacement où l'on était allé. Pour corriger ceci, déclarez le champ `char *current_folder` dans le struct `Mydata` et initialisez-le à `NULL` dans `init_mydata`. Dans la callback du `button1` : avant l'appel de `gtk_dialog_run`, si `my->current_folder` est non `NULL`, transmettez-le au `dialog` à l'aide de `gtk_file_chooser_set_current_folder` ; dans le bloc `GTK_RESPONSE_ACCEPT`, libérez `my->current_folder` puis mémorisez dedans le résultat de `gtk_file_chooser_get_current_folder`. Le problème devrait maintenant être corrigé.

VII. Barre d'état

Nous allons rajouter une barre d'état en dessous de la zone image, pour afficher des messages tels que "Loading failed : not an image" ou encore "Loading success : image 200x300". L'emplacement est déjà prévu, et actuellement occupé par le bouton `b_bottom`.

1) Dans la "Widget Gallery", cliquez sur le widget `GtkStatusbar` et lisez la section Description. Nous nous simplifierons les choses en prenant toujours pour `context_id` le numéro 0.

2) Déclarez un widget `status` dans le struct `Mydata` ; dans `on_app_activate`, créez `my->status` avec la fonction `gtk_statusbar_new` puis attachez-le à la place de `b_bottom`, que vous pouvez supprimer.

3) Écrivez la fonction `void set_status (Mydata *my, const char *msg)`, dans laquelle vous appelerez `gtk_statusbar_pop` pour le `context_id` 0, puis `gtk_statusbar_push` pour le `context_id` 0 et le texte `msg`. Dans `on_app_activate` appelez `set_status` *après* la création de `my->status`, avec le message `"Wellcome in TP1!"`.

4) Le but est d'arriver à savoir si une image a bien été chargée ou s'il y a une erreur. Avant l'appel de `gtk_image_set_from_file`, appelez `set_status` pour afficher le status "Loading image...". Remplacez ensuite l'appel de `gtk_image_set_from_file` par le bloc suivant :

- ▷ Créez `GdkPixbuf *pixbuf` avec `gdk_pixbuf_new_from_file` avec les paramètres `filename` et `NULL`.
- ▷ Ensuite, si `pixbuf` est `NULL` cela signifie que l'image n'a pas pu être chargée : affichez le status "Loading failed : not an image", enfin donnez une image à afficher pour `my->image1` en appelant `gtk_image_set_from_icon_name` avec dans les paramètres le nom `"image-missing"` et la taille `GTK_ICON_SIZE_DIALOG`.
- ▷ Sinon, affichez le status "Loading success", donnez l'image à afficher pour `my->image1` avec `gtk_image_set_from_pixbuf` en lui passant le `pixbuf`, enfin déréférencez le `pixbuf` avec `g_object_unref` (de la sorte, seul `my->image1` le référencera, ce qui provoquera sa destruction automatique au prochain chargement d'image).

Refaites des essais sur des images et d'autres fichiers.

5) Juste avant d'afficher le status "Loading success", récupérez la taille de l'image avec les fonctions `gdk_pixbuf_get_width` et `gdk_pixbuf_get_height`, puis fabriquez avec `sprintf` un message `"Loading success: image %dx%d"` avec les dimensions trouvées ; enfin affichez ce message comme status.

VIII. Rotation

Pour terminer, nous allons faire une rotation de l'image.

1) Rajoutez un bouton `b_rotate` avec pour titre "Rotate" et attachez-le entre les deux boutons actuels. Connectez une callback pour le signal `clicked`.

2) Déclarer `GdkPixbuf *pixbuf1` dans `Mydata` et initialisez-le à `NULL`. Dans la callback du bouton de chargement de l'image, remplacez les occurences de `pixbuf` par `my->pixbuf1`. Testez qu'il n'a pas de régression (c'est-à-dire que votre programme fonctionne encore).

3) Remplacez `g_object_unref (pixbuf);` par `g_clear_object (&my->pixbuf1);` en déplaçant l'appel juste avant `gdk_pixbuf_new_from_file`. De la sorte, le `pixbuf1` sera conservé en mémoire jusqu'au prochain chargement, avant lequel il sera libéré.

4) Dans la callback de `b_rotate`, testez que `my->pixbuf1` est non `NULL` sinon quittez la fonction. Créez un pixbuf temporaire `tmp` avec `gdk_pixbuf_rotate_simple` à partir de `my->pixbuf1` pour l'angle 90. Déréférencez l'actuel `my->pixbuf1` avec `g_object_unref`, puis mémorisez `tmp` dans `my->pixbuf1`. Enfin affichez le nouveau pixbuf avec `gtk_image_set_from_pixbuf` et affichez le status "Image rotated".

TP2 : plus de widgets en GTK+

I. Préparation

Cette planche est la suite du TP1 ; il est indispensable de l'avoir terminé avant de commencer celle-ci.

Récupérez ensuite les trois scripts vus en cours (`compgtk3.sh`, `allcomp.sh`, `allclean.sh`) à partir des slides sur http://j.mp/optigra et testez-les sur vos programmes.

II. Ascenseurs automatiques

Lorsque l'on charge une image trop grande ce n'est pas très pratique, car la fenêtre est automatiquement agrandie pour la contenir. Nous allons rajouter des ascenseurs autour de l'image, qui apparaîtront automatiquement si l'image est trop grande pour la fenêtre. Pour cela nous utilisons un `GtkScrolledWindow` qui est un conteneur dérivé de `GtkBin`.

Créez un widget `scroll` avec `gtk_scrolled_window_new`, en lui passant `NULL` pour que GTK crée tout automatiquement.

Initialement, `my->image1` était attaché à `vbox1`. À la place de cela, attachez `scroll` à `vbox1` de manière à ce qu'il prennent toute la place, et attachez `my->image1` à `scroll`.

Testez avec une grande image, par exemple une capture d'écran.

III. Barre de menus

Nous allons remplacer les boutons qui ornent le haut de la fenêtre par une barre de menus.

1) Mettez en commentaire la création de `hbox1` et des boutons qui lui sont attachés. Créez un `GtkMenuBar` appelé `menu_bar` et attachez-le à `vbox1` à la place de `hbox1`. Créez des `GtkMenuItem`, appelés `item_file`, `item_tools`, `item_help` et intitulés `"File"`, `"Tools"`, `"Help"` ; attachez-les dans `menu_bar`. Enfin placez la propriété `"gtk-shell-shows-menubar"` (voir fin du cours) pour que la barre de menu apparaisse bien dans la fenêtre. Testez avec et sans la propriété.

2) Créez un `GtkMenu` appelé `sub_file`, que vous attacherez comme sous-menu de `item_file` ; créez des `GtkMenuItem` appelés `item_load`, `item_quit`, et intitulés `"Load"` et `"Quit"` ; attachez-les dans le sous-menu `sub_file`. Avec le même système de notations, créez un sous-menu pour `"Tools"` avec `"Rotate"` et `"Bg color"`, et un sous-menu pour `"Help"` avec `"About"`.

3) Attachez des callbacks pour le signal `activate` à chacun des menu-item terminaux, en les appelant `on_`*nom-du-widget*`_activate`. Déplacez le code initialement déclenché pour le bouton `"Load"` dans la callback `on_item_load_activate` ; faites de même pour `Rotate` et pour `Quit`. Arrivé à ce stade, vous pouvez supprimer les parties mises en commentaire dans la sous-section 1).

4) Mémorisez `vbox1` dans `Mydata`. Découpez `on_app_activate` en fonctions, que l'on nommera dans l'ordre : `window_init`, `layout_init` (pour `vbox1`), `menus_init`, `image_init`, `status_init`. Passez-leur chaque fois `my` (ainsi que `app` pour la première).

IV. Boîte About

Nous allons rajouter une boîte de dialogue "About" à l'aide du widget `GtkAboutDialog`.

1) Connectez une callback `on_item_about_activate` au menu-item `item_about`. Dans la callback, appelez la fonction `gtk_show_about_dialog` comme dans l'exemple au début de la documentation du widget, en lui passant en propriétés :

- ▷ `"program-name"` avec la valeur `my->title`,
- ▷ `"version"` avec la valeur `"2.4"`,
- ▷ `"website"` avec la valeur `"http://j.mp/optigra"`.

N'oubliez pas le `NULL` au début et à la fin. Vous devriez obtenir une boîte de dialogue avec un lien cliquable qui renvoie sur le site donné en propriété.

2) Dans la même callback, déclarez un `char *auteurs[] = {...., NULL};` comprenant la liste des auteurs sous la forme `"Prénon Nom <adresse@email>"` ; rajoutez à `gtk_show_about_dialog` la propriété `"authors"` avec la valeur `auteurs`. Vous devriez maintenant voir un bouton à bascule `Credits` qui fait apparaître ou disparaître une liste de noms cliquables.

3) Occupons-nous enfin du logo : rajoutez une propriété `"logo-icon-name"` avec comme valeur par exemple `"face-laugh"` ou `"face-cool"`. Vous trouverez d'autres noms d'émoticones sur la page web https://developer.gnome.org/icon-naming-spec/#emotes ; vous pouvez également les prévisualiser avec le gestionnaire de fichier dans `/usr/share/icons/gnome/48x48/emotes/`.

V. Couleur du fond

La couleur du fond de la zone image provient actuellement du thème de votre bureau. Nous allons la changer en utilisant un `GtkColorChooserDialog`, qui est un widget qui dérive de `GtkDialog`.

1) Connectez une callback `on_item_bg_color_activate` au menu-item `item_bg_color`. Dans cette callback, créez un `GtkColorChooserDialog` au moyen de `gtk_color_chooser_dialog_new` avec comme titre `"Background color"` et pour parent `NULL`. Animez la boîte avec `gtk_dialog_run` puis détruisez le widget avec `gtk_widget_destroy`.

2) Si l'utilisateur sélectionne une couleur dans la boîte de dialogue et clique sur "Sélectionner", `gtk_dialog_run` renvoie `GTK_RESPONSE_OK`. Détectez ce cas et affichez les valeurs R,G,B de la couleur sélectionnée dans la barre de status en bas de la fenêtre. Voici comment faire :

Récupérez la couleur sélectionnée avec la fonction `gtk_color_chooser_get_rgba` et passez-lui par référence un `GdkRGBA bg_color`. Convertissez ensuite cette couleur en chaîne de caractère avec `gdk_rgba_to_string` (il faudra la libérer à la fin avec `g_free`). À l'aide de `sprintf` fabriquez un message du genre `"Selected bg color: ..."` puis affichez-le avec `set_status`.

3) Au moyen de `gtk_widget_override_background_color`, changez la couleur du fond du widget `my->image1` pour l'état `GTK_STATE_FLAG_NORMAL` en lui passant la couleur venant d'être sélectionnée `bg_color`.

VI. Zoom de l'image

Nous allons fabriquer une fenêtre dans laquelle nous placerons un slider pour zoomer sur l'image.

1) Créez la fonction `void win_scale_init (Mydata *my)`, que vous appellerez dans la callback `on_app_activate`. Déclarez un widget `win_scale` dans `Mydata`, et créez-le dans `win_scale_init` au moyen de `gtk_window_new`, de type `GTK_WINDOW_TOPLEVEL`. Donnez-lui pour titre `"Image scale"` ; enfin appelez `gtk_widget_show_all`. Vérifiez que la fenêtre s'affiche au lancement de l'application.

2) À la fin de `win_scale_init` cachez la fenêtre avec `gtk_widget_hide`. Rajoutez un menu-item `item_scale` intitulé `"Scale"` dans le sous-menuw `sub_tools`, et connectez-lui une callback dans laquelle vous appellerez `gtk_window_present` pour afficher la fenêtre `win_scale`. Vérifiez qu'elle s'affiche lorsque le menu est cliqué.

3) Si vous fermez cette fenêtre à l'aide de l'icone de la barre de titre, vous constaterez qu'elle ne peut plus être ré-affichée via le menu, tout simplement parce que le comportement par défaut associé à ce bouton est de détruire la fenêtre. Voila qui ne fait pas nos affaires, nous voudrions que la fenêtre soit simplement cachée. Pour ce faire il suffit de capter le signal associé au bouton fermer de la fenêtre : connectez au widget `win_scale` pour le signal `delete-event` la callback `gtk_widget_hide_on_delete`, qui est spécialement prévue pour cet usage.

4) Donnez à la fenêtre `win_scale` les dimensions 300×100 avec `gtk_window_set_default_size`. Attachez dans cette fenêtre un `GtkBox` horizontal, dans lequel vous placerez un `GtkLabel` intitulé `"Scale:"` puis un `GtkScale` horizontal désigné par `scale1` avec l'intervalle $0,02 \ldots 20,0$ et un pas de $0,02$. Attachez ces widgets au `GtkBox` avec un padding de 10 pixels.

5) Le widget `GtkScale` dérive de `GtkRange`. Utilisez `gtk_range_set_value` pour fixer la valeur par défaut `1,0`. Lorsque l'utilisateur déplace le curseur, le signal `value-changed` est émis. Connectez le widget `scale1` à ce signal avec la callback `on_scale1_value_changed`. Cherchez dans la documentation la signature de cette callback, puis récupérez la valeur actuelle *value* avec `gtk_range_get_value` et stockez-la dans un nouveau champ `scale1_value` de `my` initialisé à `1.0`. Ensuite fabriquez avec `sprintf` un message du genre `"Scale` *value*`"` avec deux chiffres après la virgule et affichez-le avec `set_status`. Vérifiez que lorsque le curseur est déplacé, les valeurs apparaissent en même temps dans la barre de status, comme dans la figure ci-dessous.

6) Nous allons zoomer sur l'image dans la callback en appliquant le facteur d'échelle stocké dans `my->scale1_value`. Si `my->pixbuf1` est `NULL` on sort immédiatement de la fonction. Récupérez les dimensions actuelles de `my->pixbuf1` à l'aide de `gdk_pixbuf_get_width` et `gdk_pixbuf_get_height` ; créez un `GdkPixbuf` temporaire `tmp_pixbuf` avec la fonction `gdk_pixbuf_scale_simple` en appliquant `my->scale1_value` sur les dimensions, avec une interpolation `GDK_INTERP_BILINEAR` ; attachez `tmp_pixbuf` au widget image puis déréférencez l'objet `tmp_pixbuf` (on peut s'inspirer du code écrit dans `on_item_load_activate`). Surtout ne modifiez pas `my->pixbuf1`.

7) Si l'on charge une nouvelle image, il faut remettre le zoom à 1. Mémorisez le widget `scale1` dans `Mydata` ; modifiez la callback `on_item_load_activate` de façon à réinitialiser le widget `my->scale1` ainsi que `my->scale1_value` à $1,0$.

Problème : La modification du widget entraîne l'émission du signal `value-changed` (si le zoom était différent de 1), qui fait recalculer l'image et afficher le status `"Scale 1.0"`, si bien que l'on ne voit pas le status de chargement de l'image avec sa taille.

Solution : réinitialisez `my->scale1_value` *avant* le widget dans `on_item_load_activate`, et d'autre part au début de `on_scale1_value_changed`, comparez `my->scale1_value` à la nouvelle valeur : si la valeur est la même on sort immédiatement de la fonction.

8) Modifiez `on_item_rotate_activate` afin que l'image résultante soit également zoomée avec le facteur `my->scale1_value` courant. Attention, `my->pixbuf1` ne doit pas être zoomé, c'est le widget `my->image1` qui doit contenir un pixbuf temporaire zoomé. Attention également aux déréférencements pour éviter les fuites de mémoire.

TP3 : zone de dessin

I. Préparation

Cette planche est la suite du TP2 ; il est indispensable de l'avoir terminé avant de commencer celle-ci (mise à part la dernière question subsidiaire).

Recopiez votre programme du TP2 sous un nouveau nom, par exemple `tp3.c`.

1) Nous allons remplacer le widget `GtkImage` par un widget `GtkDrawingArea`. Dans votre programme, mettez en commentaire toutes les lignes où apparaît `my->image1`, y compris la déclaration dans le struct `Mydata`. Vérifiez que le programme compile et qu'il est capable de charger une image (sans pouvoir l'afficher, évidemment).

2) Déclarez dans le struct `Mydata` un champ `GtkWidget *area1`. Renommez `image_init` par `area1_init`. Dans `area1_init`, remplacez la création de `my->image1` par celle du widget `my->area1` du genre `GtkDrawingArea` ; de même pour son attachement au conteneur `my->scroll`. Enfin donnez une taille minimale de 600×400 à `my->area` (voir le cours), puis vérifiez à l'exécution que les ascenseurs apparaissent ou disparaissent selon la taille que vous donnez à la fenêtre.

3) Dans `on_item_bgcolor_activate`, dé-commentez `gtk_widget_override_background_color` et appliquez-le à `my->area1`. Vérifiez que vous pouvez changer la couleur du fond avec la boîte de dialogue.

II. Affichage de l'image

1) Dans `area1_init`, connectez le signal `draw` à la callback `on_area1_draw`. Recherchez la signature, c'est à dire les paramètres de cette fonction, dans http://j.mp/optigra ▷ La hiérarchie des Objets GTK ▷ `GtkWidget` ▷ "Signals" en haut dans la barre blanche à droite ▷ `draw` (il y a une petite erreur dans la documentation, en fait le bon type pour le second paramètre est `cairo_t *cr`).

Dans la callback, récupérez la taille du widget, puis affichez `"on_area1_draw:` *largeur hauteur*`"`, enfin n'oubliez pas de renvoyer `TRUE`. Vérifiez que cette fonction est appelée au démarrage du programme et lorsque la taille de la fenêtre change fortement.

2) Dans `on_area1_draw`, testez si `my->pixbuf1` est non `NULL`, et dans ce cas effectuez les opérations suivantes : récupérez la taille du pixbuf avec `gdk_pixbuf_get_width` et `gdk_pixbuf_get_height` ; donnez le pixbuf comme source au contexte Cairo `cr` avec `gdk_cairo_set_source_pixbuf` depuis l'origine ; donnez comme chemin un rectangle de la taille du pixbuf avec `cairo_rectangle` ; enfin appliquez le chemin avec `cairo_fill`.

Chargez une image ; elle devrait s'afficher (par bonheur, GTK envoie un signal `draw` à la fenêtre principale lorsque la boîte de dialogue est fermée).

3) À ce stade, les ascenseurs du conteneur `my->scroll` sont gérés par rapport à la taille minimale que nous avons donnée dans `area1_init` et non par rapport à la taille de l'image, ce qui est assez déroutant. Supprimez l'appel de `gtk_widget_set_size_request` dans `area_init1` et placez-le dans `on_item_load_activate` avec la taille du pixbuf venant d'être chargé. Vérifiez que les ascenseurs ont désormais le bon comportement.

4) Testons maintenant la rotation de l'image. Actuellement, la rotation ne semble pas fonctionner, et pourtant si, mais il faut agrandir la fenêtre pour que l'image apparaisse tournée. La cause en est qu'après la rotation du pixbuf il n'y a pas de signal `draw` généré.

Copiez-collez la fonction `void refresh_area (GtkWidget *area)` vue en cours. Appelez cette fonction après la rotation du pixbuf. Vérifiez que désormais le menu rotation provoque le ré-affichage immédiat de l'image tournée.

5) Il y a encore un problème d'ascenseur ! En effet lorsque l'on tourne le pixbuf, il faut aussi changer la taille minimale de `my->area1` en fonction de la nouvelle taille du pixbuf. Testez sur une image rectangulaire.

III. Le problème du zoom

Il reste le problème du zoom à régler. L'approche que nous avions choisie au TP2 ne peut être transposée ici parce que nous ne pouvons pas attacher le pixbuf temporaire zoomé dans le `GtkDrawingArea`. Nous allons donc changer de stratégie. Vérifiez à chaque étape que le programme compile sans erreur.

1) Dans `Mydata`, déclarez `GdkPixbuf *pixbuf2` et `double rotate_angle` et initialisez-les. Le but va être que `pixbuf1` contiendra l'image chargée non transformée, tandis que `pixbuf2` contiendra l'image tournée et zoomée à afficher.

2) Créez la fonction `void apply_image_transforms (Mydata *my)`. Au début de la fonction, appelez `g_clear_object` sur `&my->pixbuf2`. Si `my->pixbuf1` est `NULL` quittez la fonction. En vous inspirant du code figurant actuellement dans `on_item_rotate_activate`, créez un pixbuf temporaire `tmp1` qui sera le résultat de la rotation de `my->pixbuf1` avec `my->rotate_angle` en degrés, puis effectuez le zoom de facteur `my->scale1_value` et stockez le résultat dans `my->pixbuf2`. Déréférencez enfin `tmp1`.

3) Créez la fonction `void update_area1_with_transforms (Mydata *my)`. Au début de la fonction appelez `apply_image_transforms`. Si `my->pixbuf2` est non `NULL`, changez la taille minimale de `my->area1` aux dimensions de `my->pixbuf2`. Enfin appelez `refresh_area`.

4) Branchement du nouveau code. Dans `on_area1_draw`, remplacez `my->pixbuf1` par `my->pixbuf2`. Dans `on_item_load_activate`, à la fin du bloc dans lequel on a chargé avec succès : supprimez la mise à jour de la taille minimale de `my->area1`, réinitialisez ici le zoom à 1, affectez l'angle à 0 puis appelez `update_area1_with_transforms`. Vérifiez que l'image s'affiche après chargement, et qui si le zoom était différent de 1 il est revenu à 1 dans la boîte de dialogue (à ce stade il n'a aucun effet sur l'image).

5) C'est ici que nous allons voir le bénéfice de ces changements : dans `on_item_rotate_activate`, levez tout le code sauf l'appel à `set_status`, qui va se retrouver à la fin de la fonction. Incrémentez `my->rotate_angle` de 90, et s'il est supérieur ou égal à 360, retranchez-lui 360. Appelez enfin la fonction `update_area1_with_transforms`. Vérifiez que la rotation refonctionne ainsi que la taille relative des ascenseurs.

6) Occupons-nous finalement du zoom. Dans `on_scale1_value_changed`, levez tout le code de transformation, et remplacez-le par l'appel à `update_area1_with_transforms`. Vérifiez que le zoom refonctionne, ainsi que la taille relative des ascenseurs, et que la rotation fonctionne encore également avec le zoom.
Félicitations pour être arrivé jusqu'ici, vous avez gagné une pause sucrée !

IV. Événements utilisateur

Nous avons finalement payé assez cher le passage du `GtkImage` au `GtkDrawingArea`. C'est maintenant que nous allons voir l'intérêt en terme de fonctionnalités : en effet nous allons pouvoir dessiner sur l'image, puis découper l'image en morceaux.

1) Dans `area1_init`, connectez des callbacks au widget `my->area1` pour les signaux suivants : `key-press-event`, `key-release-event`, `button-press-event`, `button-release-event`, `motion-notify-event`, `enter-notify-event`, `leave-notify-event`. Les callbacks auront pour nom `on_area1_`*nom_du_signal* sans la partie `-event` ; par exemple `on_area1_key_press`.

2) Autorisez tous les événements nécessaires avec `gtk_widget_add_events` et le masque vu en cours. Autorisez le focus avec `gtk_widget_set_can_focus`.

3) Par chance toutes ces callbacks ont la même signature. Dans chacune, récupérez l'événement à partir du `GdkEvent` comme vu en cours, puis affichez le nom de la fonction et des informations propres à l'événement, par exemple les coordonnées de la souris, le numéro du bouton, le symbole de la touche. N'oubliez pas de renvoyer `TRUE` à la fin de chacune. Exécutez et vérifiez que tous les événements sont ainsi affichés dans le terminal.

4) Dans `on_area1_enter_notify`, prenez le focus clavier avec `gtk_widget_grab_focus`. De la sorte, si `my->area1` avait perdu le focus, il le retrouvera en passant la souris dessus. Dans `on_area1_key_press`, si la touche `q` est pressée, détruisez la fenêtre principale.

V. Clic souris et découpage d'image

1) Déclarez dans `Mydata` les champs réels `click_x` et `click_y` (coordonnées de la souris), ainsi que l'entier `click_n` (numéro du bouton enfoncé de la souris, 0 pour aucun) ; initialisez-les à 0.

2) Dans `on_area1_button_press` mettez à jour les trois champs `click_` de `my` à partir des valeurs présentes dans le `GdkEventButton`. Dans `on_area_button_release` mettez `click_n` à 0. Dans `on_area1_motion_notify`, si `click_n` vaut 1 (bouton gauche de la souris enfoncé), mettez à jour `click_x` et `click_y` à partir des valeurs stockées dans le `GdkEventMotion`. Enfin, appelez `refresh_area` dans chacune des trois callbacks sus-mentionnées (sauf pour un mouvement sans bouton enfoncé).

3) Dans `on_area1_draw`, si `my->click_n` vaut 1, dessinez par dessus l'image (autrement dit, *après* l'affichage de l'image) un cercle de centre (`my->click_x`, `my->click_y`) de rayon 100 et de couleur bleue. Vérifiez que le cercle bleu apparaît, avec ou sans image, lorsque l'on clique ou l'on tire la souris, et que ce cercle disparaît sitôt le bouton relâché.

4) Dans `menus_init` rajoutez un `GtkCheckMenuItem` avec le label "Clip" au menu `Tools`. Déclarez dans `Mydata` un entier `clip_image` initialisé à `FALSE`. Connectez la callback `on_item_clip_activate` au signal `activate`. Dans cette callback, mettez à jour `my->clip_image` en fonction de l'état du widget, récupéré à l'aide de `gtk_check_menu_item_get_active` ; affichez dans la barre de status "Clipping is on" ou "Clipping is off" ; enfin appelez `refresh_area`.

5) Dans `on_area1_draw`, mettez en commentaire l'affichage de `my->pixbuf2` ; comme vu à la fin du cours, affichez à la fin de la fonction la portion d'image de `my->pixbuf2` située dans le cercle (avec pour origine (0,0) dans le pixbuf). Tirez la souris pour tester.

6) Enfin pour tenir compte du menu-item "Clip", procédez comme suit dans `on_area1_draw` : n'affichez la totalité de `my->pixbuf2` que s'il est non `NULL`, et si on n'a pas à la fois `my->clip_image` vrai et `my->click_n` égal à 1 ; n'affichez la portion d'image située dans le cercle que si `my->pixbuf2` est non `NULL` et `my->clip_image` est vrai et `my->click_n` est égal à 1.

Finalement vérifiez que le clipping est fait uniquement lorsque "Clip" est activé et que l'on clique sur l'image, et de plus qu'il est compatible avec le zoom et la rotation.

TP4 : points de contrôle

I. Préparation

Cette planche est la suite du TP3 ; il est indispensable de l'avoir terminé avant de commencer celle-ci.

Recopiez votre programme du TP3 sous un nouveau nom, par exemple `tp4.c`.

1) Modifiez la taille initiale de la fenêtre dans `init_mydata`, par exemple 500 × 400.

2) Réorganisons un peu notre code : dans `on_app_activate`, déplacez l'appel de `layout_init` *après* celui de `window_init`, `menus_init`, `area1_init` et `status_init`. Ensuite déplacez les appels à `gtk_box_pack_start` et `gtk_container_add` de ces fonctions vers `layout_init` (il vous faudra peut-être déclarer certains widgets dans `Mydata`) ainsi que la création du Scrolled Window qui contient le Drawing Area (on ne touche pas à `win_scale_init`). De cette façon la fonction `layout_init` sera responsable de l'organisation des widgets, qui sera plus facile à maintenir.

II. Boutons radio

Nous allons rajouter une zone de boutons radios à gauche du Drawing Area.

1) Rajoutez une fonction `void editing_init (Mydata *my)` et appelez-là *avant* `layout_init`. Dans `editing_init` créez un `GtkFrame` nommé "Editing" et mémorisé dans un nouveau champ `frame1` de `Mydata`. Dans `layout_init` créez un `GtkBox` horizontal mémorisé dans un nouveau champ `hbox1` de `Mydata`. Cette Box viendra s'attacher dans `my->vbox1` à la place du Scrolled Window et prendra toute la place disponible ; elle contiendra le nouveau Frame (avec un bord de 2 pixels) ainsi que le Scrolled Window.

2) Dans `editing_init`, créez des boutons radios intitulés "Add curve", "Move curve", "Remove curve", "Add control", "Move control", "Remove control" ; créez un `GtkBox` vertical, attachez-le au conteneur `my->frame1` puis attachez les boutons radios dans la Box. Vérifiez que les boutons radios fonctionnent correctement (un seul est actif à la fois).

3) Dans `menus_init` rajoutez un `GtkCheckMenuItem` avec le label "Editing" au menu `Tools`. Déclarez dans `Mydata` un entier `show_edit` initialisé à `FALSE`. Connectez au signal `activate` la callback `on_item_editing_activate`. Dans cette callback, mettez à jour `my->show_edit` en fonction de l'état du widget, récupéré à l'aide de `gtk_check_menu_item_get_active` ; affichez dans la barre de status "Editing is on" ou "Editing is off".

4) Tout à la fin de `on_app_activate`, cachez `my->frame1` avec `gtk_widget_hide`. Affichez ou cachez `my->frame1` dans `on_item_editing_activate` selon l'état de `my->show_edit`. Vérifiez que la barre latérale de boutons radio apparaît ou disparaît avec le menu-item "Editing".

5) Déclarez un `enum` avec les constantes : `EDIT_NONE`, `EDIT_ADD_CURVE`, `EDIT_MOVE_CURVE`, `EDIT_REMOVE_CURVE`, `EDIT_ADD_CONTROL`, `EDIT_MOVE_CONTROL`, `EDIT_REMOVE_CONTROL`, `EDIT_LAST`. Ces constantes représenteront le mode d'édition. Rajoutez dans `Mydata` un champ entier `edit_mode` initialisé au mode `EDIT_ADD_CURVE` ; rajoutez également le tableau `GtkWidget *edit_radios[EDIT_LAST],` puis mémorisez dans ce tableau les boutons radio lors de leur création.

6) Dans `editing_init`, pour chaque bouton radio, mémorisez le mode correspondant au bouton à l'aide de `g_object_set_data` (voir le cours), associé à la clé `"mode"`. Connectez chaque bouton à `on_edit_radio_toggled` pour le signal `toggled`. Dans `on_edit_radio_toggled`, récupérez le mode

courant à l'aide de `g_object_get_data`, stockez-le dans `my->edit_mode` puis affichez-le dans le terminal.

7) Écrivez la fonction `void set_edit_mode (Mydata *my, int mode)`, qui sort immédiatement si `mode` est hors de l'intervalle ouvert `EDIT_NONE .. EDIT_LAST` ; la fonction rend active le bouton radio correspondant à `mode`. Dans `on_area1_key_press`, associez à des touches du clavier (par exemple les premières lettres de l'alphabet) les différents modes d'édition avec `set_edit_mode`.

III. Points de contrôle

Nous allons mettre en place la gestion de points de contrôle afin de dessiner par la suite des courbes de Bézier.

1) Déclarez la structure de données ci-contre. Rajoutez dans `Mydata` un champ `Curve_infos curve_infos`. Écrivez la fonction `void init_curve_infos (Curve_infos *ci)` qui initialise `curve_count` à 0, `current_curve` et `current_control` à -1. Dans `init_mydata` appelez `init_curve_infos (&my->curve_infos)`.

```
#define CONTROL_MAX 100
#define CURVE_MAX  200

typedef struct {
  double x, y;
} Control;

typedef struct {
  int control_count;
  Control controls[CONTROL_MAX];
} Curve;

typedef struct {
  int curve_count;
  Curve curves[CURVE_MAX];
} Curve_list;

typedef struct {
  Curve_list curve_list;
  int current_curve,
      current_control;
} Curve_infos;
```

2) Écrivez la fonction `int add_curve (Curve_infos *ci)`. Si `curve_list` est plein, la fonction met `current_curve` à -1 puis renvoie -1. La fonction prend pour nouveau numéro de courbe `n` la valeur actuelle de `curve_count`, puis elle incrémente ce dernier. Elle initialise à 0 le compteur `control_count` de la courbe `n`, met `current_curve` à `n`, `current_control` à -1 puis renvoie `n`.

3) Écrivez la fonction `int add_control (Curve_infos *ci, double x, double y)`, qui renvoie -1 si `n = ci->current_curve` est hors de l'intervalle fermé `0 .. curve_count-1`. Si le tableau de points de contrôle de la courbe `n` est plein, la fonction met `current_control` à -1 puis renvoie -1. La fonction prend ensuite pour nouveau numéro de point de contrôle `k` la valeur actuelle de `control_count` pour la courbe numéro `n`. La fonction incrémente pour la courbe `n` son compteur `control_count` puis mémorise les coordonnées du nouveau point `x`,`y` en position `k`, enfin elle met `current_control` à `k` puis renvoie `k`.

Astuce : pour simplifier l'écriture, on peut passer par des variables intermédiaires telles que `Curve *curve = &ci->curve_list.curves[n]`.

4) Réorganisez `on_area1_button_press` comme suit : on mémorise les coordonnées et le bouton souris, puis si le bouton 1 est pressé et `my->show_edit` est vrai, on fait un branchement (`switch`) selon `my->edit_mode` sur l'un des 6 cas prévus, puis on rafraîchit le Drawing Area.

Dans le branchement, pour le cas du mode `EDIT_ADD_CURVE` : on appelle `add_curve` ; si le résultat est négatif on sort du branchement ; on bascule ensuite en mode `EDIT_ADD_CONTROL` à l'aide de `set_edit_mode`, puis on rajoute un point de contrôle avec `add_control`.

5) Mettez en commentaire la version actuelle de `on_area1_draw` et faites une nouvelle version de cette fonction, dans laquelle vous ne conserverez que l'affichage du pixbuf s'il est non `NULL`. Après l'affichage éventuel du pixbuf appelez une nouvelle fonction `void draw_curves (cairo_t *cr, Curve_infos *ci)`, dans laquelle pour chaque courbe, vous afficherez pour chaque point de contrôle un carré bleu de côté 6 et centré sur le point de contrôle (c'est-à-dire en enlevant 3 en `x` et en `y`).

Testez en passant en mode édition et en cliquant sur le Drawing Area. Pour avoir plusieurs points il faut pour le moment re-cliquer chaque fois sur le bouton radio "Add curve", c'est voulu.

6) Dans `on_area1_button_press`, pour le branchement `EDIT_ADD_CONTROL`, rajoutez un point de contrôle avec `add_control`. Dans `draw_curves`, modifiez la couleur de façon à ce que le point de contrôle courant de la courbe courante apparaisse en rouge.
Au début de cette fonction `draw_curves`, reliez tous les points de contrôle de chaque courbe par un segment, en jaune pour la courbe courante et en gris clair pour les autres.

7) Écrivez la fonction `int find_control (Curve_infos *ci, double x, double y)` qui reçoit des coordonnées `x`,`y`, puis cherche le premier point de contrôle situé à une distance $\leqslant 5$ de `x`,`y` (astuce : si l'écart entre les coordonnées du point de contrôle et `x`,`y` est stocké dans `dx`,`dy`, il suffit de tester si `dx*dx + dy*dy <= 5*5`). Si un tel point est trouvé, la fonction mémorise le numéro de courbe et de point de contrôle dans `current_curve` et `current_control` puis renvoie 0, sinon elle mémorise -1 dans ces champs puis renvoie -1.

Dans `on_area1_button_press`, pour le branchement `EDIT_MOVE_CONTROL`, cherchez le point de contrôle clické avec `find_control`. Vérifiez avec plusieurs courbes et les codes couleur, que vous arrivez à sélectionner un point et une courbe ou à les désélectionner en cliquant ailleurs.

8) Écrivez la fonction `int move_control (Curve_infos *ci, double dx, double dy)`, qui vérifie que la courbe courante et le point de contrôle courant existent (c'est-à-dire qu'ils sont dans les intervalles fermés `0 .. curve_count-1` et `0 .. control_count-1`), sinon renvoie -1. La fonction déplace ensuite le point de contrôle courant de l'écart `dx`, `dy` puis renvoie 0.

Réorganisez `on_area1_motion_notify` comme suit : on mémorise les coordonnées souris, puis si le bouton 1 est pressé et `my->show_edit` est vrai, on fait un branchement selon `my->edit_mode` sur l'un des 6 cas prévus puis on rafraîchit le Drawing Area.

Rajoutez dans `Mydata` des champs `double last_x, last_y`. Dans `on_area1_motion_notify`, juste avant d'enregistrer les nouvelles valeurs pour `click_x` et `click_y`, sauvegardez-les dans `last_x` et `last_y`. Enfin dans le mode `EDIT_MOVE_CONTROL`, appelez `move_control` en lui passant l'écart entre les nouvelles et précédentes coordonnées souris.

9) Écrivez la fonction `int move_curve (Curve_infos *ci, double dx, double dy)`, qui vérifie que la courbe courante existe, sinon renvoie -1. La fonction déplace ensuite tous les points de contrôle de la courbe courante de l'écart `dx`,`dy` puis renvoie 0.

Dans `on_area1_button_press`, pour le branchement `EDIT_MOVE_CURVE`, appelez `find_control`. Dans `on_area1_motion_notify`, pour le même mode, appelez `move_curve` en lui passant l'écart entre les nouvelles et précédentes coordonnées souris.

10) Écrivez la fonction `int remove_curve (Curve_infos *ci)`. Si `n = ci->current_curve` est hors de l'intervalle fermé `0 .. curve_count-1`, la fonction renvoie -1. La fonction supprime la courbe `n` en tassant le tableau, via cet appel de la fonction `memmove` de `<string.h>` :

```
memmove (ci->curve_list.curves+n, ci->curve_list.curves+n+1,  // dest, source
         sizeof(Curve)*(ci->curve_list.curve_count-1-n));     // taille
```

à la suite de quoi elle décrémente `curve_count`, met `current_curve` à -1 puis renvoie 0.

Dans `on_area1_button_press`, pour le mode `EDIT_REMOVE_CURVE`, si `find_control` trouve le point de contrôle cliqué, supprimez la courbe correspondante avec `remove_curve`.

11) Écrivez la fonction `int remove_control (Curve_infos *ci)` qui fait les vérifications habituelles, supprime le point de contrôle courant en tassant le tableau, voire supprime la courbe courante si elle est devenue vide. Branchez dans `on_area1_button_press`.

TP5 : courbes de Bézier

I. Préparation

Cette planche est la suite du TP4 ; il est indispensable de l'avoir terminé avant de commencer celle-ci.

1) Le programme commence à devenir très long et à ce stade on est convaincu de l'utilité de le découper en modules. Toutefois ce découpage implique de gérer les différentes versions des modules et de se doter des outils adéquats pour les compiler : des Makefiles.

Récupérez sur le site de l'UE le fichier archive tp05-bezier-v0.tgz

Décompactez le fichier archive en tapant la commande : `tar xvfz tp05-bezier-v0.tgz`

Ceci génère un dossier `TP05-bezier` contenant un `Makefile` principal, ainsi qu'un sous-répertoire `V0` contenant un `Makefile` et des fichiers `.c` et `.h`.

Le sous-répertoire `V0` contient la version 0 du programme du TP ; par la suite vous créerez des sous-répertoires `V1`, `V2`, etc pour contenir les versions successives du programme.

Pour compiler toutes les versions du programme, allez dans le répertoire `TP05-bezier` et tapez : `make all`. Vous pouvez également supprimer tous les fichiers compilés en tapant : `make clean` ; vous pouvez forcer la recompilation générale en tapant : `make clean all`. Enfin, vous pouvez créer un fichier archive de votre programme en tapant : `make tar` ; ce fichier s'appellera `TP05-bezier.tgz` et sera placé dans le répertoire parent.

Allons maintenant dans le sous-répertoire `V0`. Lui aussi contient un fichier `Makefile`, permettant de tout compiler (`make all`) ou tout nettoyer (`make clean`) ou encore forcer la recompilation générale (`make clean all`). Ce répertoire contient le module `font.c` accompagné de son `.h` qui a été présenté en cours, ainsi qu'un programme de démonstration `demo-font.c` qui l'utilise.

2) Créez une nouvelle version `V1` : allez dans le répertoire `TP05-bezier` et tapez : `cp -a V0 V1`

Dans `V1`, supprimez le fichier `demo-font.c`, puis recopiez votre programme du TP précédent, en le nommant par exemple `tp05-1.c`. Fermez votre éditeur puis rouvrez-le dans `V1`.

Éditez le `Makefile` de `V1`, et dans la ligne `CFILES = ...`, remplacez `demo-font.c` par `tp05-1.c` ; dans la ligne `EXEC = ...` remplacez `demo-font` par `tp05`. Enregistrez, puis dans le terminal tapez : `make all`

À ce stade, votre programme devrait compiler correctement et vous devriez pouvoir le lancer.

3) Dans le répertoire `V1`, créez un module `curve.c` et `curve.h`. Dans `curve.h` mettez une *garde* sous la forme :

```
#ifndef CURVE_H
#define CURVE_H

#endif /* CURVE_H */
```

Déplacez toutes les définitions de constantes et de types relatives aux courbes (à savoir : `CONTROL_MAX`, `CURVE_MAX`, `Control`, `Curve`, `Curve_list` et `Curve_infos`) qui sont actuellement dans `tp05-1.c` vers `curve.h` entre le `define` et le `endif`.

Déplacez le corps des fonctions manipulant les courbes (à savoir : `init_curve_infos`, `add_curve`, `remove_curve`, `add_control`, `find_control`, `move_control`, `move_curve`, `remove_control`) qui sont actuellement dans `tp05-1.c` vers `curve.c`. Au début de `curve.c` rajoutez `#include "curve.h"`. (Il faudra peut-être rajouter avant d'autres `#include`, par exemple `<string.h>` si vous avez utilisé la fonction `memmove`, ou encore `<stdio.h>` s'il y a des `printf`, etc.).

Dans `curve.h` rajoutez après la déclaration des types et avant la fin de la garde, les prototypes de toutes les fonctions implémentées dans `curve.c`.

Dans `tp05-1.c`, rajoutez `#include "curve.h"`.
Enfin dans le `Makefile`, rajoutez `curve.c` à la fin de la ligne `CFILES = ...`
Compilez avec `make clean all` et vérifiez qu'il n'y a pas d'erreurs.

4) De la même manière, créez un module `util.c` et `util.h`. Dans `util.c` déplacez les fonctions `refresh_area` et `set_status` ; au début incluez `<gtk/gtk.h>`.
Il va y avoir un problème avec `set_status` car actuellement cette fonction attend un paramètre `Mydata` qui est inconnu dans le module. Solution : remplacer le premier paramètre `Mydata *my` par `GtkWidget *status`, puis dans la fonction, remplacer `my->status` par `status` ; enfin dans `tp05-1.c`, remplacer les appels `set_status (my, ..)` par `set_status (my->status, ..)`.
Profitons-en pour appliquer le chapitre sur les fonctions variadiques vu en cours. Remplacez votre fonction `set_status` par celle figurant dans les transparents du cours, et mettez à jour le prototype dans `util.h`. Dans `tp05-1.c`, remplacez systématiquement les appels à `sprintf` + `set_status` par un appel à `set_status` exploitant le formatage. Par exemple vous remplacerez ce genre de code :
`char buf[100]; sprintf(buf, "valeur = %d", x); set_status (my->status, buf);`
par celui-ci : `set_status (my->status, "valeur = %d", x);`

5) Créez un module `mydata.c` et `mydata.h` ; déplacez la définition des énumérés `EDIT_NONE`, `EDIT_ADD_CURVE`, etc., ainsi que la déclaration du type `Mydata` dans `my_data.h` ; déplacez les fonctions `get_mydata`, `init_mydata` et `set_edit_mode` dans `mydata.c` ; dans ce dernier, incluez dans l'ordre : `<gtk/gtk.h>`, `"curve.h"`, `"mydata.h"`.

6) Créez un module `area1.c` et `area1.h` ; déplacez dans `area1.c` les fonctions suivantes : `apply_image_transforms`, `update_area1_with_transforms`, `draw_curves`, toutes les callbacks `on_area1_*`, et en dernier `area1_init`.
Dans `area1.h` déclarez les prototypes de toutes les fonctions sauf celles des callbacks (en effet, les callbacks sont privées et seul `area1_init` doit les connaître).
Dans `area1.c` incluez dans l'ordre : `<gtk/gtk.h>`, `"util.h"`, `"curve.h"`, `"mydata.h"`, et enfin `"area1.h"`.

7) Créez un module `menus.c` et `menus.h`. Dans `menus.c` déplacez toutes les callbacks `on_item-*` et en dernier la fonction `menus_init`. Dans `menus.h` ne déclarez que le prototype de `menus_init`. Que faut-il inclure dans `menus.c` ?

8) Créez un module `gui.c` et `gui.h` (pour "Graphical User Interface"). Dans `gui.c` déplacez toutes les fonctions de `tp05-1.c` sauf `main` et `on_app_activate` ; dans `gui.h` déclarez tous les prototypes des fonctions sauf ceux des callbacks.

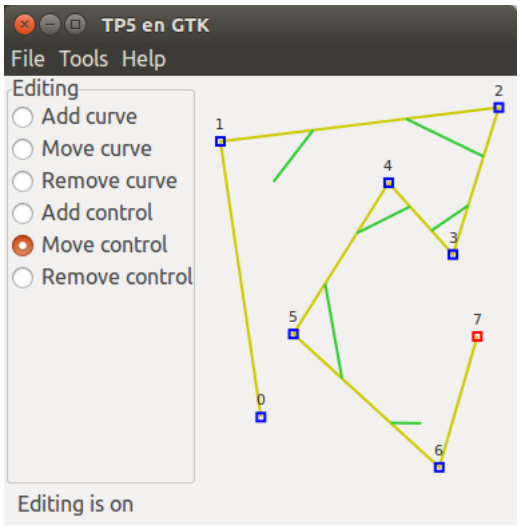
9) On y est : renommez `tp05-1.c` en `main-app.c` et mettez à jour le `Makefile`. Vous avez maintenant un programme découpé en 8 modules.

II. Tracé des courbes

1) Créez une nouvelle version V2 de votre programme : allez dans le répertoire `TP5-bezier`, tapez `make clean` puis `cp -a V1 V2`. Fermez votre éditeur puis rouvrez-le dans le répertoire `V2`.

2) Dans le module `area1.c` incluez `"font.h"`. Créez une fonction `void draw_control_labels (cairo_t *cr, PangoLayout *layout, Curve_infos *ci)` qui choisit la fonte `"Sans, 8"` grâce à la fonction `font_set_name`, fixe la couleur courante gris foncé avec `cairo_set_source_rgb`, puis pour chaque point de contrôle de chaque courbe, affiche le numéro du point de contrôle juste au dessus de lui, avec `font_draw_text`. Dans `on_area1_draw`, créez un `layout` comme vu en cours, appelez `draw_control_labels` puis déréférencez le `layout`.

3) Les points de contrôle actuels sont à partir de maintenant considérés comme des points de contrôle de courbes de B-splines cubiques uniformes. Pour dessiner les courbes correspondantes nous allons les convertir en courbes de Bézier équivalentes.



Dans `area1.c` et `area1.h`, renommez `draw_curves` en `draw_control_polygons`.

Dans le module `curve.c`, écrivez une fonction `void convert_bsp3_to_bezier (double p[4], double b[4])` qui convertit les coordonnées des points de contrôle de B-spline cubique uniforme p_0, p_1, p_2, p_3 en coordonnées de points de contrôle de Bézier cubique b_0, b_1, b_2, b_3 selon les formules vues en cours.

Dans le module `curve.c`, écrivez une fonction `void compute_bezier_points (Curve *curve, int i, Control bez_points[4])` qui prend en entrée les points de contrôle de la B-spline cubique uniforme stockés dans `curve` aux indices `i` à `i+3`, recopie les coordonnées dans des tableaux temporaires `px[4]` et `py[4]`, les convertit en appelant deux fois `convert_bsp3_to_bezier` respectivement dans des tableaux temporaires `bx[4]` et `by[4]`, puis recopie les résultats dans `bez_point`.

Dans le module `area1.c`, écrivez la fonction `void draw_bezier_polygons_open (cairo_t *cr, Curve_infos *ci)` qui pour chaque courbe de `ci`, pour chaque `i` entre 0 et le nombre de sommets de la courbe -3, appelle `compute_bezier_points` puis relie les sommets de Bézier d'indice 0 et 1 par un segment de couleur verte, et fait de même pour les sommets d'indice 2 et 3 (voir la figure ci-dessus, et le cours sur Cairo pour le dessin d'un segment).

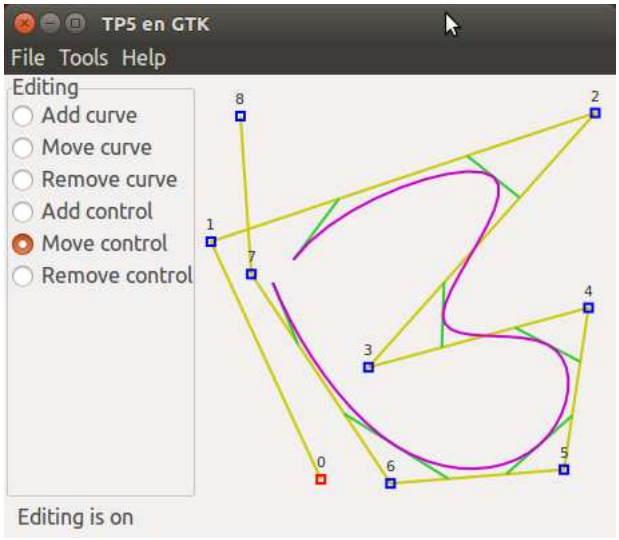
Enfin dans `on_area1_draw`, appelez `draw_bezier_polygons_open`.

4) Dans le module `curve.c`, écrivez une fonction `double compute_bezier_cubic (double b[4], double t)` qui reçoit des coordonnées de sommets de Bézier b_0, b_1, b_2, b_3 et $t \in [0, 1]$, puis renvoie la valeur du polynôme de Bézier cubique pour t.

Dans le module `area1.c`, écrivez la fonction `void draw_bezier_curve (cairo_t *cr, Control bez_points[4], double theta)` qui affiche la courbe de Bézier cubique pour les points de contrôle `bez_points` avec l'échantillonnage `theta`. Concrètement, il s'agit de calculer les coordonnées des points pour $t = 0$, $t = \theta$, $t = 2\theta$, ..., 1 avec `compute_bezier_cubic` et de les relier par des segments de droite, comme vu en cours. Pour vous simplifier la vie, recopiez au début de la fonction les coordonnées dans deux tableaux temporaires `bx[4]` et `by[4]`.

Dans le module `area1.c`, écrivez la fonction `void draw_bezier_curves_open (cairo_t *cr, Curve_infos *ci, double theta)` qui pour chaque courbe de `ci`, pour chaque `i` entre 0 et le nombre de sommets de la courbe -3, appelle `compute_bezier_points` puis `draw_bezier_curve`, de façon à tracer la courbe de Bézier en mauve.

Enfin dans `on_area1_draw`, appelez la fonction `draw_bezier_curves_open` avec `theta = 0.1`; agrandissez la fenêtre et essayez avec d'autres valeurs (par exemple `0.2` ou `0.02`).



TP6 : courbes de Bézier - suite

I. Préparation

Cette planche est la suite du TP5 ; il est indispensable de l'avoir terminé avant de commencer celle-ci.

Placez-vous dans votre répertoire `TP05-bezier` et créez une nouvelle version V3 de votre programme en tapant : `cp -a V2 V3`. Éditez le fichier `Makefile` et remplacez le nom de l'exécutable par `tp06`.

II. Courbes fermées ou prolongées

Nous allons rajouter des boutons radio pour choisir le type d'affichage des courbes B-Splines cubiques uniformes : courbe ouverte (le cas actuel), courbe fermée, courbe prolongée aux extrémités.

1) Dans `mydata.h` déclarez un `enum` avec les constantes : `BSP_FIRST`, `BSP_OPEN`, `BSP_CLOSE`, `BSP_PROLONG`, `BSP_LAST`. Ces constantes représenteront le mode d'affichage. Rajoutez un champ entier `bsp_mode` initialisé au mode `BSP_OPEN` ; rajoutez également le tableau `GtkWidget *bsp_radios[BSP_LAST]`.

2) Dans `gui.c`, dans la fonction `editing_init`, rajoutez dans le Box du `frame1` un `GtkSeparator` horizontal, puis un groupe de boutons radios intitulés "Opened", "Closed", "Prolongated", que vous mémoriserez dans le tableau `bsp_radios`. Associez pour chaque bouton la constante énumérée correspondante en vous servant de `g_object_set_data` avec le mot clé `"mode"`.

3) Attachez la callback `on_bsp_radio_toggled` à chaque bouton radio pour le signal `toggled` ; grâce à cette callback stockez le mode du bouton actif dans le champ `bsp_mode` et rafraîchissez le Drawing Area.

Dans `on_area1_draw`, rajoutez un branchement en fonction du champ `bsp_mode`, dans lequel la fonction `draw_bezier_curves_open` est appelée uniquement pour le cas `BSP_OPEN`.

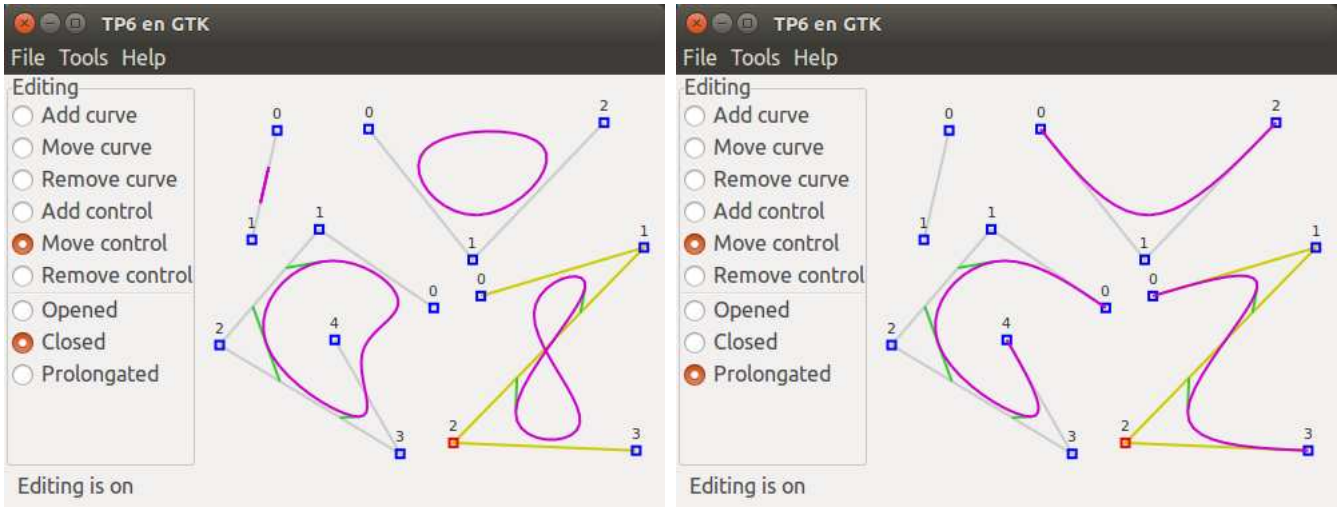
4) Dans le module `curve.c` et ailleurs si nécessaire, renommez la fonction `compute_bezier_points` en `compute_bezier_points_open`. Dupliquez cette fonction en `compute_bezier_points_close` et modifiez cette dernière de manière à ce qu'elle calcule les points de contrôle de Bézier pour une courbe fermée.

Concrètement, si le paramètre `i` est supérieur ou égal à `control_count-3`, il suffit de faire un modulo sur les indices dans la première phase de recopie des points.

5) Dans le module `area1.c`, dupliquez la fonction `draw_bezier_curves_open` en une fonction `draw_bezier_curves_close`, et modifiez cette dernière de façon à ce qu'elle affiche la courbe fermée.

Dans `on_area1_draw`, appelez cette fonction pour le mode `BSP_CLOSE`.

6) On s'occupe enfin du mode `BSP_PROLONG`, où la courbe est prolongée jusqu'aux extrémités du polygone de contrôle. Dans le module `curve.c` écrivez la fonction `void convert_bsp3_to_bezier_prolong_first (double p[3], double b[4])` qui reçoit les coordonnées des trois premiers points de contrôle dans `p[]` puis calcule les coordonnées des points de Bézier équivalents et les stocke dans `b[]` avec les formules : $B_0 = P_0$, $B_1 = (2P_0 + P_1)/3$, $B_2 = (P_0 + 2P_1)/3$, $B_3 = (P_0 + 4P_1 + P_2)/6$. De la même manière, écrivez la fonction `void convert_bsp3_to_bezier_prolong_last (double p[3], double b[4])` qui reçoit les coordonnées des trois derniers points de contrôle dans `p[]` puis calcule les coordonnées des points de Bézier équivalents et les stocke dans `b[]` avec les formules : $B_0 = (P_0 + 4P_1 + P_2)/6$, $B_1 = (2P_1 + P_2)/3$, $B_2 = (P_1 + 2P_2)/3$, $B_3 = P_2$.



7) Toujours dans `curve.c`, écrivez la fonction `void compute_bezier_points_prolong_first ( Curve *curve, Control bez_points[4])` qui calcule et stocke dans `bez_point[]` les coordonnées des 4 premiers points de Bézier équivalents pour la courbe prolongée aux extrémités. De façon analogue à `compute_bezier_points`, la fonction recopie les coordonnées des trois premiers points de contrôle dans des tableaux temporaires `px[3]` et `py[3]`, les convertit en appelant deux fois `convert_bsp3_to_bezier_prolong_first` respectivement dans des tableaux temporaires `bx[4]` et `by[4]`, puis recopie les résultats dans `bez_point`. Procédez de même dans l'écriture de la fonction `void compute_bezier_points_prolong_last (Curve *curve, Control bez_points[4])`.

8) Enfin dans `area1.c`, écrivez la fonction `void draw_bezier_curves_prolong (cairo_t *cr, Curve_infos *ci, double theta)` qui trace les courbes B-spline cubiques uniformes prolongées aux extrémités. On pourra utilement s'inspirer de `draw_bezier_curves_open`, en rajoutant simplement le tracé des tronçons de courbes au début et à la fin. Pour éviter des affichages intempestifs, cette fonction ne fera rien si le nombre de points de contrôles est $\leqslant 3$.

Dans `on_area1_draw`, appelez cette fonction pour le mode `BSP_PROLONG`.

III. Découpage d'images

Dans cette dernière partie, on se propose de découper une image avec des courbes fermées, puis de les déplacer. Créez une nouvelle version V4 de votre programme.

1) Dans `mydata.h` rajoutez dans le `enum` après `BSP_PROLONG` les modes `BSP_FILL` et `BSP_CLIP`. Dans `gui.c`, rajoutez des boutons radios intitulés "Fill" et "Clip image" après le bouton "Prolongated".

2) Dans `area1.c` écrivez la fonction `void generate_bezier_path (cairo_t *cr, Control bez_points[4], double theta, int is_first)` qui crée un chemin Cairo correspondant à la courbe de Bézier mais sans la tracer. Concrètement, il suffit de recopier le code de `draw_bezier_curve` avec les modifications suivantes :

- ▷ on ne place le premier point (avec un appel à `cairo_move_to`) que si `is_first` est vrai ;
- ▷ on supprime l'appel final à `cairo_stroke`.

3) Toujours dans `area1.c`, écrivez la fonction `void draw_bezier_curves_fill (cairo_t *cr, Curve_infos *ci, double theta)` qui traces les courbes B-spline cubiques uniformes fermées et remplies. Vous pourrez recopier le code de `draw_bezier_curves_close` en remplaçant l'appel de `draw_bezier_curve` par un appel à `generate_bezier_path`. Pour chacun des polygones de contrôle de `ci`, on passera `TRUE` en dernier paramètre à `generate_bezier_path` pour la première courbe de Bézier calculée, de façon à initialiser le chemin ; lorsque pour le polygône considéré, tous les chemins de Bézier auront été générés, on réalisera le remplissage de la courbe en appelant `cairo_fill`.

Dans `on_area1_draw`, appelez cette fonction pour le mode `BSP_FILL`.

4) Dans `area1.c` dupliquez la fonction `draw_bezier_curves_fill` en `void draw_bezier_curves_clip (cairo_t *cr, Curve_infos *ci, double theta, Mydata *my)`. Testez au début de la fonction si `my->pixbuf2` est non `NULL`, et dans ce cas mettez comme source `my->pixbuf2` avec `gdk_cairo_set_source_pixbuf`, sinon laissez la couleur mauve.

Dans `on_area1_draw` ne dessinez pas l'image si le mode est `BSP_CLIP` (sinon on ne verra pas la découpe de l'image !) ; enfin, appelez `draw_bezier_curves_clip` pour le mode `BSP_CLIP`.

5) Dans `mydata.h` rajoutez les modes `EDIT_MOVE_CLIP` et `EDIT_RESET_CLIP` après `EDIT_REMOVE_CONTROL`. Dans `gui.c`, rajoutez les boutons radio intitulés "Move clip" et "Reset clip" après le bouton "Remove control".

Dans `curve.h`, rajoutez dans le struct `Curve` deux champs réels `shift_x` et `shift_y`. Initialisez ces champs à 0 dans `add_curve`.

Dans le module `area1.c`, modifiez la fonction `draw_bezier_curves_clip` : fixez la source de couleur pour chaque courbe, en passant à `gdk_cairo_set_source_pixbuf` comme deux derniers paramètres les champs `shift_x` et `shift_y`.

6) Dans `curve.c` écrivez la fonction `int move_shift (Curve_infos *ci, double dx, double dy)` qui incrémente de `dx` et `dy`, les champs `shift_x` et `shift_y` de la courbe courante, respectivement. Elle renvoie 0 s'il y a une courbe courante, -1 sinon. Dans `area1.c`, rajoutez dans `on_area1_button_press` le cas `EDIT_MOVE_CLIP`, pour lequel vous appellerez `find_control`. Ajoutez aussi ce cas dans `on_area1_motion_notify`, pour lequel vous appellerez `move_curve` et `move_shift`, en leur donnant les mêmes incréments que dans le cas `EDIT_MOVE_CURVE`.

7) Dans `curve.c` écrivez la fonction `int reset_shift (Curve_infos *ci)`, qui remet à 0 les champs `shift_x` et `shift_y` de la courbe courante. Elle renvoie 0 s'il y a une courbe courante, -1 sinon. Dans `area1.c`, rajoutez dans `on_area1_button_press` le cas `EDIT_RESET_CLIP`, pour lequel vous appellerez `find_control` puis `reset_shift`.