Guide d'utilisation de gimp-plugin-template

Par Médéric Laumon et Jérémy Laumon

Le gimp-plugin-template est un modèle complet de plugin pour Gimp. Il permet de développer des projets volumineux assez facilement grâce à ses makefiles et configure inclus. Il suffit de modifier les fichiers sources se trouvant dans le dossier src pour faire son propre plugin. Cette documentation a été réalisée d'après la version 2.2.0 de gimp-plugin-template et a pour but d'expliquer comment construire un plugin de Gimp en utilisant ce modèle mais ne donne pas d'explications très étendues sur la programmation.

Pour en savoir plus sur la programmation de plugins Gimp vous pouvez lire le document « Writing-A-PlugIn » de Dave Neary et consulter la documentation de l'API Gimp (http://developer.gimp.org/api/2.0/index.html).

Table des matières

I. Où trouver la dernière version

La dernière version est disponible à l'adresse ftp://ftp.gimp.org/pub/gimp/plugin-template/

II. Compilation et installation

1. Linux

a. Prérequis

La compilation d'un plugin pour Gimp 2.x nécessite d'avoir le paquet libgimp2.0-dev et ses dépendances installés (gtk, cairo, pango, atk, glib, etc.).

b. Marche à suivre

1. Ouvrir une invite de commande et se placer dans le bon dossier
2. Taper ./configure afin de configurer le paquet en fonction de votre système
3. Taper make pour compiler le plugin
4. Taper make install pour installer le fichier binaire compilé dans le dossier des plugins de Gimp. Cette commande nécessite les droits d'administrateur pour copier le fichier dans le répertoire de Gimp. Une alternative à cette commande est de copier manuellement le fichier binaire dans ~/.gimp-2.0/plug-ins/ dans le cas où l'on n'a pas des droits suffisants.
5. Les fichiers créés par la commande make peuvent être supprimés en tapant make clean.

2. Windows

a. Prérequis

Pour compiler un plugin pour Gimp 2.x sous Windows, la solution la plus simple est probablement d'utiliser cygwin. Cygwin est un environnement Linux fonctionnant sur Windows et permettant de construire des applications Windows natives (http://www.cygwin.com pour le télécharger).

Le site http://jlhamel.club.fr/FILES/index.html explique très bien comment compiler un plugin basique avec cygwin. Il propose de télécharger une archive « gimp-2.4_dev » qui contient déjà toutes les librairies requises pour la compilation.
Pour l'utiliser, il faut l'extraire et modifier le script setgimp24.sh contenu à l'intérieur de façon à ce que les chemins des trois premiers « export » pointent correctement le dossier de Windows, le dossier où a été décompressée l'archive et le dossier de Gimp. Ces chemins sont au format Unix de cygwin, ce qui signifie que « D:\foo\bar » devient « /cygdrive/d/foo/bar ».

Pour compiler un plugin qui utilise gimp-plugin-template-2.2.0, il faut installer quelques paquets supplémentaires qui ne sont pas sélectionnés par défaut lors de l'installation de cygwin :
Devel : gcc-core
Devel : gettext-devel
Devel : glib-devel
Devel : make

Devel : pango
Libs : glib
Perl : perl-locale-gettext

Une fois tout cela installé, cygwin est prêt pour compiler gimp-plugin-template.

b. Marche à suivre

1. Lancer Cygwin et se placer dans le dossier où a été extrait « gimp2.4_dev »
2. Taper source setgimp24.sh
3. Se placer dans le dossier du plugin a compiler
4. Taper ./configure afin de configurer le paquet en fonction du système
5. Taper make pour compiler le plugin
6. Taper make install pour installer le plugin ou bien copier manuellement le .exe généré qui se trouve dans le dossier src, dans le dossier lib/gimp/2.0/plug-ins du répertoire ou est installé Gimp.
7. Les fichiers créés par la commande make peuvent être supprimés en tapant make clean.

III. Modifications

1. Description des fichiers source

Les fichiers sources se trouvent dans le sous répertoire src du dossier de gimp-plugin-template.

a. Main.c

Constantes et variables globales

```
#define PROCEDURE_NAME   "gimp_plugin_template"
#define DATA_KEY_VALS    "plug_in_template"
#define DATA_KEY_UI_VALS "plug_in_template_ui"
#define PARASITE_KEY     "plug-in-template-options"
```

Ces constantes servent entre autres à identifier certaines valeurs passées à Gimp. Elles doivent être uniques, donc à changer pour chaque plugin que l'on fait. On peut par exemple les adapter au nom du plugin : `plugin_detection_ellipse_ui` par exemple.

```
static PlugInVals         vals;
```

Cette structure contient tous les paramètres dont le plugin a besoin pour faire son travail. Par exemple, pour un plugin qui trace des cercles, cela pourrait être le diamètre, l'épaisseur du trait et la couleur. La définition du type PlugInVals se trouve dans main.h et doit être adaptée pour chaque plugin, les champs présents par défaut ne sont que des exemples et sont donc à supprimer.

```
const PlugInVals default_vals =
{
  0,
  1,
  2,
  0,
  FALSE
};
```

Ce sont les valeurs par défaut qui seront attribuées à la structure vals au début du run(). A adapter suivant votre déclaration du type PlugInVals.

Fonction query()

La fonction query() contient toutes les informations relatives à l'identification du plugin. Ses paramètres, son nom, son emplacement, sa description, où il doit apparaître dans l'interface de Gimp etc.

```
static void
query (void)
{
  gchar *help_path;
  gchar *help_uri;

  static GimpParamDef args[] =
  {
    { GIMP_PDB_INT32,    "run_mode",   "Interactive, non-interactive"    },
    { GIMP_PDB_IMAGE,    "image",      "Input image"                     },
    { GIMP_PDB_DRAWABLE, "drawable",   "Input drawable"                  },
    { GIMP_PDB_INT32,    "dummy",      "dummy1"                          },
    { GIMP_PDB_INT32,    "dummy",      "dummy2"                          },
    { GIMP_PDB_INT32,    "dummy",      "dummy3"                          },
    { GIMP_PDB_INT32,    "seed",       "Seed value (used only if randomize is FALSE)" },
    { GIMP_PDB_INT32,    "randomize",  "Use a random seed (TRUE, FALSE)" }
  };

  gimp_plugin_domain_register (PLUGIN_NAME, LOCALEDIR);

  help_path = g_build_filename (DATADIR, "help", NULL);
  help_uri = g_filename_to_uri (help_path, NULL, NULL);
  g_free (help_path);

  gimp_plugin_help_register ("http://developer.gimp.org/plug-in-template/help",
                             help_uri);

  gimp_install_procedure (PROCEDURE_NAME,
                      "Blurb",
                      "Help",
                      "Michael Natterer <mitch@gimp.org>",
                      "Michael Natterer <mitch@gimp.org>",
                      "2000-2004",
                      N_("Plug-In Template..."),
                      "RGB*, GRAY*, INDEXED*",
                      GIMP_PLUGIN,
                      G_N_ELEMENTS (args), 0,
                      args, NULL);

  gimp_plugin_menu_register (PROCEDURE_NAME, "<Image>/Filters/Misc/");
}
```

Ce qui est important :
GimpParamDef contient trois informations sur chaque paramètre : son type, son nom et une courte description. A adapter à votre déclaration du type PlugInVals.
Gimp_install_procedure() déclare :

- le nom du plugin
- une courte description
- une courte chaîne d'aide
- le nom de l'auteur
- le copyright

- la date
- le nom porté par le plugin dans l'interface Gimp
- le type d'images supportées par le plugin (l'étoile signifie que la transparence est acceptée). Un plugin peut n'accepter aucun type d'image. Il pourra dans ce cas être appelé lorsque aucune image n'est ouverte (c'est le cas par exemple pour un plugin qui doit traiter automatiquement toutes les images d'un dossier).

Gimp_plugin_menu_register() permet de choisir l'emplacement du plugin dans les rubriques de l'interface de Gimp. <Image> correspond à la fenêtre où l'image est ouverte alors que <Toolbox> correspond à la fenêtre principale de Gimp.

Fonction run()

Run() est la fonction appelée lorsque le plugin est exécuté.

```
static void
run (const gchar      *name,
     gint              n_params,
     const GimpParam  *param,
     gint             *nreturn_vals,
     GimpParam       **return_vals)
{
  static GimpParam   values[1];
  GimpDrawable      *drawable;
  gint32             image_ID;
  GimpRunMode        run_mode;
  GimpPDBStatusType  status = GIMP_PDB_SUCCESS;

  *nreturn_vals = 1;
  *return_vals  = values;

  /*  Initialize i18n support  */
  bindtextdomain (GETTEXT_PACKAGE, LOCALEDIR);
#ifdef HAVE_BIND_TEXTDOMAIN_CODESET
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
#endif
  textdomain (GETTEXT_PACKAGE);

  run_mode = param[0].data.d_int32;
  image_ID = param[1].data.d_int32;
  drawable = gimp_drawable_get (param[2].data.d_drawable);

  /*  Initialize with default values  */
  vals          = default_vals;
  image_vals    = default_image_vals;
  drawable_vals = default_drawable_vals;
  ui_vals       = default_ui_vals;

  if (strcmp (name, PROCEDURE_NAME) == 0)
    {
      switch (run_mode)
       {
       case GIMP_RUN_NONINTERACTIVE:
         if (n_params != 8)
           {
             status = GIMP_PDB_CALLING_ERROR;
           }
         else
           {
             vals.dummy1      = param[3].data.d_int32;
             vals.dummy2      = param[4].data.d_int32;
```

```
            vals.dummy3      = param[5].data.d_int32;
            vals.seed        = param[6].data.d_int32;
            vals.random_seed = param[7].data.d_int32;

             if (vals.random_seed)
               vals.seed = g_random_int ();
          }
        break;

      case GIMP_RUN_INTERACTIVE:
        /*  Possibly retrieve data  */
        gimp_get_data (DATA_KEY_VALS,    &vals);
        gimp_get_data (DATA_KEY_UI_VALS, &ui_vals);

        if (! dialog (image_ID, drawable,
                      &vals, &image_vals, &drawable_vals, &ui_vals))
          {
            status = GIMP_PDB_CANCEL;
          }
        break;

      case GIMP_RUN_WITH_LAST_VALS:
        /*  Possibly retrieve data  */
        gimp_get_data (DATA_KEY_VALS, &vals);

         if (vals.random_seed)
           vals.seed = g_random_int ();
        break;

      default:
        break;
      }
    }
  else
    {
      status = GIMP_PDB_CALLING_ERROR;
    }

  if (status == GIMP_PDB_SUCCESS)
    {
      render (image_ID, drawable, &vals, &image_vals, &drawable_vals);

      if (run_mode != GIMP_RUN_NONINTERACTIVE)
       gimp_displays_flush ();

      if (run_mode == GIMP_RUN_INTERACTIVE)
       {
         gimp_set_data (DATA_KEY_VALS,    &vals,    sizeof (vals));
         gimp_set_data (DATA_KEY_UI_VALS, &ui_vals, sizeof (ui_vals));
       }

      gimp_drawable_detach (drawable);
    }

  values[0].type = GIMP_PDB_STATUS;
  values[0].data.d_status = status;
}
```

Ce qui est important :

Le plugin peut être appelé dans trois « run mode » différents : interactive, non-interactive et run with last vals. Le switch(run_mode) permet de différencier ces trois cas :

```
case GIMP_RUN_NONINTERACTIVE:
        if (n_params != 8)
```

```
          {
            status = GIMP_PDB_CALLING_ERROR;
          }
        else
          {
            vals.dummy1      = param[3].data.d_int32;
            vals.dummy2      = param[4].data.d_int32;
            vals.dummy3      = param[5].data.d_int32;
            vals.seed        = param[6].data.d_int32;
            vals.random_seed = param[7].data.d_int32;

             if (vals.random_seed)
               vals.seed = g_random_int ();
          }
        break;
```

Dans ce cas là, toutes les variables nécessaires au plugin sont passées en paramètre lors de l'appel du plugin et sont ici copiées dans la structure vals. Cette situation se produit par exemple lorsque le plugin est appelé depuis un autre plugin.

```
case GIMP_RUN_INTERACTIVE:
        /*  Possibly retrieve data  */
        gimp_get_data (DATA_KEY_VALS,    &vals);
        gimp_get_data (DATA_KEY_UI_VALS, &ui_vals);

        if (! dialog (image_ID, drawable,
                      &vals, &image_vals, &drawable_vals, &ui_vals))
          {
            status = GIMP_PDB_CANCEL;
          }
        break;
```

Ce cas est le plus courant, c'est lorsque le plugin est appelé depuis l'interface graphique de Gimp. La fonction dialog() est alors appelée afin de déterminer les différents paramètres. Cette fonction affiche l'interface graphique du plugin. Elle est décrite dans interface.c et renvoie TRUE si l'utilisateur a cliqué sur le bouton OK, FALSE sinon.

```
case GIMP_RUN_WITH_LAST_VALS:
        /*  Possibly retrieve data  */
        gimp_get_data (DATA_KEY_VALS, &vals);

         if (vals.random_seed)
           vals.seed = g_random_int ();
        break;
```

Ce cas se produit lorsque le plugin est appelé depuis « Filtres/Répéter le dernier ». Les valeurs de vals précédentes sont automatiquement retrouvées grâce à gimp_get_data().

```
if (status == GIMP_PDB_SUCCESS)
    {
      render (image_ID, drawable, &vals, &image_vals, &drawable_vals);

      if (run_mode != GIMP_RUN_NONINTERACTIVE)
       gimp_displays_flush ();

      if (run_mode == GIMP_RUN_INTERACTIVE)
       {
         gimp_set_data (DATA_KEY_VALS,    &vals,    sizeof (vals));
         gimp_set_data (DATA_KEY_UI_VALS, &ui_vals, sizeof (ui_vals));
       }

      gimp_drawable_detach (drawable);
    }
```

Si il n'y a pas de problème, la fonction render() est ensuite appelée. Gimp_display_flush() permet de

mettre à jour l'affichage et gim_set_data() de sauvegarder les valeurs dans le cas d'un prochain appel en « run with last vals ». Gimp_drawable_detach() permet entre autre de libérer la mémoire allouée et doit être appelée lorsque le plugin a fini son travail.

b. Render.c

Le render.c contient le traitement de l'image, les actions que le plugin effectue.

```
void
render (gint32              image_ID,
       GimpDrawable       *drawable,
       PlugInVals         *vals,
       PlugInImageVals    *image_vals,
       PlugInDrawableVals *drawable_vals)
{
  g_message (_("This plug-in is just a dummy. "
               "It has now finished doing nothing."));
}
```

Par défaut, il n'y a aucun traitement sur l'image, simplement un message.

Voici un algorithme basique permettant d'accéder aux données de l'image :

```
gint x1, x2, y1, y2;
gint width, height;
gint i, j, channels;
guchar *row;
GimpPixelRgn rgn_in, rgn_out;

gimp_drawable_mask_bounds (drawable->drawable_id,
                                          &x1, &y1,
                                          &x2, &y2);
width = x2 - x1;
height = y2 - y1;
channels = gimp_drawable_bpp (drawable->drawable_id);

gimp_pixel_rgn_init (&rgn_in,
                     drawable,
                     x1, y1,
                     width, height,
                     FALSE, FALSE);

gimp_pixel_rgn_init (&rgn_out,
                     drawable,
                     x1, y1,
                     width, height,
                     TRUE, TRUE);

row = (guchar*)malloc(sizeof(guchar)*channels*width);

for (i = 0; i < height; i++)
{
       gimp_pixel_rgn_get_row (&rgn_in, row, x1, i+y1, width);

       for (j = 0; j < width; j++)
       {

              // Traitement de la ligne de pixels

       }
       gimp_pixel_rgn_set_row (&rgn_out, row, x1, i+y1, width);
}
```

```
gimp_drawable_flush (drawable);
gimp_drawable_merge_shadow (drawable->drawable_id, TRUE);
gimp_drawable_update (drawable->drawable_id, x1, y1, x2 - x1, y2 – y1);
```

- Gimp_drawable_mask_bounds() permet de récupérer les coordonnées des limites de la sélection afin de ne faire les calculs que sur cette zone.
- Gimp_drawable_bpp() renvoie le nombre d'octets par pixel. Par exemple, une image RGB a trois octets par pixels alors qu'une image en niveau de gris n'en a qu'un.
- Gimp_pixel_rgn_init() permet d'initialiser une structure GimpPixelRgn, qui permet d'accéder aux données. Les paramètres sont différents si l'on veut lire ou écrire des données. Plus d'informations sur ce sujet peuvent être trouvées dans l'API de Gimp mais généralement, il faut passer comme derniers paramètres FALSE et FALSE lorsqu'on veut lire des données et TRUE, TRUE lorsqu'on veut écrire.
- Dans cet exemple, les données vont être lues ligne par ligne mais il est aussi possible de lire par pixel, par colonne ou par rectangle (cf. API). Les données sont stockées dans le tableau row de taille width*channels de char et la fonction gimp_pixel_rgn_get_row() permet de les récupérer. Comme le tableau est un tableau de char, cela signifie que pour une image couleur les données sont stockées comme suit : [R][G][B][R][G][B] etc.
- Gimp_pixel_rgn_set_row() permet d'écrire les données modifiées dans la GimpPixelRgn rgn_out qui va ensuite être envoyée à Gimp pour l'affichage grâce aux trois fonctions gimp_drawable_flush(), gimp_drawable_merge_shadow() et gimp_drawable_update().

c. Interface.c

Interface.c gère l'affichage de l'interface du plugin. Grâce à cette interface, l'utilisateur va pouvoir définir les paramètres du plugin.
L'interface est gérée grâce à la bibliothèque GTK+. Les possibilités de GTK+ étant très étendues, son utilisation ne sera pas expliquée ici. Des tutoriels peuvent être trouvés partout sur le net (http://www.gtk-fr.org en propose un très bien) et le logiciel Glade peut aussi beaucoup faciliter la création d'une interface.

```
gboolean
dialog (gint32              image_ID,
       GimpDrawable       *drawable,
       PlugInVals         *vals,
       PlugInImageVals    *image_vals,
       PlugInDrawableVals *drawable_vals,
       PlugInUIVals       *ui_vals)
{

  ...

  run = (gimp_dialog_run (GIMP_DIALOG (dlg)) == GTK_RESPONSE_OK);

  gtk_widget_destroy (dlg);

  return run;
}
```

Par défaut, cette fonction est remplie d'exemples de l'utilisation de GTK+ qui ne sont pas repris ici.
La fonction gimp_dialog_run() appelle un équivalent de boucle principale qui se termine lorsque l'utilisateur clique sur OK ou Annuler. Toutes les déclarations et les placements des GtkWidget sont à placer avant l'appel de cette fonction.

2. Ajouter ou retirer des fichiers sources ou des librairies

Pour facilement modifier les fichiers et les librairies inclus à la compilation, il faut modifier le fichier src/Makefile.am.
La ligne `gimp_plugin_template_SOURCE =...` contient la liste des fichiers sources.
La ligne `INCLUDES =...` contient la liste des dossiers à inclure (cf. option -I de GCC).
La ligne `LDADD =...` contient la liste des librairies à inclure (cf. option -L de GCC).
Chaque fois que ce fichier est modifié, il faut lancer le script autogen.sh qui va générer un nouveau configure et le lancer. Le nouveau configure va quant à lui générer de nouveaux Makefiles qui prendront en compte les modifications apportées au fichier Makefile.am. L'exécution de ce script nécessite les paquets automake et autoconf (attention, les versions supérieures à 1.9 de autoconf semblent ne pas être détectées correctement).

3. Modifier le nom et la version du plugin

Le nom et la version du plugin sont contenus dans le fichier configure.in. De même que pour Makefile.am, chaque fois que configure.in est modifié, il faut que autogen.sh soit exécuté pour que les modifications soient prises en compte.

La ligne `m4_define([plugin_name], [gimp-plugin-template])` contient le nom du fichier. Il suffit de remplacer `gimp-plugin-template` par le nouveau nom.
Attention, deux lignes doivent aussi être modifiées dans le fichier src/Makefile.am afin de changer le nom de l'exécutable produit :
`bin_PROGRAMS = gimp-plugin-template` qui va déterminer le nom du fichier exécutable.
`gimp_plugin_template_SOURCES =...` qui doit porter le même nom que la ligne précédente en remplaçant les '–' par des '_' et en ajoutant '_SOURCES' à la fin. Si ces deux lignes ne correspondent pas l'une à l'autre, la compilation échouera.
Ces modifications ne sont pas obligatoire mais la logique voudrait que l'exécutable porte le même nom que le plugin.

Les lignes :
```
m4_define([plugin_major_version], [2])
m4_define([plugin_minor_version], [2])
```
`m4_define([plugin_micro_version], [0])` contiennent les trois chiffres de version du plugin.