Written By Dave Neary

PART I

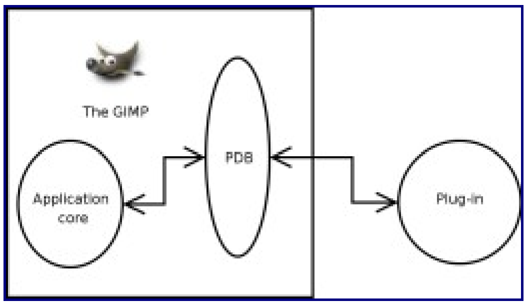
In this article, I present GIMP plug-ins basics and introduce the libgimp API. I will also show how to use the PDB to make our plug-in available to other script authors.

Introduction

New developers are often intimidated by The GIMP size and its reputation. They think that writing a plug-in would be a difficult task. The goal of these articles is to dumb this feeling down, by showing how easily one can make a C plug-in.

In this part, I present a plug-in's basic elements. We will see how to install a plug-in and how to get data from an image and directly manipulate it.

Architecture



The GIMP script interface is centered on the Procedural database (PDB). At startup, The GIMP looks into a predefined set of places for scripts and plug-ins, and asks each new script to identify itself.

The plug-in declares itself to the PDB at that time, and passes informations like the position it wishes to get in the menu hierarchy, input parameters, and output parameters.

When a script or a plug-in wants to use our plug-in, it gets through the PDB, which manages communicating parameters in one direction and the other in a transparent way.

Internal functions that wish to get exposed to plug-ins have to be packaged first in the core, that will register them in the PDB, and secondly in the libgimp that will allow the function to be called as a normal one.

This was the introduction - now, we will look closer at our first plug-in, a "Hello, world!".

Compiling the plug-in

To be able to compile simple plug-ins for The GIMP, one needs libgimp headers, as well as an associated utility named gimptool.

With that utility, one can install a plug-in either in a private directory (~/.gimp-2.0/plug-ins), or in the global plug-in directory.

Syntax is

```
gimptool-2.0 --install plugin.c or gimptool --install-admin plugin.c
```

This utility, with other options, can also be used to install scripts, or uninstall plug-ins.

Behaviour

A GIMP plug-in can typically behave three different ways. It can take image data, modify it, and send back the modified image, like edge detection. It can generate an image and send it back, like some script-fus, or file reading plug-ins like jpeg. Or it can get an image, and process it without modifying its data, like a file saver plug-in.

Essentials

```
#include <libgimp/gimp.h>
```

This header makes all basic plug-in elements available to us.

```
GimpPlugInInfo PLUG_IN_INFO = {
  init,
  quit,
  query,
  run
};
```

This structure has to have that name. It contains four pointers to functions, which will be called at set times of the plug-in life. init and quit are optional, and thus can hold NULL values, but the last two functions, query and run, are mandatory.

The init() function is called each time The GIMP starts up. This function is not typically used. Some plug-ins use it to make a secondary search that is not done by the core. This function is not used by any standard GIMP plug-in, but could be useful for example for a plug-in that would like to register some procedure conditionally on some files presence.

The quit() function is not used much either. It is called when The GIMP is about to be closed, to allow it to free some resources. It is used in the script-fu plug-in.

The query() function is called the first time the plug-in is present, and then each time the plug-in changes.

The run() function is the plug-in's centrepiece. It is called when the plug-in is asked to run. It gets the plug-in name (as a plug-in can register several procedures), input parameters, and a pointer to output parameters, then determines if it is launched in a interactive way or by a script, and does all the plug-in processing. Its prototype is

```
void run (const gchar      *name,
          gint              nparams,
          const GimpParam  *param,
          gint             *nreturn_vals,
                  GimpParam       **return_vals);
```

MAIN ()

MAIN is a C macro that holds a bit of dark magic to initialise arguments. It also calls the appropriate PLUG_IN_INFO function depending on the timing. Your plug-in needs it.

The query() function

query() deals with the procedure registration and input arguments definition. These informations are saved to speed up startup time, and refreshed only when the plug-in is modified.

For our "Hello, world!" plug-in, the query function will look like this:

```
static void
query (void)
  {
    static GimpParamDef args[] = {
      {
        GIMP_PDB_INT32,
        "run-mode",
        "Run mode"
      },
      {
        GIMP_PDB_IMAGE,
        "image",
        "Input image"
      },
      {
        GIMP_PDB_DRAWABLE,
        "drawable",
        "Input drawable"
      }
    };

    gimp_install_procedure (
      "plug-in-hello",
      "Hello, world!",
      "Displays \"Hello, world!\" in a dialog",
      "David Neary",
      "Copyright David Neary",
      "2004",
      "_Hello world...",
      "RGB*, GRAY*",
      GIMP_PLUGIN,
      G_N_ELEMENTS (args), 0,
      args, NULL);

      gimp_plugin_menu_register ("plug-in-hello",
                                 "/Filters/Misc");
  }
```

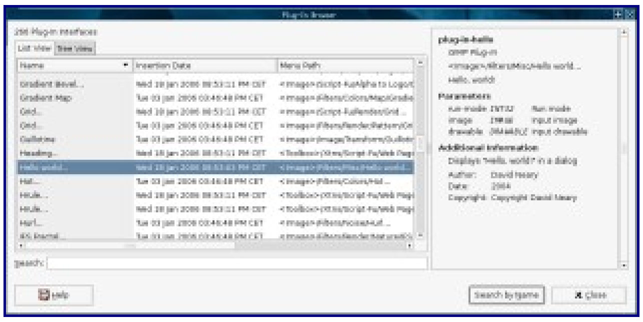
GimpParamDef contains three things - the parameter type, its name, and a string describing the parameter.

gimp_install_procedure declares the procedure name, some description and help strings, menu path where the plug-in should sit, image types handled by the plug-in, and at the end, input and output parameters number, as well as the parameters descriptors.

"RGB*, GRAY*" declares the image types handled. It can be RGB, INDEXED or GRAY, with or without Alpha. So "RGB*, GRAY*" describes RGB, RGBA, GRAY or GRAY image type.

GIMP_PLUGIN declares this procedure to be external, and not to be executed in The GIMP core.

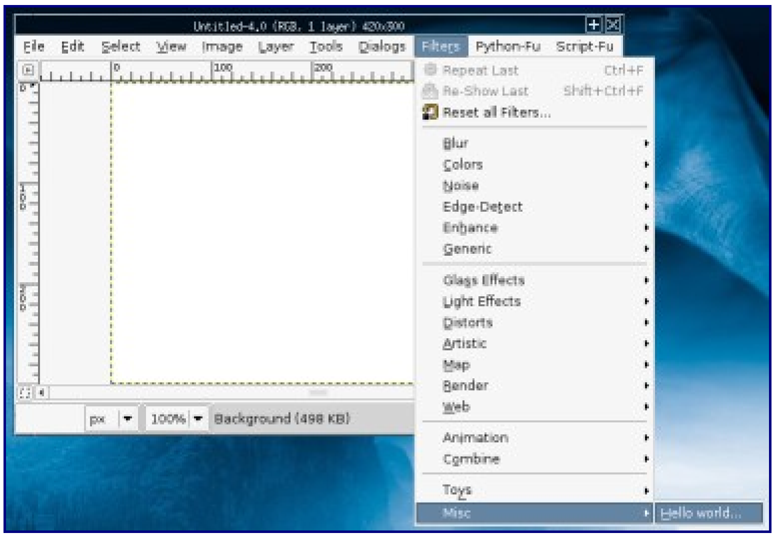
By adding a stub run function now, we can check that our plug-in has all the essential elements, and test that it registers itself in the PDB with the "Xtns->Plug-in Details" plug-in.



plug-in-hello
GIMP Plug-in
<Image>/Filters/Misc/Hello world...
Hello, world!

Parameters
run-mode INT32 Run mode
image IMAGE Input image
drawable DRAWABLE Input drawable

Additional Information
Displays "Hello, world!" in a dialog
Author: David Neary
Date: 2004
Copyright: Copyright David Neary



The other required function for PLUG_IN_INFO is run. The core of the plug-in stands there.

Output values (return_vals in the prototype) must have at least one value associated - the plug-in status. Typically, this parameter will hold "GIMP_PDB_SUCCESS".

Run-modes

One can run a plug-in in several different ways, it can be run from a GIMP menu if The GIMP is run interactively, or from a script or a batch, or from the "Filters->Repeat Last" shortcut.

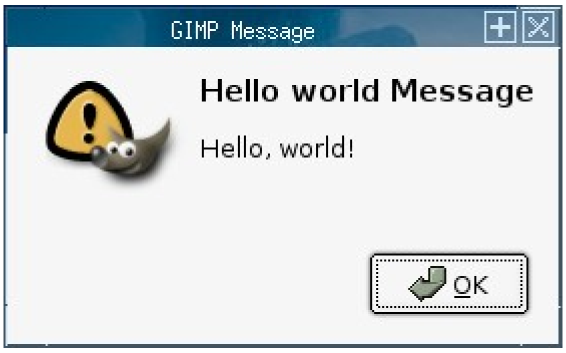
The "run_mode" input parameter can hold one of these values: "GIMP_RUN_INTERACTIVE", "GIMP_RUN_NONINTERACTIVE" or "GIMP_RUN_WITH_LAST_VALS".

"GIMP_RUN_INTERACTIVE" is typically the only case where one creates an options dialog. Otherwise, one directly calls the processing with values from input parameters or from memory.

For our test plug-in, we will simply display a dialog containing a "Hello, world!" message. Thankfully, this is really easy with GTK+. Our run function could be:

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

  /* Setting mandatory output values */
  *nreturn_vals = 1;
  *return_vals  = values;

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

  /* Getting run_mode - we won't display a dialog if
   * we are in NONINTERACTIVE mode */
  run_mode = param[0].data.d_int32;

  if (run_mode != GIMP_RUN_NONINTERACTIVE)
    g_message("Hello, world!\n");
}
```

Now, when we run our plug-in, there is action:



Hello world Message

Hello, world!

PART II

In the first part, I presented essential elements to build a plug-in interface with The GIMP. Now we will produce a simple but useful algorithm that we could use in our plug-in.

Introduction

The algorithm we are going to implement is a simple blur. It is included in The GIMP as "Filters->Blur->Blur" with default parameters.

That algorithm is very simple. Each pixel in our image is replaced by a mean value of its neighbours. For example, if we look at the simplest case where the neighbourhood is 3x3 (see figure 1), in that case the center value will be replaced with 5, the mean of the 9 numbers in its neighbourhood.

With this method, edge differences are splatted, giving a blurred result. One can choose another radius, using a (2r + 1) x (2r + 1) matrix.

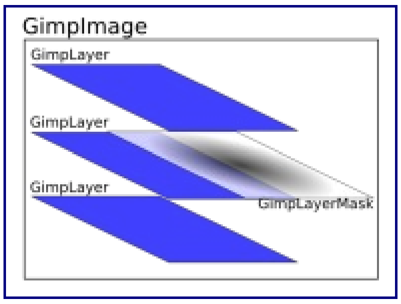
Image structure

Last month, we wrote a run() function that did nothing useful. Let's look again at run() prototype:

```
static void run (const gchar     *name,
                 gint             nparams,
                 const GimpParam *param,
                 gint            *nreturn_vals,
                 GimpParam      **return_vals);
```

We saw that for a filter (i.e. a plug-in that modifies the image), the first three input parameters were the run mode, an identifier for the image, and another one for the active drawable (layer or mask).

A GIMP image is a structure that contains, among others, guides, layers, layer masks, and any data associated to the image. The word "drawable" is often used in GIMP internal structures. A "drawable" is an object where you can get, and sometimes modify, raw data. So : layers, layer masks, selections are all "drawables".



Accessing the data

To get a GimpDrawable from its identifier, we need the gimp_drawable_get() function:

```
GimpDrawable *gimp_drawable_get (gint32 drawable_id);
```

From this structure, one can access drawable data through a GimpPixelRgn structure, and one can check the drawable type (RGB, gray level). The full listing of functions available for a GimpDrawable can be found in the API.

Two very important functions for plug-ins are gimp_drawable_mask_bounds() and gimp_pixel_rgn_init(). The first gives the active selection limits on the drawable, and the second initialises the GimpPixelRgn we will use to access the data.

As soon as we have a well initialised GimpPixelRgn, we can access the image data in several different ways, by pixel, by rectangle, by row or by column. The best method will depend on the algorithm one plans to use. Moreover, The GIMP uses a tile-based architecture, and loading or unloading data is expensive, so we should not use it more than necessary.

The main functions to get and set image data are:

```
      void gimp_pixel_rgn_get_pixel (GimpPixelRgn *pr,
                                     guchar       *buf,
                                     gint          x,
                                     gint          y);
      void gimp_pixel_rgn_get_row   (GimpPixelRgn *pr,
                                     guchar       *buf,
                                     gint          x,
                                     gint          y,
                                     gint          width);
      void gimp_pixel_rgn_get_col   (GimpPixelRgn *pr,
                                     guchar       *buf,
                                     gint          x,
                                     gint          y,
                                     gint          height);
      void gimp_pixel_rgn_get_rect  (GimpPixelRgn *pr,
                                     guchar       *buf,
                                     gint          x,
                                     gint          y,
                                     gint          width,
                                     gint          height);
      void gimp_pixel_rgn_set_pixel (GimpPixelRgn *pr,
                                     const guchar *buf,
                                     gint          x,
                                     gint          y);
      void gimp_pixel_rgn_set_row   (GimpPixelRgn *pr,
                                     const guchar *buf,
                                     gint          x,
                                     gint          y,
                                     gint          width);
      void gimp_pixel_rgn_set_col   (GimpPixelRgn *pr,
                                     const guchar *buf,
                                     gint          x,
                                     gint          y,
                                     gint          height);
      void gimp_pixel_rgn_set_rect  (GimpPixelRgn *pr,
                                     const guchar *buf,
                                     gint          x,
                                     gint          y,
                                     gint          width,
                                     gint          height);
```

There is also another way to access image data (it's even used more often), that allows to manage data at the tile level. We will look at it in detail later.

Updating the image

At last, a plug-in that has modified a drawable data must flush it to send data to the core, and to tell the application that the display must be updated. This is done with the following function:

```
gimp_displays_flush ();
gimp_drawable_detach (drawable);
```

Implementing blur()

To be able to try out several different processing methods, we will delegate the job to a blur() function. Our run() is below.

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

  /* Setting mandatory output values */
  *nreturn_vals = 1;
  *return_vals  = values;

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

  /* Getting run_mode - we won't display a dialog if
   * we are in NONINTERACTIVE mode */
  run_mode = param[0].data.d_int32;

  /*  Get the specified drawable  */
  drawable = gimp_drawable_get (param[2].data.d_drawable);

  gimp_progress_init ("My Blur...");
```

```
    /* Let's time blur
     *
     *   GTimer timer = g_timer_new time ();
     */

    blur (drawable);

    /*   g_print ("blur() took %g seconds.\n", g_timer_elapsed (timer));
     *   g_timer_destroy (timer);
     */

    gimp_displays_flush ();
    gimp_drawable_detach (drawable);
  }
```

There are a few lines here that need to be explained a bit more. The call to gimp_progress_init() initialises a progress measurement for our plug-in. Later, if we call gimp_progress_update(double percent), the percentage given as an input parameter will be shown graphically. The run_mode tells us whether the plug-in was launched in a way such as we can display a graphical interface or not. Possible values are GIMP_RUN_INTERACTIVE, GIMP_RUN_NONINTERACTIVE or GIMP_RUN_WITH_LAST_VALS, which mean the plug-in was executed from The GIMP, from a script, or from the "Repeat last filter" menu entry.

Regarding the blur algorithm itself, the first version using gimp_pixel_rgn_(get|set)_pixel() is found below. Some functions in it have not been explained yet.

gimp_drawable_mask_bounds() allows calculation of the filter's effect limits, excluding any region that is not in the active selection. Limiting the processing this way allows an important performance improvement.

gimp_pixel_rgn_init() takes as input parameters the drawable, its limits for the processing, and two booleans that significantly modify the behaviour of the resulting GimpPixelRgn. The first one tells that "set" operations must be done on shadow tiles, in order to leave original data as is until gimp_drawable_merge_shadow() is called, when all modified data will be merged. The second one tells that modified tiles should be tagged "dirty" and sent to the core to be merged. Most of the time, to read data, one uses FALSE and FALSE for these two parameters, and to write data, one uses TRUE and TRUE. Other combinations are possible but seldom used.

```
  static void
  blur (GimpDrawable *drawable)
  {
    gint         i, j, k, channels;
    gint         x1, y1, x2, y2;
    GimpPixelRgn rgn_in, rgn_out;
    guchar       output[4];

    /* Gets upper left and lower right coordinates,
     * and layers number in the image */
    gimp_drawable_mask_bounds (drawable->drawable_id,
                               &x1, &y1,
                               &x2, &y2);
    channels = gimp_drawable_bpp (drawable->drawable_id);

    /* Initialises two PixelRgns, one to read original data,
     * and the other to write output data. That second one will
     * be merged at the end by the call to
     * gimp_drawable_merge_shadow() */
```

```
gimp_pixel_rgn_init (&rgn_in,
                     drawable,
                     x1, y1,
                     x2 - x1, y2 - y1,
                     FALSE, FALSE);
gimp_pixel_rgn_init (&rgn_out,
                     drawable,
                     x1, y1,
                     x2 - x1, y2 - y1,
                     TRUE, TRUE);

for (i = x1; i < x2; i++)
  {
    for (j = y1; j < y2; j++)
      {
        guchar pixel[9][4];

        /* Get nine pixels */
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[0],
                                  MAX (i - 1, x1),
                                  MAX (j - 1, y1));
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[1],
                                  MAX (i - 1, x1),
                                  j);
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[2],
                                  MAX (i - 1, x1),
                                  MIN (j + 1, y2 - 1));

        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[3],
                                  i,
                                  MAX (j - 1, y1));
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[4],
                                  i,
                                  j);
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[5],
                                  i,
                                  MIN (j + 1, y2 - 1));

        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[6],
                                  MIN (i + 1, x2 - 1),
                                  MAX (j - 1, y1));
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[7],
                                  MIN (i + 1, x2 - 1),
                                  j);
        gimp_pixel_rgn_get_pixel (&rgn_in,
                                  pixel[8],
                                  MIN (i + 1, x2 - 1),
                                  MIN (j + 1, y2 - 1));

        /* For each layer, compute the average of the
         * nine */
        for (k = 0; k < channels; k++)
          {
            int tmp, sum = 0;
```

```
              for (tmp = 0; tmp < 9; tmp++)
                sum += pixel[tmp][k];
              output[k] = sum / 9;
            }

          gimp_pixel_rgn_set_pixel (&rgn_out,
                                    output,
                                    i, j);
        }

      if (i % 10 == 0)
        gimp_progress_update ((gdouble) (i - x1) / (gdouble) (x2 - x1));
    }

  /*  Update the modified region */
  gimp_drawable_flush (drawable);
  gimp_drawable_merge_shadow (drawable->drawable_id, TRUE);
  gimp_drawable_update (drawable->drawable_id,
                        x1, y1,
                        x2 - x1, y2 - y1);
}
```

Row processing

Our function has a bug drawback: performance. On a 300x300 selection, with the timing code uncommented, blur() took 12 minutes on my K6-2 350MHz, well loaded with other stuff. To compare, on the same selection, Gaussian blur took 3 seconds.

If we modify our function to rather use gimp_pixel_rgn_(get|set)_row() the result is far better. We reduce the timing for the 300x300 selection from 760 seconds to 6 seconds. blur() V2 is below:

```
static void
blur (GimpDrawable *drawable)
{
  gint         i, j, k, channels;
  gint         x1, y1, x2, y2;
  GimpPixelRgn rgn_in, rgn_out;
  guchar      *row1, *row2, *row3;
  guchar      *outrow;

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

  gimp_pixel_rgn_init (&rgn_in,
                       drawable,
                       x1, y1,
                       x2 - x1, y2 - y1,
                       FALSE, FALSE);
  gimp_pixel_rgn_init (&rgn_out,
                       drawable,
                       x1, y1,
                       x2 - x1, y2 - y1,
                       TRUE, TRUE);

  /* Initialise enough memory for row1, row2, row3, outrow */
  row1 = g_new (guchar, channels * (x2 - x1));
  row2 = g_new (guchar, channels * (x2 - x1));
```

```
        row3 = g_new (guchar, channels * (x2 - x1));
        outrow = g_new (guchar, channels * (x2 - x1));

        for (i = y1; i < y2; i++)
          {
            /* Get row i-1, i, i+1 */
            gimp_pixel_rgn_get_row (&rgn_in,
                                    row1,
                                    x1, MAX (y1, i - 1),
                                    x2 - x1);
            gimp_pixel_rgn_get_row (&rgn_in,
                                    row2,
                                    x1, i,
                                    x2 - x1);
            gimp_pixel_rgn_get_row (&rgn_in,
                                    row3,
                                    x1, MIN (y2 - 1, i + 1),
                                    x2 - x1);

            for (j = x1; j < x2; j++)
              {
                /* For each layer, compute the average of the nine
                 * pixels */
                for (k = 0; k < channels; k++)
                  {
                    int sum = 0;
                    sum = row1[channels * MAX ((j - 1 - x1), 0) + k]           +
                          row1[channels * (j - x1) + k]                        +
                          row1[channels * MIN ((j + 1 - x1), x2 - x1 - 1) + k] +
                          row2[channels * MAX ((j - 1 - x1), 0) + k]           +
                          row2[channels * (j - x1) + k]                        +
                          row2[channels * MIN ((j + 1 - x1), x2 - x1 - 1) + k] +
                          row3[channels * MAX ((j - 1 - x1), 0) + k]           +
                          row3[channels * (j - x1) + k]                        +
                          row3[channels * MIN ((j + 1 - x1), x2 - x1 - 1) + k];
                    outrow[channels * (j - x1) + k] = sum / 9;
                  }

             }

             gimp_pixel_rgn_set_row (&rgn_out,
                                     outrow,
                                     x1, i,
                                     x2 - x1);

             if (i % 10 == 0)
                  gimp_progress_update ((gdouble) (i - y1) / (gdouble) (y2 -
y1));
        }

        g_free (row1);
        g_free (row2);
        g_free (row3);
        g_free (outrow);

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

PART III

In the second part, I told you about manipulating image data by pixel or row. This time, I will go farther and process data by tile, which will improve our plug-in performance. I will also update our algorithm to take larger radius into account, and build a graphical interface to allow changing that parameter.

Introduction

Let's have a look at our simple algorithm: for each pixel, generate a (2r+1)x(2r+1) neighbourhood and for each layer, replace the layer's pixel value with the average value in the neighbourhood.

It's a bit more complex than that - we have to be careful near image borders for example, but this algorithm makes a blur effect that is not so bad in general.

But until now, we wrote the algorithm for a 3x3 neighbourhood. Time has come to generalise this part and to introduce the radius as a parameter.

First, a word on tiles.

Tile management

A tile is an image data block with a 64x64 size. Usually, tiles are sent to the plug-in on demand one by one, by shared memory. Of course this process needs huge resources and should be avoided.

Usually, one doesn't need any particular cache, each tile is sent when one needs it and freed when one asks for another one. Nevertheless, we can tell our plug-in to keep a tile cache to avoid this constant round trip, by calling the function:

```
      gimp_tile_cache_ntiles (gulong ntiles);
```

In the second part example, we called gimp_pixel_rgn_get_row() and gimp_pixel_rgn_set_row() but without using any cache.

The number of tiles in a tile row will be the layer width divided by the tile width, plus one. So, for a layer width of 65, we will cache two tiles. As we usually also process shadow tiles, we can double that number to compute the ideal cache size for our plug-in.

```
      gimp_tile_cache_ntiles (2 * (drawable->width /
                              gimp_tile_width () + 1));
```

With the cache, our slow plug-in becomes fast. On a 300x300 selection, our last blur took 3 seconds, but on a 2000x1500 selection it was much slower - 142 seconds.

Adding the above line of code, things are getting better: 11 seconds. We still lose transition time when we reach tile borders, we can go down to 10 seconds when multiplying by 4 instead of 2 (meaning we cache two tiles rows), but the more tiles we cache, the more hard disk access we make, which reduce the time gain at a point.

Algorithm generalisation

We can modify the algorithm to take a parameter into account: radius. With a radius of 3, the neighbourhood of a pixel will be 7x7, instead of 3x3 with a radius of 1. To achieve this I modify the previous algorithm:

- allocate space for 2r+1 tile rows
- initialise this rows array, taking care of borders
- for each tile row
  - for each pixel in the tile row
    - compute the neighbourhood average, taking care of borders
  - get a new tile row and cycle rows

This algorithm is more complex than the last one, because the average computing will be a O(r^2) algorithm.

The modified code to get this behaviour is below. Most of the work is done in the process_row function. init_mem and shuffle are there to keep the blur code clean and small.

```
static void blur        (GimpDrawable *drawable);

static void init_mem    (guchar     ***row,
                         guchar      **outrow,
                         gint          num_bytes);
static void process_row (guchar      **row,
                         guchar       *outrow,
                         gint          x1,
                         gint          y1,
                         gint          width,
                         gint          height,
                         gint          channels,
                         gint          i);
static void shuffle     (GimpPixelRgn *rgn_in,
                         guchar      **row,
                         gint          x1,
                         gint          y1,
                         gint          width,
                         gint          height,
                         gint          ypos);

/* The radius is still a constant, we'll change that when the
 * graphical interface will be built. */
static gint radius = 3;
...

static void
blur (GimpDrawable *drawable)
{
  gint         i, ii, channels;
  gint         x1, y1, x2, y2;
  GimpPixelRgn rgn_in, rgn_out;
  guchar     **row;
  guchar      *outrow;
  gint         width, height;

  gimp_progress_init ("My Blur...");

  /* Gets upper left and lower right coordinates,
   * and layers number in the image */
```

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

channels = gimp_drawable_bpp (drawable->drawable_id);

/* Allocate a big enough tile cache */
gimp_tile_cache_ntiles (2 * (drawable->width /
                             gimp_tile_width () + 1));

/* Initialises two PixelRgns, one to read original data,
 * and the other to write output data. That second one will
 * be merged at the end by the call to
 * gimp_drawable_merge_shadow() */
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);

/* Allocate memory for input and output tile rows */
init_mem (&row, &outrow, width * channels);

for (ii = -radius; ii <= radius; ii++)
  {
    gimp_pixel_rgn_get_row (&rgn_in,
                            row[radius + ii],
                            x1, y1 + CLAMP (ii, 0, height - 1),
                            width);
  }

for (i = 0; i < height; i++)
  {
    /* To be done for each tile row */
    process_row (row,
                 outrow,
                 x1, y1,
                 width, height,
                 channels,
                 i);
    gimp_pixel_rgn_set_row (&rgn_out,
                            outrow,
                            x1, i + y1,
                            width);
    /* shift tile rows to insert the new one at the end */
    shuffle (&rgn_in,
             row,
             x1, y1,
             width, height,
             i);
    if (i % 10 == 0)
      gimp_progress_update ((gdouble) i / (gdouble) height);
  }

/* We could also put that in a separate function but it's
```

```
   * rather simple */
  for (ii = 0; ii < 2 * radius + 1; ii++)
    g_free (row[ii]);

  g_free (row);
  g_free (outrow);

  /*  Update the modified region */
  gimp_drawable_flush (drawable);
  gimp_drawable_merge_shadow (drawable->drawable_id, TRUE);
  gimp_drawable_update (drawable->drawable_id,
                        x1, y1,
                        width, height);
}

static void
init_mem (guchar ***row,
          guchar  **outrow,
          gint      num_bytes)
{
  gint i;

  /* Allocate enough memory for row and outrow */
  *row = g_new (char *, (2 * radius + 1));

  for (i = -radius; i <= radius; i++)
    (*row)[i + radius] = g_new (guchar, num_bytes);

  *outrow = g_new (guchar, num_bytes);
}

static void
process_row (guchar **row,
             guchar  *outrow,
             gint     x1,
             gint     y1,
             gint     width,
             gint     height,
             gint     channels,
             gint     i)
{
  gint j;

  for (j = 0; j < width; j++)
    {
      gint k, ii, jj;
      gint left = (j - radius),
           right = (j + radius);

      /* For each layer, compute the average of the
       * (2r+1)x(2r+1) pixels */
      for (k = 0; k < channels; k++)
        {
          gint sum = 0;

          for (ii = 0; ii < 2 * radius + 1; ii++)
            for (jj = left; jj <= right; jj++)
              sum += row[ii][channels * CLAMP (jj, 0, width - 1) + k];

          outrow[channels * j + k] =
            sum / (4 * radius * radius + 4 * radius + 1);
        }
    }
}

static void
shuffle (GimpPixelRgn *rgn_in,
         guchar      **row,
         gint          x1,
         gint          y1,
         gint          width,
         gint          height,
         gint          ypos)
{
  gint    i;
  guchar *tmp_row;

  /* Get tile row (i + radius + 1) into row[0] */
  gimp_pixel_rgn_get_row (rgn_in,
                          row[0],
                          x1, MIN (ypos + radius + y1, y1 + height - 1),
                          width);

  /* Permute row[i] with row[i-1] and row[0] with row[2r] */
  tmp_row = row[0];
  for (i = 1; i < 2 * radius + 1; i++)
    row[i - 1] = row[i];
  row[2 * radius] = tmp_row;
}
```

Adding a graphical interface and saving parameters

To let the user modify the radius, or let a non-interactive script give it as a parameter, we now need to get back to our run() function and settle some simple things.

First we create a structure to allow saving and returning options. Usually one does this even for plug-ins with only one parameter.

```
typedef struct
{
  gint radius;
} MyBlurVals;


/* Set up default values for options */
static MyBlurVals bvals =
{
  3  /* radius */
};
```

Next, we modify the run() function so that execution modes are taken into account. In interactive mode and repeat last filter mode, we try to get the last values used by the gimp_get_data() function, which takes a unique data identifier as its first input parameter. Usually, one uses the procedure's name.

Finally, in interactive mode, we add a few lines that will build the graphical interface allowing options modification.

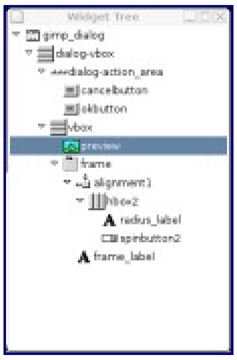
```
static void
run (const gchar      *name,
     gint              nparams,
     const GimpParam  *param,
     gint             *nreturn_vals,
     GimpParam       **return_vals)
{
  static GimpParam  values[1];
  GimpPDBStatusType status = GIMP_PDB_SUCCESS;
  GimpRunMode       run_mode;
  GimpDrawable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

  /* Setting mandatory output values */
  *nreturn_vals = 1;
  *return_vals  = values;

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

  /* Getting run_mode - we won't display a dialog if
   * we are in NONINTERACTIVE mode */
  run_mode = param[0].data.d_int32;

  /*  Get the specified drawable  */
  drawable = gimp_drawable_get (param[2].data.d_drawable);

  switch (run_mode)
    {
    case GIMP_RUN_INTERACTIVE:
      /* Get options last values if needed */
      gimp_get_data ("plug-in-myblur", &bvals);

      /* Display the dialog */
      if (! blur_dialog (drawable))
        return;
      break;

    case GIMP_RUN_NONINTERACTIVE:
      if (nparams != 4)
        status = GIMP_PDB_CALLING_ERROR;
      if (status == GIMP_PDB_SUCCESS)
        bvals.radius = param[3].data.d_int32;
      break;

    case GIMP_RUN_WITH_LAST_VALS:
      /*  Get options last values if needed  */
      gimp_get_data ("plug-in-myblur", &bvals);
      break;

    default:
      break;
    }

  blur (drawable);

  gimp_displays_flush ();
  gimp_drawable_detach (drawable);

  /*  Finally, set options in the core  */
  if (run_mode == GIMP_RUN_INTERACTIVE)
    gimp_set_data ("plug-in-myblur", &bvals, sizeof (MyBlurVals));

  return;
}
```

The graphical interface

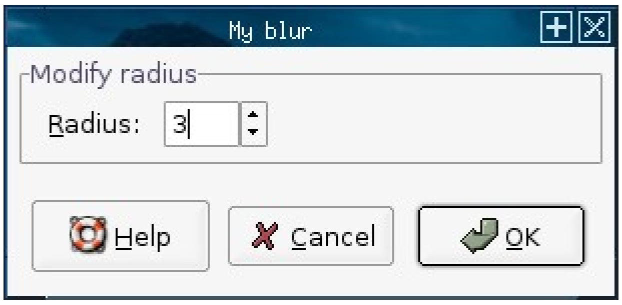
I won't detail GTK+ programming as this is done very well in other places. Our first try will be very simple. We will use the utility widget of GIMP, the GimpDialog, to create a window with a header, a numeric control of type GtkSpinButton (associated with a GtkAdjustment) and its label, nicely framed in a GtkFrame.

In the following parts, in order to show how easy one can do such things, I will add a preview in the dialog to show real time effects of the parameters.

Our final dialog will look like this (tree generated with Glade):



In The GIMP 2.2, there is a number of widgets that come bundled with parameters that allow a coherent behaviour, consistent with GNOME Human Interface Guidelines. GimpPreview also appeared in 2.2. Let's make a first try without it:



```
static gboolean
blur_dialog (GimpDrawable *drawable)
{
  GtkWidget *dialog;
  GtkWidget *main_vbox;
  GtkWidget *main_hbox;
  GtkWidget *frame;
  GtkWidget *radius_label;
  GtkWidget *alignment;
  GtkWidget *spinbutton;
  GtkObject *spinbutton_adj;
  GtkWidget *frame_label;
```

```
        gboolean   run;

        gimp_ui_init ("myblur", FALSE);

        dialog = gimp_dialog_new ("My blur", "myblur",
                                  NULL, 0,
                                  gimp_standard_help_func, "plug-in-myblur",

                                  GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                  GTK_STOCK_OK,     GTK_RESPONSE_OK,

                                  NULL);

        main_vbox = gtk_vbox_new (FALSE, 6);
        gtk_container_add (GTK_CONTAINER (GTK_DIALOG (dialog)->vbox),
main_vbox);
        gtk_widget_show (main_vbox);

        frame = gtk_frame_new (NULL);
        gtk_widget_show (frame);
        gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 6);

        alignment = gtk_alignment_new (0.5, 0.5, 1, 1);
        gtk_widget_show (alignment);
        gtk_container_add (GTK_CONTAINER (frame), alignment);
        gtk_alignment_set_padding (GTK_ALIGNMENT (alignment), 6, 6, 6, 6);

        main_hbox = gtk_hbox_new (FALSE, 0);
        gtk_widget_show (main_hbox);
        gtk_container_add (GTK_CONTAINER (alignment), main_hbox);

        radius_label = gtk_label_new_with_mnemonic ("_Radius:");
        gtk_widget_show (radius_label);
        gtk_box_pack_start (GTK_BOX (main_hbox), radius_label, FALSE, FALSE, 6);
        gtk_label_set_justify (GTK_LABEL (radius_label), GTK_JUSTIFY_RIGHT);

        spinbutton_adj = gtk_adjustment_new (3, 1, 16, 1, 5, 5);
        spinbutton = gtk_spin_button_new (GTK_ADJUSTMENT (spinbutton_adj), 1,
0);
        gtk_widget_show (spinbutton);
        gtk_box_pack_start (GTK_BOX (main_hbox), spinbutton, FALSE, FALSE, 6);
        gtk_spin_button_set_numeric (GTK_SPIN_BUTTON (spinbutton), TRUE);

        frame_label = gtk_label_new ("Modify radius");
        gtk_widget_show (frame_label);
        gtk_frame_set_label_widget (GTK_FRAME (frame), frame_label);
        gtk_label_set_use_markup (GTK_LABEL (frame_label), TRUE);

        g_signal_connect (spinbutton_adj, "value_changed",
                          G_CALLBACK (gimp_int_adjustment_update),
                          &bvals.radius);
        gtk_widget_show (dialog);

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

        gtk_widget_destroy (dialog);

        return run;
      }
```

Adding a GimpPreview

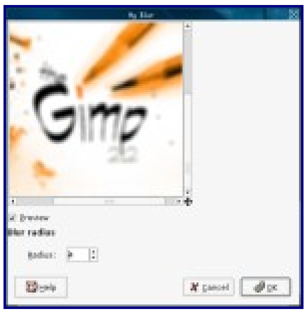
Adding a GimpPreview is quite easy. First we create a GtkWidget with gimp_drawable_preview_new(), then we attach an invalidated signal to it, which will call the blur function to update the preview. We also add a second parameter to MyBlurVals to remember the activation state of the preview.

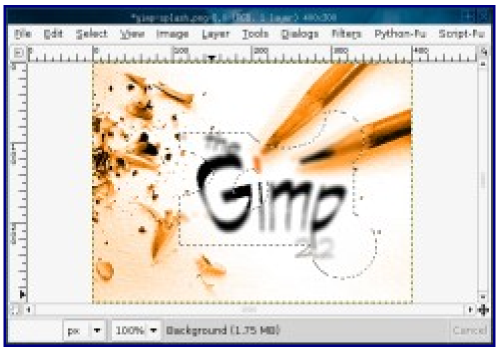
A method to update easily the preview is to add a preview parameter in the blur function, and if preview is not NULL, to take GimpPreview limits. So when we call blur from run(), we set the preview parameter to NULL.

To take GimpPreview limits, we use gimp_preview_get_position() and gimp_preview_get_size(), so we can generate only what will be displayed.

To achieve this the right way we'll tune some of the code - we don't need to update the progress bar while generating the preview, and we should tell at GimpPixelRgn init time that the tiles should not be sent back to the core.

Finally, we display the updated preview with the gimp_drawable_preview_draw_region() function. We get a dialog box that shows us in real time the plug-in effects. Moreover, thanks to the GIMP core, our plug-in already takes selections into account.





Here are the two functions in their last version:

```
static void
blur (GimpDrawable *drawable,
      GimpPreview  *preview)
{
  gint         i, ii, channels;
  gint         x1, y1, x2, y2;
  GimpPixelRgn rgn_in, rgn_out;
  guchar     **row;
  guchar      *outrow;
  gint         width, height;

  if (!preview)
    gimp_progress_init ("My Blur...");

  /* Gets upper left and lower right coordinates,
   * and layers number in the image */
  if (preview)
  {
    gimp_preview_get_position (preview, &x1, &y1);
    gimp_preview_get_size (preview, &width, &height);
    x2 = x1 + width;
    y2 = y1 + height;
  }
  else
  {
    gimp_drawable_mask_bounds (drawable->drawable_id,
                               &x1, &y1,
                               &x2, &y2);
    width = x2 - x1;
    height = y2 - y1;
  }

  channels = gimp_drawable_bpp (drawable->drawable_id);

  /* Allocate a big enough tile cache */
  gimp_tile_cache_ntiles (2 * (drawable->width /
                               gimp_tile_width () + 1));

  /* Initialises two PixelRgns, one to read original data,
   * and the other to write output data. That second one will
   * be merged at the end by the call to
   * gimp_drawable_merge_shadow() */
  gimp_pixel_rgn_init (&rgn_in,
                       drawable,
                       x1, y1,
                       width, height,
                       FALSE, FALSE);
  gimp_pixel_rgn_init (&rgn_out,
                       drawable,
                       x1, y1,
                       width, height,
                       preview == NULL, TRUE);

  /* Allocate memory for input and output tile rows */
  init_mem (&row, &outrow, width * channels);

  for (ii = -bvals.radius; ii <= bvals.radius; ii++)
    {
      gimp_pixel_rgn_get_row (&rgn_in,
                              row[bvals.radius + ii],
                              x1, y1 + CLAMP (ii, 0, height - 1),
                              width);
    }

  for (i = 0; i < height; i++)
    {
      /* To be done for each tile row */
      process_row (row,
                   outrow,
                   x1, y1,
                   width, height,
                   channels,
                   i);
      gimp_pixel_rgn_set_row (&rgn_out,
                            outrow,
                            x1, i + y1,
                            width);
      /* shift tile rows to insert the new one at the end */
      shuffle (&rgn_in,
               row,
               x1, y1,
               width, height,
               i);
      if (i % 10 == 0 && !preview)
        gimp_progress_update ((gdouble) i / (gdouble) height);
    }

  for (ii = 0; ii < 2 * bvals.radius + 1; ii++)
    g_free (row[ii]);

  g_free (row);
  g_free (outrow);

  /*  Update the modified region  */
  if (preview)
    {
      gimp_drawable_preview_draw_region (GIMP_DRAWABLE_PREVIEW (preview),
                                         &rgn_out);
    }
  else
    {
      gimp_drawable_flush (drawable);
      gimp_drawable_merge_shadow (drawable->drawable_id, TRUE);
      gimp_drawable_update (drawable->drawable_id,
                            x1, y1,
                            width, height);
    }
}

static gboolean
blur_dialog (GimpDrawable *drawable)
{
  GtkWidget *dialog;
  GtkWidget *main_vbox;
  GtkWidget *main_hbox;
  GtkWidget *preview;
  GtkWidget *frame;
  GtkWidget *radius_label;
  GtkWidget *alignment;
  GtkWidget *spinbutton;
  GtkObject *spinbutton_adj;
  GtkWidget *frame_label;
  gboolean   run;

  gimp_ui_init ("myblur", FALSE);

  dialog = gimp_dialog_new ("My blur", "myblur",
                            NULL, 0,
                            gimp_standard_help_func, "plug-in-myblur",

                            GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                            GTK_STOCK_OK,     GTK_RESPONSE_OK,

                            NULL);

  main_vbox = gtk_vbox_new (FALSE, 6);
```

```
        gtk_container_add (GTK_CONTAINER (GTK_DIALOG (dialog)->vbox),
main_vbox);
        gtk_widget_show (main_vbox);

        preview = gimp_drawable_preview_new (drawable, &bvals.preview);
        gtk_box_pack_start (GTK_BOX (main_vbox), preview, TRUE, TRUE, 0);
        gtk_widget_show (preview);

        frame = gimp_frame_new ("Blur radius");
        gtk_box_pack_start (GTK_BOX (main_vbox), frame, FALSE, FALSE, 0);
        gtk_widget_show (frame);

        alignment = gtk_alignment_new (0.5, 0.5, 1, 1);
        gtk_widget_show (alignment);
        gtk_container_add (GTK_CONTAINER (frame), alignment);
        gtk_alignment_set_padding (GTK_ALIGNMENT (alignment), 6, 6, 6, 6);

        main_hbox = gtk_hbox_new (FALSE, 12);
        gtk_container_set_border_width (GTK_CONTAINER (main_hbox), 12);
        gtk_widget_show (main_hbox);
        gtk_container_add (GTK_CONTAINER (alignment), main_hbox);

        radius_label = gtk_label_new_with_mnemonic ("_Radius:");
        gtk_widget_show (radius_label);
        gtk_box_pack_start (GTK_BOX (main_hbox), radius_label, FALSE, FALSE, 6);
        gtk_label_set_justify (GTK_LABEL (radius_label), GTK_JUSTIFY_RIGHT);

        spinbutton = gimp_spin_button_new (&spinbutton_adj, bvals.radius,
                                           1, 32, 1, 1, 1, 5, 0);
        gtk_box_pack_start (GTK_BOX (main_hbox), spinbutton, FALSE, FALSE, 0);
        gtk_widget_show (spinbutton);

        g_signal_connect_swapped (preview, "invalidated",
                                  G_CALLBACK (blur),
                                  drawable);
        g_signal_connect_swapped (spinbutton_adj, "value_changed",
                                  G_CALLBACK (gimp_preview_invalidate),
                                  preview);

        blur (drawable, GIMP_PREVIEW (preview));

        g_signal_connect (spinbutton_adj, "value_changed",
                          G_CALLBACK (gimp_int_adjustment_update),
                          &bvals.radius);
        gtk_widget_show (dialog);

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

        gtk_widget_destroy (dialog);

        return run;
      }
```

Conclusion

In these articles, we saw basic concepts for several aspects of a GIMP plug-in. We messed with image data treatment through a simple algorithm, and followed a path that showed us how to avoid performance problems. Finally, we generalised the algorithm and added parameters to it, and we used some GIMP widgets to make a nice user interface.