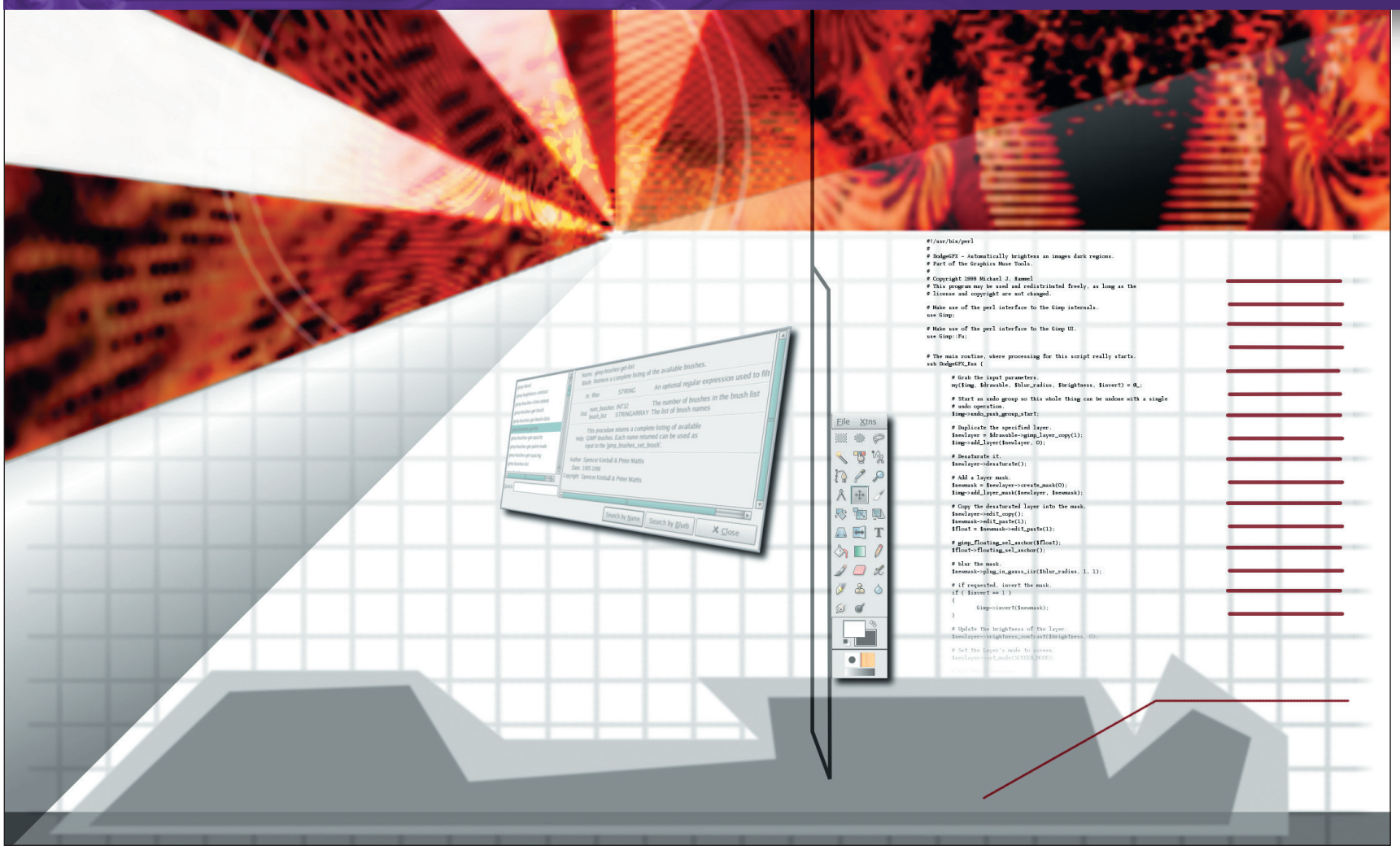


GIMP PROGRAMMING

Writing GIMP Plug-ins in Perl

PART 1 The magic of *The GIMP* lies not just in what Wilbur can show you, but in what Wilbur can be taught.

For the past year, the tutorials in this *GIMP* series have presented readers with creative ideas and solutions for use with *The GIMP*. Most tutorials followed a step-by-step format with accompanying images for clarification. This month, we're starting to take a more traditional approach to learning, to cover a more technical topic: *GIMP scripting with Perl*. This tutorial is part one of two on that subject.

Scripting is a means to automate common and/or repetitive tasks with *The GIMP*. Examples might be the generation of certain styles of logos, or processing directories of images in a uniform manner. *The GIMP* has a native scripting language called Script-FU, but this language has all the friendliness of a rabid dog. Script-FU is based on the Scheme language (related to the Lisp language, if you care about such things) and the lack of available documentation for that language – not to mention its technical orientation – makes it more suited to the tech-only crowd.

Perl, on the other hand, is a language that is widely understood. There is a wealth of printed material on the subject. Unlike Scheme, Perl is not a truly difficult language to learn. It provides both procedural and object-oriented APIs and doesn't abuse curly braces the way Script-FU does. That means that once you learn the *GIMP* specifics, you need only pick up a few extras of the language to start your own set of power tools. Scripts written in Perl for use with *The GIMP* are referred to as GIMP Perl scripts. GIMP Perl is also used to refer to the extensions to Perl that allow these scripts to work with GIMP.

This tutorial will be of most interest to those that do a lot of image processing, as opposed to those who just use *The GIMP* to create a holiday card or two. But even if you only use *The GIMP* to manage your scanned photos, scripting in *The GIMP* can make your graphics work far more productive. You need not be a Perl expert to follow along, but you should have at least a little background in programming: such as knowing what functions are, or how to do assignment and loop statements. Specifics to the Perl language can be found at the **http://perl.com** website.

Installation is an apt-snap

The Perl extension to *The GIMP* is not included with the application by default, and getting all the prerequisite software to build it manually can be difficult. Fortunately, this issue is moot if

you use apt-get to update your *GIMP* packages. A quick search of some common RPM-based *apt-get* repositories shows the following packages:

- **gimp** – The GNU Image Manipulation Program.
- **gimp-data-extras** – Extra files for *The GIMP*.
- **gimp-devel** – *The GIMP* plug-in and extension development kit.
- **gimp-perl** – Perl extensions and plug-ins for *The GIMP*.

The names on the left are the names of the available *GIMP* packages. This list was generated using:

```
apt-cache search gimp
```

There are other *GIMP*-related packages, but for scripting with *The GIMP* this is all you need. To find out which packages you already have, simply use **rpm**:

```
rpm -qa | grep gimp
```

The results will look something like this:

```
gimp-1.2.3-16
gimp-devel-1.2.3-16
gimp-data-extras-1.2.0-8
```

In this case only the *gimp-perl* package is missing. To install the missing packages, you need to have *apt-get* installed. If you were able to run *apt-cache,* then you already have *apt-get*. If not, check out **www.freshrpms.net** to find a version that will work with your system. Once *apt-get* is installed, you can retrieve the missing packages:

```
apt-get install gimp-perl
```

Because *apt* uses RPM to install packages in system directories, this command will need to be run as the root user.

Meat and garnish, hold the taters.

The basic structure of a GIMP Perl script looks like this:

```
#!/usr/bin/perl

use Gimp;
use Gimp::Fu;

register();

sub my_script {
}

exit main;
```

The script is composed of two bits of meat and some garnish. The meat of the script consists of a call to the registration function and a subroutine that does the real work. Before we look at those, we need to look at the garnish – the first three lines of the script.

```
#!/usr/bin/perl
use Gimp;
use Gimp::Fu;
```

The first two lines are required for all GIMP Perl scripts. The third is used by nearly all GIMP Perl scripts except for those that require complex user interfaces. Line 1 tells the system to run this as a Perl script. Line 2 tells Perl to use a Perl module called **Gimp** that is the glue between scripts and *GIMP* functions. The third line is used to automate creation of the user interface. We'll look more in depth at what happens in the third line in next month's concluding episode of this two-part series.

When a GIMP-Perl script starts, the first thing it must do is let *The GIMP* know it's available and where it will be found in the menus. The script does this by calling a function provided by the GIMP Perl module (ie the module referenced by **use Gimp** at the top of the script) called **register()**.

Here's an example piece of code using the **register()** function with GIMP Perl:

```
register (
  "gfxoffset",
  "Find all layer offsets",
  "Find all layer offsets",
  "Michael J. Hammel",
  "Copyright 2004 Michael J. Hammel",
  "V1.0",
  "<Image>/Filters/GFXMuse/GFXOffsets",
  "*",
  [
    [ PF_STRING, "filename",
      "Where should the output be saved?",
      "/tmp/offsets.txt"
    ]
  ],
  \&OffsetGFX_Run
);
```

The call to **register()** includes many arguments. The first is the name of the script. GIMP Perl doesn't like it if you use uppercase for this name, so use the same name as the script but in all lowercase. This name is used internally by GIMP Perl and is not displayed to the user except possibly in the window title bar (depending on your window manager). The second argument is a short description while the third is a longer one. The only difference here is that the short description must be no more than one line long. The next two arguments are used to specify the author of the script and any copyright information. After that comes the version number.

Now we get to the truly important parts of the call to **register()**. The seventh argument is a menu path. The first part of this path is used to define where in the *GIMP* menus to place your script. Normally you'll put your scripts under the Image menu, usually in the Filters submenu though that is not required. To put the script in the Image menu, you prefix your menu path with **<Image>**. The only other option is **<Toolbox>**, which puts your script under one of the menus in the Toolbox. In the preceding example, the script is destined to be found in the Image menu under Filters>GFXMuse>GFXOffsets. 'GFXMuse' is a new submenu under Filters, which shows you that you can create »

GIMP 2.0
Perl vs Python

GIMP 2.0 does not include GIMP Perl by default. Python has been selected as the default language (beyond Script-FU). Users of *GIMP1.2* may wonder if learning GIMP Perl for that release is of any value now that *GIMP 2.0* is here and doesn't appear to actively support Perl. The answer is a resounding "Yes!" – it is of value.

GIMP Perl for *GIMP 2.0* will be released as its own add-on package at some point after *GIMP 2.0* is released. Don't be alarmed. This change in packaging is not unique to GIMP Perl. The *GAP* plug-ins (*GIMP Animation Plug-ins*) are also being moved to their own package separate from *The GIMP*. This has happened in the past as well, when *GIMP-Print* moved out on its own, not to mention the eldest brethren of *The GIMP*'s offspring, the *GTK+* toolkit.

GIMP Python will not have nearly as much public support initially, since many script writers grew up on GIMP Perl. Additionally, it will take some time before *GIMP 2.0* is actively included in distributions. Perl is also not terribly different than Python in its syntax (the look of the language), which means learning one helps in learning the other. Finally, many example scripts already exist for GIMP Perl while few exist for Python. For these reasons, learning GIMP Perl now definitely has long-term value.

your own submenu this way, perhaps grouping all your personal scripts in a single menu.

The eighth argument determines the types of images the script can work on. *The GIMP* understands RGB, Greyscale and Indexed images, either with or without transparency. An asterisk can be used as a wildcard. So **RBG*** would mean both RGB and RGB with transparency (aka RGBA) would be supported. In this example, all image types are supported. You can also use **GRAY**, and **INDEXED** (with a trailing **A for** transparency – the **A** means "includes an Alpha channel").

The next argument is a set of Perl arrays that define the layout of the window used to configure the script prior to running it. This section can get complex so for now we'll use this simple example that only provides a text entry field to query the user for where the output from this script (which will be ordinary text, not image data) should be saved. In next month's tutorial, we'll dive into this part of GIMP Perl a bit deeper.

Finally, the last argument to **register()** tells GIMP Perl the name of the function we wrote that does the actual work. After the user fills in the window presented by the script and clicks the 'OK' button, GIMP Perl will call the function named here (which is in our script) to do whatever work we require.

On to the code

GIMP Perl is so easy to learn that we can start by showing the complete code for a real-world example script now, leaving the details for later. At I company for whom I periodically do some work, a recent requirement arose to produce a set of images from a single photograph. The images produced would be overlaid onto a background, one image at a time, by a web-based backend graphics processor. To make this work easier, the photograph was first manually broken into its component pieces as individual layers, using a common naming scheme for the layers. Then, a script was written to output the offsets of the layers so that the layers, once saved to separate image files, could be overlaid properly by the backend.

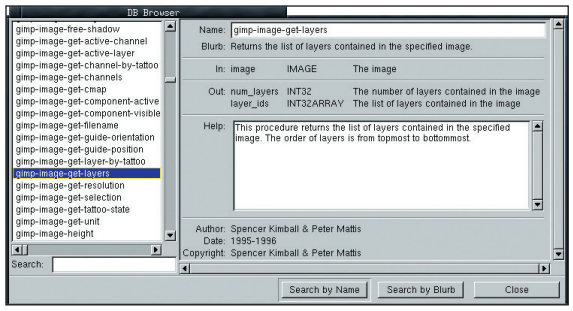
There were actually two scripts for this project, one for saving each layer to a separate file, and one for creating a single file with all the layer offsets. The first we called GFXLayerSave.pl and the latter GFXOffsets.pl. Note that the work of creating the layers was manual – if the bits of the photograph to convert to layers were uniform in location and shape (which they were not), this too could have been automated. Both of these script files are available on our website at **www.linuxformat.co.uk/gimp/54.zip**

We start by looking at the smaller of the two scripts, GFXOffsets.pl:

```
#!/usr/bin/perl
use Gimp;
use Gimp::Fu;

# Our function - get the list of layers and print their offsets to a file.
sub OffsetGFX_Run {

  my($img, $drawable, $filename) = @_;

  my @layers = $img->get_layers();

  my $count = scalar(@layers);
  Gimp->progress_init("GFXLayerSave is working...");
  my $progress_increment = 1 / $count;
  my $progress = 0.0;

  open (FD, ">$filename") || die "GFXOffsets: Can't open $filename\n";
  print FD " X    Y     Layername\n";
  print FD "---- ----  ----------------------------------------\n";
  foreach (@layers)
  {
    my ($xoffset, $yoffset) = $_->drawable_offsets();
    my $layername = $_->layer_get_name();
    printf(FD "%4d %4d   %s\n", $xoffset, $yoffset, $layername);

    # Update the progress bar.
    $progress += $progress_increment;
    Gimp->progress_update ($progress);

  }
  close(FD);
  return();
}


# Register this script with the Gimp's PDB.
register (
  "gfxoffset",
  "Find all layer offsets",
  "Find all layer offsets",
  "Michael J. Hammel",
  "Copyright 2004 Michael J. Hammel",
  "V1.0",
  "<Image>/Filters/GFXMuse/GFXOffsets",
  "*",
  [
    [ PF_STRING, "filename",
      "Where should the output be saved?",
      "/tmp/offsets.txt"
    ]
  ],
  \&OffsetGFX_Run
);

exit main();
```

The user interface for GFXOffsets – simplicity is a built-in feature with GIMP Perl.

Here we see the **register()** function follows a function called **OffsetGFX_Run()**. Functions in Perl are prefixed by the keyword **sub**, which means 'subroutine', that happens to also be a synonym for 'function'. Note also that the order of the two



The DB Browser allows you to search the procedural database quickly and easily.

functions doesn't matter – GIMP Perl will find either function and call them when needed. The meat of our code is in our subroutine, **OffsetGFX_Run()**. Let's put in plain English what we're doing in our code:

1 Retrieve important values passed to us by GIMP Perl
2 Retrieve the set of layers in our image
3 Run through that set to
a retrieve the pixel offset of the current layer
b print the relevant data to a file

In addition, we also update the display to let the user know that something is happening. For images with a small number of layers, this isn't very important. For images with a very large number layers – as was the case for our selection of images – then this helps let the user know something is really happening and, just as important, when its done.

GIMP Perl passes in three values to our script: A value used to identify the image being worked on (*ie* the Canvas), a value used to identify a 'drawable' (which would be the currently active layer), and the name of the file as provided by the user via the window created using argument 9 of the **register()** function.

```
my($img, $drawable, $filename) = @_;
```

The **$img** value is important to us because we'll use it to find all the layers in the image on which we're working. The **$drawable** value isn't used here because we aren't interested in just the current active layer, we'll be iterating over all the layers in the image. But many (if not most) other GIMP Perl scripts work primarily on the active drawable and use this value extensively. The name of the file to save to, as provided by the user, is passed in as the **$filename** argument.

The subroutine specified in the **register()** call will be passed the values supplied by the user in the order they are listed in the **register()** function in argument 9. Again, we'll cover developing user interfaces in part two of this tutorial next issue.

Now that we have these values, we can start to do real work.

```
my @layers = $img->get_layers();
```

The next thing to be done is get that list of layers. The Perl language can be used either with a procedural (like the C language) or object oriented (like C++) interface. These two methods can even be mixed. Here we use the object methodology to ask the **$img** object to run the **get_layers()** method and save the layer ids to an array called **@layers**. How did we know about **get_layers()**? We used *The GIMP*'s Procedural Database.

The DB Browser (found in the Toolbox menus as Xtns>DB Browser) lists all functions that can be called by a GIMP Perl script. We'll cover this next month as well, but until then, you should know that anything that is prefixed with **gimp-image** in the procedural database can be accessed using the object-oriented syntax, minus that prefix. Alternatively, you can call **gimp_image_get_layers()** – the dashes in the database name are replaced with underscores. This is the procedural interface. Arguments provided differ between the procedural and object interface. The object interface doesn't need image or drawable ids depending on the class the function belongs to.

```
my $count = scalar(@layers);
Gimp->progress_init("GFXLayerSave is working...");
my $progress_increment = 1 / $count;
my $progress = 0.0;
```

These lines are all used to provide feedback to the user. The first line assigns the count of layers to the variable **$count**. The next line prints a message at the bottom of the image window. The next two lines are used to initialise a progress bar – a horizontal scrollbar that fills in as we do our work. When the scrollbar runs end to end, we're done.

```
open (FD, ">$filename") || die "GFXOffsets: Can't open $filename\n";
print FD " X    Y     Layername\n";
print FD "---- ----  ----------------------------------------
----\n";
```

The next three lines open the output file and print a header to it. If the file can't be opened for some reason, the script will exit (*ie* **die**) and print a message.

```
my ($xoffset, $yoffset) = $_->drawable_offsets();
my $layername = $_->layer_get_name();
printf(FD "%4d %4d  %s\n", $xoffset, $yoffset, $layername);
```

If the subroutine is the crispy, griddled meat of this scripting steak, then these next three lines must be the moist and bloody red meat at the centre. The first line grabs the offsets for our layer. The **foreach()** loop inside which we find these lines sets the **$_** variable to each layers object. So the call to **$_->drawable_offsets()** is how we get a single layers offsets.

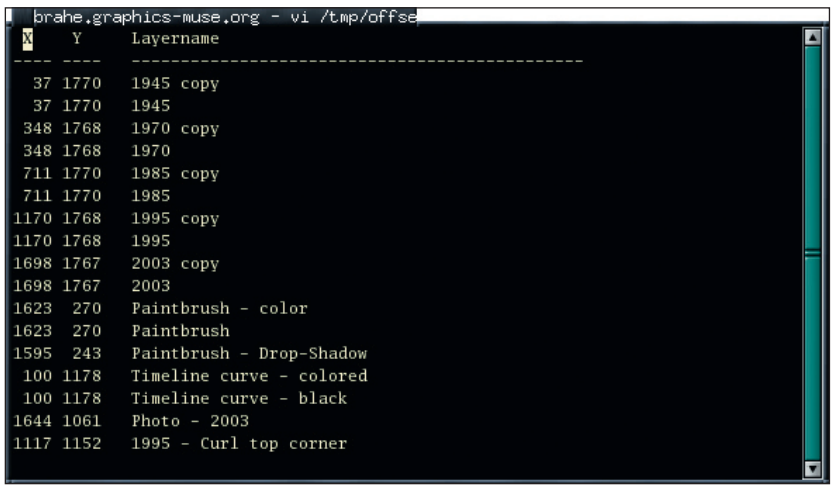
The next line grabs the layer name and that, along with the offsets, is then written to our output file.

```
$progress += $progress_increment;
Gimp->progress_update ($progress);
```

These lines print visible status changes to the progress bar, incrementing it an equal amount for each layer we process.

Where to go from here

The output from this script is very basic. It could be parsed manually or fed to a parser that feeds the graphics processor that builds dynamic images for the web. LXF



The output is simple, but formatted for easy parsing.

NEXT MONTH

IN the concluding part of our examination of The GIMP and Perl, we'll look at the other script, GFXLayerSave.pl, which is a little more complex (but barely) and dig deeper into creating user interfaces and using the DB Browser to find the functions we need for our scripts.