Perl Gtk3 Programming

Table of Contents

1. Introduction

This tutorial was written jointly by my son and I. The aim was to learn some perl Gtk3 programming and maybe pass on some of what we learnt.

Firstly I need to give credit to all the great sources of information that we came across while putting this together:

> The Gtk2 Perl Study Guide
> Dirk van der Walt.
> http://gtk2-perl.sourceforge.net/doc/gtk2-perl-study-guide/
> Our main inspiration.

> Introduction to GUI Programming with Gtk2-Perl
> Ross McFarland
> http://gtk2-perl.sourceforge.net/doc/intro/

> GTK+ 3 Reference Manual
> http://developer.gnome.org/gtk3/stable/
> Although intended for C, the is very useful. Most of the bindings to Perl are deliberately logical and it soon becomes easy to interpret.

> Gtk2-Perl: An Introduction: Or How I Learned to Stop Worrying and Make Things Pretty
> Scott Paul Robertson
> http://media.scottr.org/presentations/gtk2-perl.pdf

2. Getting ready

We need to find out which version of perl you are actually running. The version below comes from my installation of Debian Wheezy (testing) on an AMD64 architecture.

```
# perl -v
This is perl 5, version 14, subversion 2 (v5.14.2) built for x86_64-linux-gnu-thread-multi
(with 61 registered patches, see perl -V for more detail)
```

We need to check we have the gtk3 perl library installed.

```
aptitude search libgtk3-perl
i   libgtk3-perl                          - Perl bindings for the GTK+ graphical user interface
library
```

If it is not installed then install with

```
# aptitude install libgtk3-perl
```

You will need to be root to install a new package.

3. The First Gtk3 program

We will make a simple window that we can maximise, minimise and drag around the screen.

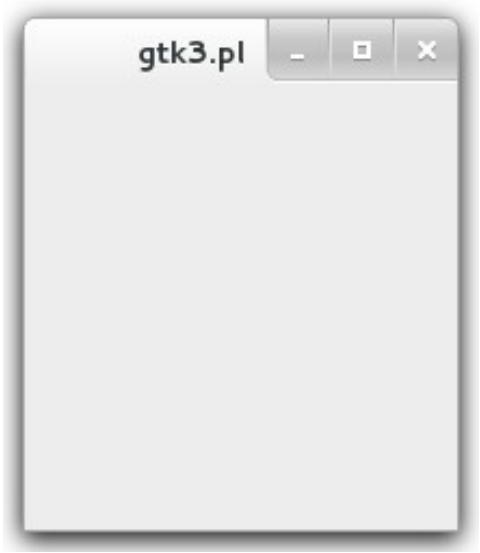

Figure 1: A basic window

```
#! /usr/bin/perl

use strict;
use warnings;
use diagnostics;
use feature ':5.14';
use Gtk3 '-init';

my $window = Gtk3::Window->new('toplevel');
$window->show_all;
```

```
Gtk3->main;
```

One you have typed the above into a file, safe as 1-A-basic-window.pl , set the execute permission and run.

```
$ chmod 700 1-A-basic-window.pl
$ ./1-A-basic-window.pl
^C
$
```

Lets have a look at what we have, how it works and then see what it does.

The first line tells the system where the perl compiler is. If you get an error saying:

```
bad interpreter: No such file or directory
```

then check you are pointing to the right place and change the line to point to where your Perl interpretor actually is.

```
# which perl
/usr/bin/perl
```

```
use strict;
use warnings;
```

The next two lines are highly recommended in all your Perl programs and prevent some bad Perl practices as well as issuing warnings for others.

```
use diagnostics;
```

This is optional, but if you have an error it will give you a more detailed report on the problem. It might help you find

```
use feature ':5.14';
```

This turns on some of the more recent language features such as using

say “Hello”;

in place of

print “Hello\n”;

```
use Gtk3 '-init';
```

This is where we initialise the Gtk3 toolkit. Every script that uses Gtk3 will need this line. A few caveats on using this line:

Gtk3 needs to be initialised before its used. You can declare Gtk3 in the use statement and initialise it later but it is safer to do the two together.

If your program relies several modules then Gtk3 needs to be initialised in the main program, you shouldn't initialise it in every module.

```
my $window = Gtk3::Window->new('toplevel');
```

In this line we initialise a new variable ($window) and we set it up as a new Gtk3 window. We have a choice of window types, in this case we choose a 'toplevel' window, other types include 'popup' for information and quick dialogue boxes.

```
$window->show_all;
```

We need to tell Gtk3 that we want to show the window on the screen. This statement shows the window and all its children. At the moment we have no children so it just shows the window.

```
Gtk3->main;
```

Finally this causes Gtk3 to wait for user input. But as we have nothing set up to catch that input the window just sits there doing nothing and with little functionality. The window can be dragged around the screen, the maximise and minimise buttons work and if you click the close button the window disappears. However the program as not quitted properly and you still need to press Ctrl-C from the terminal to kill the program and get the command prompt back.

When an event occurs in a Gtk3 application that event generates a signal. With the program as it stands most of those signals are just lost. To add functionality we tell Gtk3 to look out for certain signals and pass them to a function we set up.

The close button on the top of the window generates a “delete_event“ signal. We need to look out for that signal and when its triggered we need to shut the program down properly by telling Gtk3 to quit.

To fix the problem we need to add one line.

```
#! /usr/bin/perl

use strict;
use warnings;
use diagnostics;

use Gtk3 '-init';

my $window = Gtk3::Window->new('toplevel');

$window->signal_connect (delete_event => sub { Gtk3->main_quit });

$window->show_all;
Gtk3->main;
```

Now when you click the close button on the window the program shouts down properly.

We could have set up a separate function and ran that when the event was fired. However in this case the function was very short and we could use an anonymous (as opposed to a named) subroutine embedded into the statement.

4. A better Window

In this section we will add our own title, centre the window in the screen and set the default size to something a little more sensible.

```
#! /usr/bin/perl

use strict;
use warnings;
use diagnostics;

use Gtk3 '-init';

my $window = Gtk3::Window->new('toplevel');
$window->set_title("My Title");
$window->set_position("center");
$window->set_default_size(400, 300);
$window->set_border_width(20);

$window->signal_connect (delete_event => sub { Gtk3->main_quit });

$window->show_all;
Gtk3->main;
```

$window->set_title("My Title");

The first new line simply replaces the text in the title bar with some text we supply.

$window->set_position("center");

we cause set_position to change where the window will be placed. In this case we have moved it to the centre of the screen. Other options include:

none	Don't influence where the window is created.
center	Create the window in the centre of the screen.
mouse	Create the window under the current mouse pointer.

$window->set_default_size(400, 300);

This sets the default width and height of the window and is measured in pixels.

$window->set_border_width(20)

Adds a thicker border to the window.

5. Adding buttons

In Gtk3 terms a child is an element that is created inside another – the parent. Hence your Gtk3 program will look like a tree structure with your window object containing its children who may then contain their children.

A Gtk3 window has a major restriction, it can only have one child. So you can't add multiple buttons or any other widgets directly to a window. To handle this we first add a container widget and this will hold our buttons and any other widgets, maybe even different containers..

A container widget is a box for other widgets to fit into. Its purpose is to organise the layout of your window. You tell it how you want your widgets laid out. In the basic box container you tell the container whether to stack the widgets on top of each other (vertical) or line them up next to each other (horizontal).

In this section we will had two buttons to the window and connect both to a quit function.

Figure 2: A better window

```
#! /usr/bin/perl

use strict;
use warnings;
use diagnostics;
use feature ':5.14';
use Gtk3 '-init';

use Glib qw/TRUE FALSE/;

sub quit_function {
        say "Exiting Gtk3";
        Gtk3->main_quit;
        return FALSE;
}

my $window = Gtk3::Window->new('toplevel');
$window->set_title("My Title");
$window->set_position("mouse");
$window->set_default_size(400, 50);
$window->set_border_width(20);
$window->signal_connect (delete_event => \&quit_function);

my $button1 = Gtk3::Button->new("Quit");
$button1->signal_connect (clicked => \&quit_function);
```

```
my $button2 = Gtk3::Button->new("Another Quit");
$button2->signal_connect (clicked => \&quit_function);

my $hbox = Gtk3::Box->new("horizontal", 5);
$hbox->pack_start($button1, TRUE, TRUE, 0);
$hbox->pack_start($button2, TRUE, TRUE, 0);

$hbox->set_homogeneous (TRUE);

$window->add($hbox);

$window->show_all;
Gtk3->main;
```

In this program we have added a few needs lines.

```
use Glib qw/TRUE FALSE/;
```

Perl has no predefined values for true and false, it uses 1 and 0. To make your programs more readable it is good practice to import the predefined values already defined in the Glib library. This line imports the Glib values for TRUE and FALSE so they can be used.

```
sub quit_function {
        say "Exiting Gtk3";
        Gtk3->main_quit;
        return FALSE;
}
```

The next new item is a small function. It will take over the place of the anonymous function in the previous program that we used to close the program down. The function will print a short closing message to the terminal, quit Gtk3 and destroy the window. Any button we want to use to close the program will now point to this function.

```
$window->signal_connect (delete_event => \&quit_function);
```

The next new line is where we connect the delete event to our new quit_function. Note that the signal connect function requires a reference to a function and not the function name itself. To make a reference in Perl you prefix the function name with \&.

```
my $button1 = Gtk3::Button->new("Quit");
$button1->signal_connect (clicked => \&quit_function);

my $button2 = Gtk3::Button->new("Another Quit");
$button2->signal_connect (clicked => \&quit_function);
```

In this section of code we set up two buttons and connect the clicked event for each of them to our new quit_function.

```
my $hbox = Gtk3::Box->new("horizontal", 5);
$hbox->pack_start($button1, TRUE, TRUE, 0);
$hbox->pack_start($button2, TRUE, TRUE, 0);
```

```
$hbox->set_homogeneous (TRUE);

$window->add($hbox);
```

Finally we define a new box (our container) and pass it two parameters. The first tells Gtk3 that we want to stack items in the box horizontally and the second parameter is the spacing between the widgets in pixels.

Now we have a box we start adding things to it. To do this we use the pack_start function which packs widgets into the container starting at the left hand edge and adding more to the right.

The pack_start function has several extra parameters which are:

$hbox->pack_start($button2, expand, fill, padding);

Expand : TRUE – expand the container to make space for the button.

Fill : TRUE – the extra space created by the expand option above is allocated to the widgets and not just filled with padding.

Padding : An extra number of pixels to add between widgets to create some space.

```
$hbox->set_homogeneous (TRUE);
```

The set_homogeneous function tells the box to allocate the same amount of space to each widget, so we get two buttons the same width;

```
$window->add($hbox);
```

Finally we had our new container to the main window.

6. Adding button functions

Now we are going to get our buttons to do something more interesting. We will get them to change the text on a label.

Figure 3: Adding button functions

```
#! /usr/bin/perl

use strict;
```

```
use warnings;
use diagnostics;
use feature ':5.14';
use Gtk3 '-init';

use Glib qw/TRUE FALSE/;

#### DECLARATIONS

my $window = Gtk3::Window->new('toplevel');
$window->set_title("My Title");
$window->set_position("mouse");
$window->set_default_size(400, 50);
$window->set_border_width(20);
$window->signal_connect (delete_event => \&quit_function);

my $button1 = Gtk3::Button->new("Quit");
$button1->signal_connect (clicked => \&quit_function);

my $button2 = Gtk3::Button->new("Say Hello");
$button2->signal_connect (clicked => \&say_something, "Hello");

my $button3 = Gtk3::Button->new("Say Goodbye");
$button3->signal_connect (clicked => \&say_something, "Goodbye");

my $hbox = Gtk3::Box->new("horizontal", 5);
$hbox->set_homogeneous (TRUE);
$hbox->pack_start($button1, TRUE, TRUE, 0);
$hbox->pack_start($button2, TRUE, TRUE, 0);
$hbox->pack_start($button3, TRUE, TRUE, 0);

my $vbox = Gtk3::Box->new("vertical", 5);
$vbox->add($hbox);

my $label = Gtk3::Label->new("Am I connected?");
$vbox->add($label);

$window->add($vbox);
$window->show_all;

#### FUNCTIONS

sub quit_function {
        say "Exiting Gtk3";
        Gtk3->main_quit;
        return FALSE;
}

sub say_something {
   my ($button, $userdata) = @_;
   $label->set_label( $userdata );
```

```
    return TRUE;
}

Gtk3->main;
```

```
my $button2 = Gtk3::Button->new("Say Hello");
$button2->signal_connect (clicked => \&say_something, "Hello");

my $button3 = Gtk3::Button->new("Say Goodbye");
$button3->signal_connect (clicked => \&say_something, "Goodbye");
```

The button functions have changed slightly. We now declare two buttons, both point to the same function but they also contain a second parameter which will be passed to the function when its activated. So if you press button2 you will activate the say_something function and pass it the string “Hello”. If you press button3 you will activate the same function but this time you will pass it the string “Goodbye”.

```
my $vbox = Gtk3::Box->new("vertical", 5);
$vbox->add($hbox);

my $label = Gtk3::Label->new("Am I connected?");
$vbox->add($label);

$window->add($vbox);
```

The buttons are packed into a horizontal box as before but that box is not added directly to the window. Instead we create another box, a vertical box, and we add the horizontal button box to this, followed by a new label with the text “Am I connected”. The effect is to have a row of button with a label underneath them.

Finally we add the vbox (which has the hbox and label) to the window. Our object hierarchy now looks like.

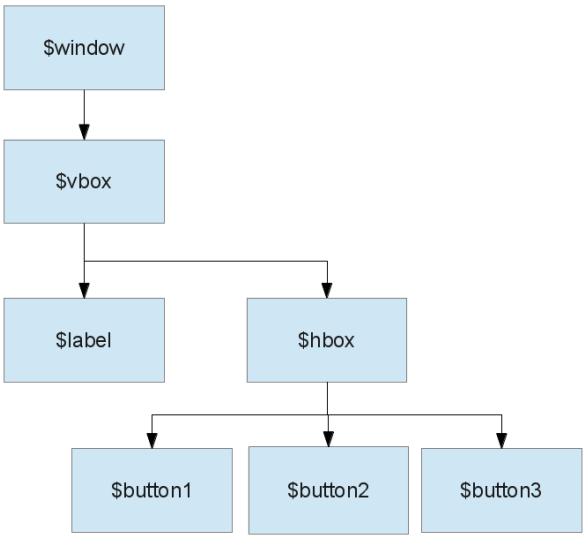

Figure 4: Object hierachy

7. Fun with labels

A demo of how you can style labels using Pango markup and other effects.

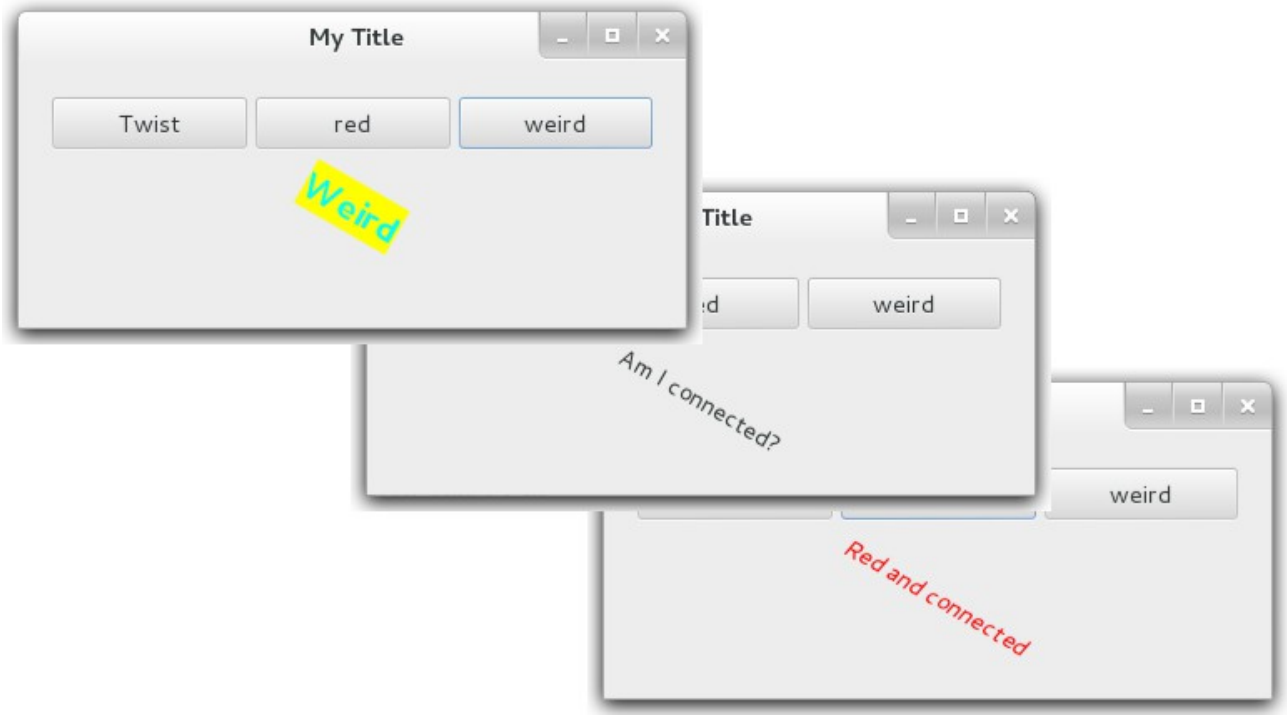

Figure 5: Fun with labels

```
#! /usr/bin/perl

use strict;
use warnings;
use diagnostics;
use feature ':5.14';
use Gtk3 '-init';

use Glib qw/TRUE FALSE/;

#### DECLARATIONS

my $window = Gtk3::Window->new('toplevel');
$window->set_title("My Title");
$window->set_position("mouse");
$window->set_default_size(400, 50);
$window->set_border_width(20);
$window->signal_connect (delete_event => \&quit_function);

my $button1 = Gtk3::Button->new("Twist");
$button1->signal_connect (clicked => \&say_something, "twist");

my $button2 = Gtk3::Button->new("red");
$button2->signal_connect (clicked => \&say_something, "red");
```

```
my $button3 = Gtk3::Button->new("weird");
$button3->signal_connect (clicked => \&say_something, "weird");

my $hbox = Gtk3::Box->new("horizontal", 5);
$hbox->set_homogeneous (TRUE);
$hbox->pack_start($button1, TRUE, TRUE, 0);
$hbox->pack_start($button2, TRUE, TRUE, 0);
$hbox->pack_start($button3, TRUE, TRUE, 0);

my $vbox = Gtk3::Box->new("vertical", 5);
$vbox->add($hbox);

my $label = Gtk3::Label->new("Am I connected?");
$vbox->add($label);

$window->add($vbox);
$window->show_all;

#### FUNCTIONS

sub quit_function {
        say "Exiting Gtk3";
        Gtk3->main_quit;
        return FALSE;
}

sub say_something {
   my ($button, $userdata) = @_;
   if ($userdata eq "twist") {
      my $angle = $label->get_angle;
      $label->set_angle( $angle - 30 );
   } elsif ($userdata eq "red") {
      $label->set_markup('<span foreground="red"><i>Red and connected</i></span>');
   } elsif ($userdata eq "weird") {
      $label->set_markup(
'<span foreground="cyan" background="yellow" size="x-large"><b>Weird</b></span>');
   }
   return TRUE;
}

Gtk3->main;
```

```
my $button1 = Gtk3::Button->new("Twist");
$button1->signal_connect (clicked => \&say_something, "twist");

my $button2 = Gtk3::Button->new("red");
$button2->signal_connect (clicked => \&say_something, "red");
```

```
my $button3 = Gtk3::Button->new("weird");
$button3->signal_connect (clicked => \&say_something, "weird");
```

We declare three buttons, give them different names and connect each ones clicked event to the same function (say_something). Each button passes a different parameter telling the function what it wants to happen to the label.

```
sub say_something {
   my ($button, $userdata) = @_;
   if ($userdata eq "twist") {
      my $angle = $label->get_angle;
      $label->set_angle( $angle - 30 );
   } elsif ($userdata eq "red") {
      $label->set_markup('<span foreground="red"><i>Red and connected</i></span>');
   } elsif ($userdata eq "weird") {
      $label->set_markup(
'<span foreground="cyan" background="yellow" size="x-large"><b>Weird</b></span>');
   }
   return TRUE;
}
```

The rest of the program is the same as before until we get to the say_something function. This time we check the userdata parameter passed and trigger a different label function depending on what we receive.

If the userdata is “twist” we read the label angle, take away 30 degrees and reset the label to the new angle. Each time we press the button the label moves by another 30 degrees.

If the userdata is “red” we change the text using the set_markup function which allows us to add html type styling. If the userdata is weird we change the styling to cyan with a yellow background.