Introduction

Perl’s GD::Graph module is a tool that allows a software developer to quickly and easily generate graphical representations of data. Originally written by Martien Verbruggen in 1995, the package has matured into a very flexible and popular tool. It is ideally suited to any situation where a dynamic data set, from a database or elsewhere, needs to be fetched and represented in an on-the-fly manner. It is widely used in corporate Intranets, where many a webmaster has used it to generate graphs that show data in exactly the format management likes.

This article explores the capabilities of GD::Graph, and presents some Perl code that a developer can use to get started with the module.

Installation

GD::Graph sits on top of a set of low level utility modules. At the bottom of the pile is Thomas Boutell’s ‘GD’ graphics primitives library which provides a structure for drawing assorted lines, polygons and text onto an in-memory canvas. On top of this sit two Perl modules: Lincoln Stein’s GD module which allows Perl developers to access the underlying GD facilities, and Martien Verbruggen’s GD::Text module which provides a Perl interface for drawing text onto a GD based image. The GD::Graph module sits on top of the pile, using features from all the underlying modules to draw images containing graphs.

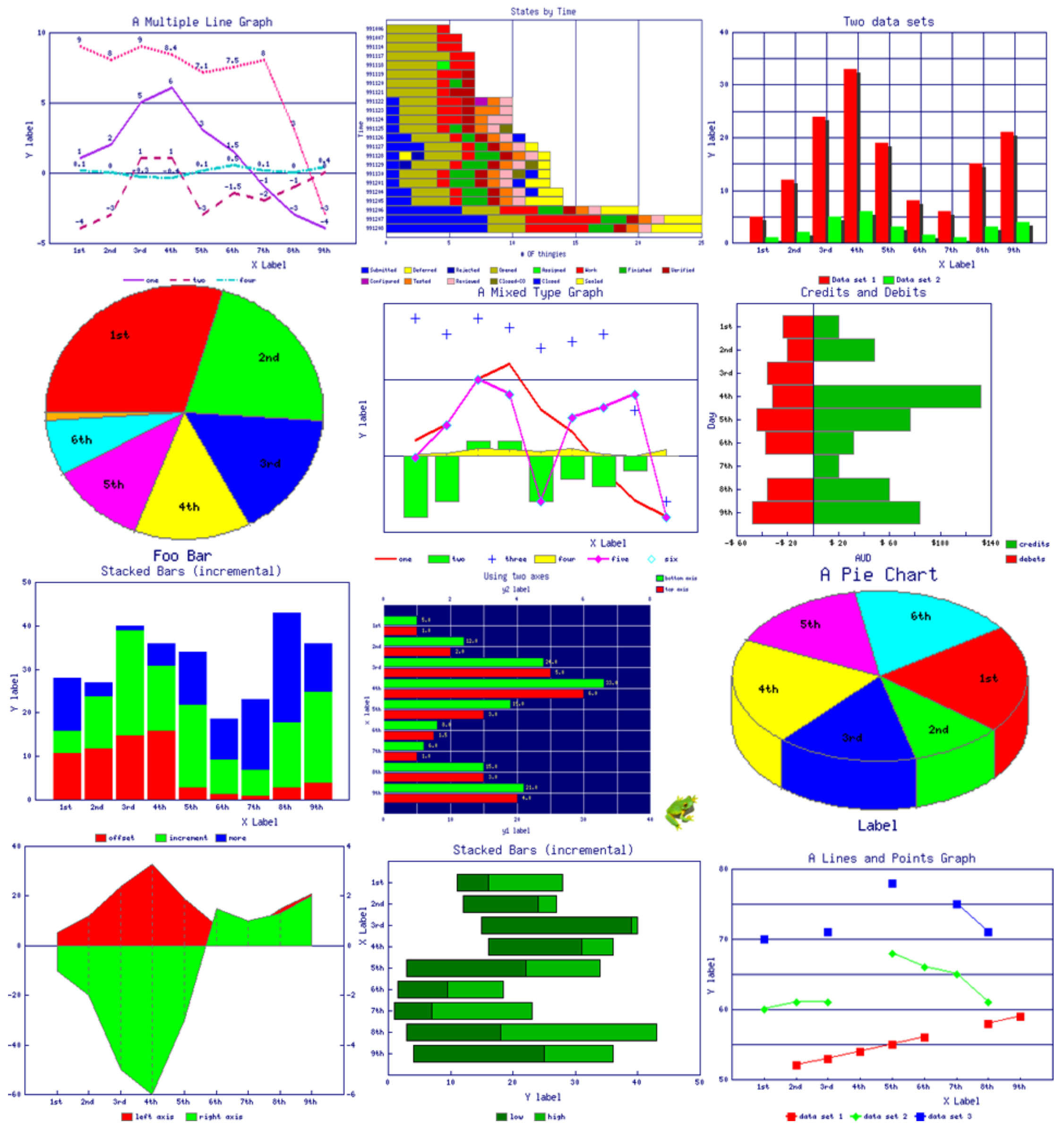
The GD library is included in all modern Linux distributions, and the source is available for other platforms. The Perl specific parts of GD::Graph are either included with your Linux distribution, or are all available from CPAN. The most up to date versions can all be retrieved with the single command (as root):

```
perl –MCPAN –e ‘install GD::Graph’
```

You’ll need the Perl CPAN module, the GD development package and a C compiler for this installation command to complete successfully. These are standard fit on most modern development workstations.

GD::Graph Capabilities

The GD::Graph module takes as its input a set of data, and produces as its output an image showing a graphical representation of that data. The value of the module comes from its extraordinary flexibility. It can produce graphs of many different types: line charts, point charts, bar charts, area graphs and pie charts, and can mix these to show several data sets in one image. It provides complete control over the graph’s axes, tick marks, colours, keys, icons, logos and so on. The underlying GD module provides plenty more lower level routines which can be used to add extra text to an image, frame it, add alpha blending and so on. Figure 1 shows a selection of graph styles produced by the sample scripts that come with GD::Graph.



Simple usage

Typical usage of GD::Graph involves three steps. Firstly, the data has to be retrieved from the database or wherever it's coming from, and packed into a Perl data structure. Next, a GD::Graph object is created from this data structure using a set of additional parameters which instruct the module exactly how the graph should be drawn. Finally, the graph image that is built and stored in memory, needs to be delivered. That would normally mean saving it to a file, pushing it down a network connection or somehow sending it to an application that can use it.

A graph always starts with a multidimensional array containing the data sets. In Perl, multidimensional arrays are stored as an array containing a number of array references. GD::Graph data carries the labels for the X-axis in the first array reference, and the actual data sets in the subsequent ones. The simplest Perl code to set up a GD::Graph data array with two data sets looks like this:

```
my @xLabels  = qw( Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec );
my @data2002 = qw(  17  19  26  38  56  64  67  53  40  29  21  13 );
my @data2003 = qw(  19  24  27  41  56  69  75  60  44  33  22  15 );
my @data     = ( \@xLabels, \@data2002, \@data2003 );
```

The arrays should all be the same size. If a value is unknown it can be set to `undef`, and GD::Graph will not attempt to plot it.
Naming the arrays is rarely necessary, so it is common to see the anonymous array composer used to create the data:

```
my @data = ( ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"],
             [ 17, 19, 26, 38, 56, 64, 67, 53, 40, 29, 21, 13],
             [ 19, 24, 27, 41, 56, 69, 75, 60, 44, 33, 22, 15] );
```

Once the data is prepared, the graph object can be created with a call to the appropriate constructor. All the graph constructors take two parameters: the width and height of the required graph image. So, for example, creating a new bar chart object, 800 pixels wide by 600 pixels high, looks like this:

```
my $graph = GD::Graph::bars->new( 800, 600 );
```

At this stage the new object just contains the storage for the image. The data and the image creation itself come later. The next step is to tune the graph object so it appears as required. This step uses the graph object's `set()` method. There are dozens of options available here, and many will be discussed later in this article. For now, here's how to set a title and a label for the graph's Y-axis:

```
$graph->set( title   => "Rainfall 2002/2003",
             y_label => "Millimetres" );
```

Now it's time to actually draw the graph. This is easy – just a single method call:

```
my $image = $graph->plot( \@data )
```

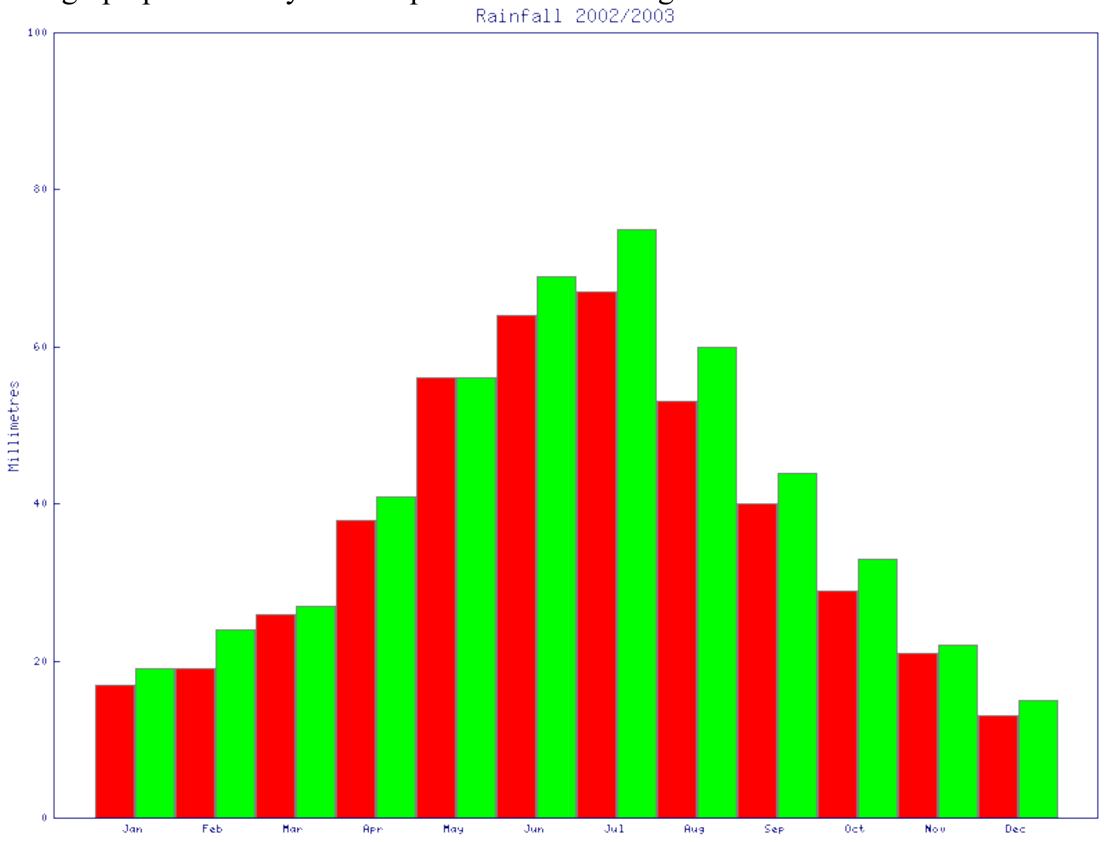
Note that the data array is passed as a reference. The return value from this call is a low level image object from the GD library, or an undefined value if the plotting of the graph failed for some reason.
The final step is to convert the GD image to a useable format, then do something with it. If you have the ImageMagick package on your machine, you can convert to PNG format and display it like this:

```
my $pngData = $image->png();
open( OUT, "| display -" ) or die( "Can't display image: $!" );
binmode OUT;
print OUT $pngData;
close OUT;
```

Forcing the output to binary mode using `binmode` isn't necessary on Unix or Linux, but is required on some other systems.
Putting all this together gives this script shown in Listing 1, to generate and display a very simple, two data set bar chart:

```
#!/usr/bin/perl -w
use strict;

use GD::Graph::bars;

my @xLabels  = qw( Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec );
my @data2002 = qw(  17  19  26  38  56  64  67  53  40  29  21  13 );
my @data2003 = qw(  19  24  27  41  56  69  75  60  44  33  22  15 );
my @data     = ( \@xLabels, \@data2002, \@data2003 );

my $graph = GD::Graph::bars->new( 800, 600 );

$graph->set( title   => "Rainfall 2002/2003",
             y_label => "Millimetres" );

my $image = $graph->plot( \@data ) or die( "Cannot create image" );

open( OUT, "| display -") or die( "Cannot display image: $!" );
binmode OUT;
print OUT $image->png();
close OUT;
```

The graph produced by this script can be seen in Figure 2.



Advanced usage

Advanced usage of the GD::Graph module basically means choosing the graph style you want, then tweaking the options to tune the output graph to exactly what is required.
Choosing the graph style simply involves using the appropriate module and calling its constructor. In the example above the bars module was used for a bar chart. The alternatives are: `lines`, `points`, `linespoints`, `area`, `mixed` or `pie`.
Choosing graph options is nowhere near as simple. There are many options, some of which can be applied to all graph types, and some of which are type specific. There are far too many options in GD::Graph to try to cover in one article, so I'll just pick out a few that indicate some of the possibilities the module presents. The generic options that apply to most graph types are set using the graph object's `set()` method.

As we shall see, there are also additional methods for setting more specialised options.
Let's start with colours. If you want to control colours in your graph, by far the easiest way is to use the GD::Graph::colour module which provides access to system colours. If you add these lines to the top of your script:

```
use GD::Graph::colour qw( :files );
GD::Graph::colour::read_rgb( "/usr/X11/lib/X11/rgb.txt" );
```

you can use the colour names in the system table (at least on a Linux system). Note the British spelling of "colour" in the module name. Given this module, we can make our rainfall bar chart look a little less garish by adding these options to the call to the `set()` method:

```
boxclr       => "LightGrey"
dclrs        => [ "DeepSkyBlue", "SteelBlue" ]
shadowclr    => "DarkSlateGrey"
shadow_depth => 3
```

You might have noticed that Listing 1 doesn't give any indication of the numbers to be used on the Y-axis of this bar chart. This is because the GD::Graph module is smart enough to work out a range that fits the data being used. This intelligence can be overridden with the `y_max_value` and `y_min_value` options. For this graph, a maximum of 80 on the Y-axis works a little better than the default of 100:

```
y_max_value => 80
```

There are lots of other options for the `set()` method which make bar charts like this one look better depending on their contents and nature. For example, try setting:

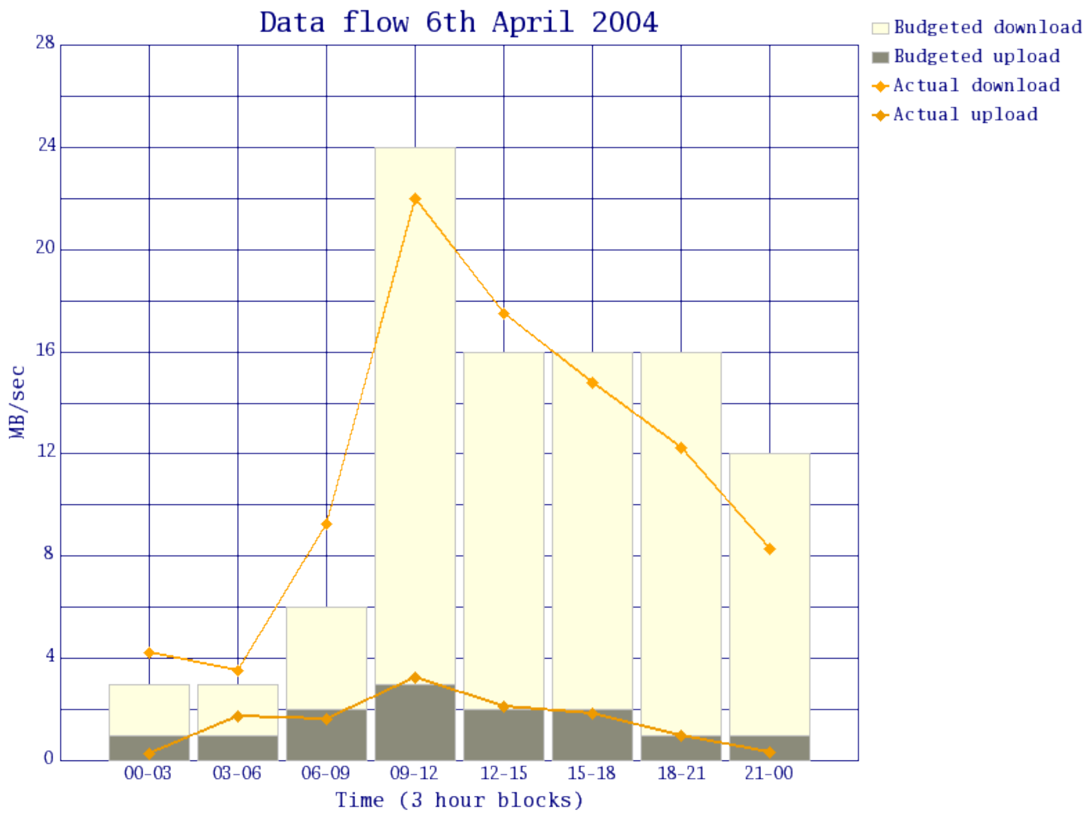
```
cumulate => 1
```

or:

```
overwrite => 1
```

You'll need to play with the test data somewhat to see the effect of that second one.
Let's move on and look at a few more options on a different graph. Listing 2 shows a Perl script that produces a graph of type `mixed`. The output image from this script is shown in Figure 3.

```
#!/usr/bin/perl -w
use strict;

use GD::Graph::mixed;
use GD::Graph::colour qw( :files );
use GD::Text;

my @xLabels  = qw( 00-03 03-06 06-09 09-12 12-15 15-18 18-21 21-00 );
my @avDown   = qw(     3     3     6    24    16    16    16    12 );
my @avUp     = qw(     1     1     2     3     2     2     1     1 );
my @down     = qw(   4.2   3.5  9.25    22  17.5  14.8  12.25  8.3 );
my @up       = qw(   0.25 1.75  1.65   3.25  2.1  1.85   0.95  0.3 );
my @data     = ( \@xLabels, \@avDown, \@avUp, \@down, \@up );
my $graph = GD::Graph::mixed->new( 800, 600 );

GD::Graph::colour::read_rgb( "/usr/X11/lib/X11/rgb.txt" ) or
  die( "Can't read colours" );

$graph->set( title             => "Data flow 6th April 2004",
           t_margin         => 10,
           b_margin         => 10,
           l_margin         => 10,
           r_margin         => 10,
           x_label          => "Time (3 hour blocks)",
           x_label_position => 0.5,
           y_label          => "MB/sec",
           types            => [ qw(bars bars
                              linespoints linespoints) ],
           dclrs            => [ qw(LightYellow1 LightYellow4
                              orange1 orange2) ],
           y_max_value      => 28,
           y_tick_number    => 14,
           y_label_skip     => 2,
           line_width       => 2,
           long_ticks       => 1,
           bar_width        => 10,
           bar_spacing      => 3,
           markers          => [ 5, 5 ],
           legend_placement => "RT" );

$graph->set_legend( "Budgeted download",
                "Budgeted upload",
                "Actual download",
                "Actual upload" );

GD::Text->font_path( "/usr/lib/X11/fonts/truetype/" );
$graph->set_title_font( "luximr", 16 );
$graph->set_legend_font( "luximr", 10 );
$graph->set_x_axis_font( "luximr", 9 );
$graph->set_x_label_font( "luximr", 11 );
$graph->set_y_axis_font( "luximr", 9 );
$graph->set_y_label_font( "luximr", 11 );

my $image = $graph->plot( \@data ) or die( "Cannot create image" );

open( OUT, "| display -") or die( "Cannot display image: $!" );
binmode OUT;
print OUT $image->png();
close OUT;
```



A mixed graph allows you to show several data sets in different ways in the same image. The `types` option tells GD::Graph how to present the data sets. In this case I want the first 2 sets as bars, indicating a given day's budgeted network bandwidth, and the second 2 sets as lines, showing the actual bandwidth used.
With this graph I've set the line points to filled diamonds (with the `markers` option) and I've spaced the bars out a little (with the `bar_width` and `bar_spacing` options). The tick lines have been drawn right across the graph (with the `long_ticks` option), and I've set the scale and steps on the Y-axis (with the `y_tick_number` and `y_label_skip` options) to make the information more readable. This graph also has a legend in the top right corner. Setting legend details is done with a call to the graph's `set_legend()` method.
I've also specified the font handling just as I want it. GD::Graph uses the GD::Text module to do its text handling. GD::Text doesn't attempt to use the native windowing system's font searching, so I pointed it at the truetype fonts directory on my SuSE system, then specified an exact font and size for each of the text labels. When distributing a script that uses specific fonts, it is wise to either include the font file with the script, or always stick to fonts found on all of your target systems.

CGI usage

A very common usage for GD::Graph is the dynamic generation of graphs to be viewed in a web browser. The trusty old combination of Perl and CGI remains one of the simplest ways to implement on the fly graph generation for most intranet webmasters. A simple CGI script to generate a graph in JPEG format looks like Listing 3:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use GD::Graph::pie;

my $cgi = CGI->new();
print $cgi->header( -type => "image/jpeg" );

my @data = ( ["Apache","IIS","Other"], [ 67.2, 21.0, 11.8] );

my $graph = new GD::Graph::pie( 250, 200 );

$graph->set( title       => "Web server Usage, March 2004",
             dclrs       => [ qw( #D6D6FF #CECECE #FFFFFF ) ],
             pie_height  => 32,
             start_angle => 90 );

print $graph->plot(\@data)->jpeg();
```

This script delivers a pie chart to the requesting browser using the standard CGI module. The difference in image format type (JPEG instead of PNG) comes from simply calling a different method of the plotted data image object.

Conclusion

The GD::Graph module is a remarkable tool capable of generating all sorts of graphs and charts. It is fast, flexible, and built entirely from free software. A justifiable favourite amongst webmasters, it is also extremely useful to administrators, developers, and anyone who needs to produce graphical representations of data.