Mail-merged envelopes with Perl and PostScript

Post Letter Perfect

You do not need either an Office package or LaTeX to print the envelopes for a mail-shot. Perl's PostScript modules, an address database and the script from this article are the perfect solution for creating envelopes for mass mailing jobs.

BY MICHAEL SCHILLI

Prajuab Manklang, visipix.com

Linux can do more or less anything. Read digital images from cameras, play digital tracks, write CDs – you can even get USB scanners to work, with a little help from *XSane*.

But I had to fire up Windows occasionally to handle one task, and that was printing my address and the recipients' addresses from a database for about 20 mail-merged letters a month. I used to use an ancient Windows program for that, but not any more! Ghostscript can quickly turn your plain old household printer into a mean PostScript machine.

If your distribution has not already taken care of setting this up, go to [1] for a how-to. As Figure 1 shows, I had no trouble exporting the address database I had been using on Windows to a comma-separated format (CSV). All I had to do now, was generate a PostScript file for each envelope, and then send those files to my printer. And that is child's play with CPAN modules such as *PostScript::File* and *PostScript::TextBlock*, as [2] will tell you. PostScript is basically just another programming language. PostScript files are made up of readable ASCII text and contain a list of the commands needed to generate a printed page.

Painting by Numbers

However, PostScript uses a so-called mathematical coordinate system, and this is slightly unusual for a layout program. The origin of this system is the bottom left corner of the page. The x-axis extends to the right, and the y-axis upward from this point, just like your old math teacher told you. PostScript uses pica points, or PostScript points, which are 1/72 of an inch. The following commands would print the name *John Doe* in the address box of an envelope:

```
0 setgray 401.95 156 moveto
/Helvetica-iso findfont
18 scalefont setfont
(John Doe) show
```

Starting at the bottom left of the envelope, this would move almost 402 points (or 5.6 inches) to the right and 156 points (about 2.2 inches) up, to print the letters in the brackets in the specified font (Helvetica-iso), and the specified size, 18 points, from left to right on the paper.

CPAN provides the *PostScript::File* and *PostScript::TextBlock* modules to simplify this double Dutch. The former inserts the PostScript header, as in

```
%!PS-Adobe-3.0
```

and takes care of things like the page orientation, the borders, and the page order. *PostScript::TextBlock* accepts multi-line strings and writes from the given co-ordinates. However, you can look forward to many hours of tinkering with these modules to produce the desired layout at the right place on the page.

Envelopes typically use the following layout: the text block with the sender's address will start about 4/5 of an inch down and to the right of the top left corner. The text block for the addressee can extend to within 4/5 of an inch of the bottom right corner on both the x and y

Figure 1: The fields in the comma-separated address file

THE AUTHOR

Michael Schilli works as a Web engineer for AOL/Netscape in Mountain View, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.

direction. In other words, we do not want to specify the start of the address field; instead we will simply define the position of the bottom right corner of the text block. This ensures that the block will always end neatly in the same position, no matter what variables we use, and no matter how long the address data may be.

The sample script in Listing 1 defines a constant *$SENDER* (line 14). It reads the address data from a *.csv* file and outputs an envelope as shown in Figures 2 or 3 on your printer for each address it finds.

Importing from Windows

$ADDR_CSV in line 13 specifies the name of the address file, which should have the same structure as the example shown in Figure 2. The command to send a PostScript file to your printer is defined in *$PRINT_CMD* in line 17. If you want to perform a trial run, without wasting reams of paper, just replace *"lpr"* with *"ghostview"* to send your virtual envelopes to the screen.

The code in line 19 opens the address file, and the *while* block starting in line 22 iterates against the entries, which are parsed using regular expressions. Instead of this technique, we could have used the CPAN *Text::CSV_XS* module. As the address entries in our sample file are extremely simple, and do not use complicated things like quotation marks or embedded commas, that might be slightly over the top.

Manual Labor vs. Module Power

Line 23 interprets any lines that start with a hash sign # (and whitespace) as comments. This is handy, if you just want to select a few entries for printing. It is quite simple to comment out all the other entries by pre-pending a hash sign, #. The *split* command in line 24 splits lines where separating commas appear. *map* then removes the double quotes. As the substitution that follows *s/"//g;* does not return the result string, *$_;* is simply appended.

Line 27 creates the *PostScript::File* object that uses the *landscape* keyword to rotate the page format. *reencode = > 'ISOLatin1Encoding'* provides support for all Latin1 characters. The envelope format is set to *Envelope-DL*. If you need a different format, because you use a different size of envelope, you can easily modify the script. A DIN A 6 envelope measures about 4 by 6 inches (10.47 by 14.81 cm), so the following definition should do the trick:

```
my $ps = new PostScript::File(
landscape   => 1,
reencode => 'ISOLatin1Encoding',
width       => cm(10.47),
height      => cm(14.81),
  );
```

Line 36 stores the address fields in the variables *$last*, *$first*, *$city* and *$str*. Line 39 calls the *textbox()* function, which is defined further down, and expects a multiple line string, a font name, a font size, a the line spacing in PostScript points. I decided to use *Helvetica-iso*, as *Helvetica* is installed by default. The *-iso* suffix also supports accented characters. *textbox()* returns three values: a *PostScript::TextBlock* object and the width and height of the generated text block in PostScript points.

Following this, line 41 calls the *Write()* method of the *PostScript::TextBlock* object to create PostScript code. *Write()* expects four parameters: width and height of the text block, and the x and y-offsets from the origin. The width and height are provided by the *textbox()* function called prior to this.

Offset Thinking

The x-offset (the distance from the left margin) is 0.8 inches, which we can express as a centimeter value, *cm(2)*, as the *cm()* function defined lower down converts centimeters to PostScript points. The y-offset is more complex, as *Write()* expects a distance from the bottom margin, whereas we need to stipulate 2 centimeters from the top margin. But not to worry: the *$ps->get_width()* method of the *PostScript::File* objects provides the height of the envelope, and we can simply subtract *cm(2)* from this in line 42.

Note that *PostScript::File* retains the original notions of width and height, despite using *landscape* mode, where the page is rotated through 90 degrees. In our case, *get_width()* returns the height, and *get_height()* the width. *Write* returns a list, where the first element is the PostScript code of the text block. Line 43 adds this code to the current PostScript page.

The same approach is used for the addressee: line 46 concatenates the first and family names, street and city to create a multiple line string. The *textbox()* function uses a slightly larger font and line spacing this time. The x-offset from the top left corner of the textbox to the PostScript origin is provided by the length of the envelope (*$ps->get_height()*) minus the width of the textbox (*$bw*) minus 2 centimeters (*cm(2)*) for a border. The y-offset, that is the distance from the top edge of the textbox to the bottom edge of the envelope is provided by adding 2 centimeters to the height of the textbox (*$bh + cm(2)*).

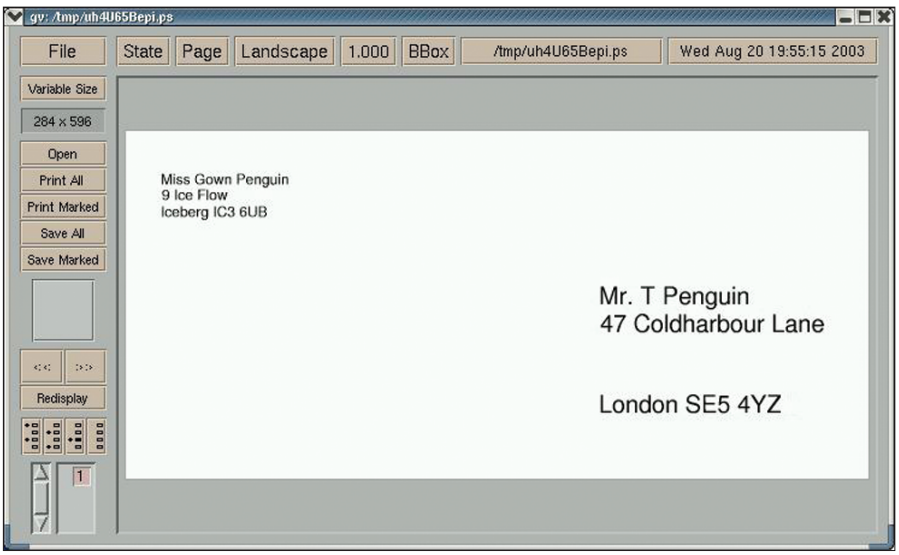

Figure 2: No matter whether the sender has a short...

Listing 1: *envelope*

```
001 #!/usr/bin/perl
002
##################################
003 # envelope - Print paper
    envelopes
004 # Mike Schilli, 2003
    (m@perlmeister.com)
005
##################################
006 use warnings;
007 use strict;
008
009 use PostScript::File;
010 use PostScript::TextBlock;
011 use File::Temp qw(tempfile);
012
013 my $ADDR_CSV  =
    "mailaddr.csv";
014 my $SENDER    = q{Steven
    Sender,
015 9 Sender Street,
016 San Francisco, CA 94107};
017 my $PRINT_CMD = "lpr";
018
019 open FILE, $ADDR_CSV or
020     die "Cannot open
    $ADDR_CSV";
021
022 while(<FILE>) {
023   next if /^\s*#/;
024   my @addr = split /,/, $_;
025   @addr = map { s/"//g; $_; }
    @addr;
026
027   my $ps = PostScript::File
    ->new(
028     landscape   => 1,
029     reencode    =>
    'ISOLatin1Encoding',
030     paper       => "Envelope-
DL",
031   );
032
033   my ($tmp_fh, $tmp_file) =
034
    tempfile(SUFFIX => ".ps");
035
036   my($last, $first, $city,
    $str) = @addr;
037
038     # Sender
039   my($bw, $bh, $b) =
    textbox($SENDER,
040                   "Helvetica-
    iso", 10, 12);
041   my ($code) = $b->Write($bw,
    $bh, cm(2),
042                   $ps-
    >get_width() - cm(2));
043   $ps->add_to_page($code);
044
045     # Recipient
046   my $to = "$first
    $last\n$str\n\n$city\n";
047   ($bw, $bh, $b) =
    textbox($to,
048                   "Helvetica-
    iso", 18, 20);
049   ($code) = $b->Write($bw,
    $bh,
050             $ps->get_height()
    - $bw - cm(2),
051             $bh + cm(2));
052   $ps->add_to_page($code);
053
054     # Print to temporary file
055   (my $base = $tmp_file) =~
    s/\.ps$//;
056   $ps->output($base);
057
058     # Send to printer
059   system("$PRINT_CMD
    $tmp_file") and
060       die "$PRINT_CMD
    $tmp_file: $!";
061
062     # Delete
063   unlink "$tmp_file" or
064       die "Cannot unlink
    $tmp_file: $!";
065 }
066
067
##################################
068 sub textbox {
069
##################################
070     my($text, $font, $size,
    $leading) = @_;
071
072     my $b =
    PostScript::TextBlock->new();
073
074     $b->addText(
075         font    => $font,
076         text    => $text,
077         size    => $size,
078         leading => $leading);
079
080     return(tb_width($text,
    $font, $size),
081            tb_height($text,
    $leading),
082            $b);
083 }
084
085
##################################
086 sub cm {
087
##################################
088     return int($_[0]*72/
    2.54);
089 }
090
091
##################################
092 sub tb_width {
093
##################################
094     my($text, $font, $size)
    = @_;
095
096     $font =~ s/-iso//;
097
098     my $max_width = 0;
099
100     for(split /\n/, $text) {
101         s/[äÄöÖüÜß]/A/ig;
102         my $w =
103         PostScript::Metrics::
    stringwidth(
104                           $_,
    $font, $size);
105         $max_width = $w if $w
    > $max_width;
106     }
107
108     return $max_width;
109 }
110
111
##################################
112 sub tb_height {
113
##################################
114     my($text, $leading) = @_;
115
116     my $lines = 1;
117     $lines++ for $text =~
    /\n/g;
118
119     return $lines*$leading;
120 }
```

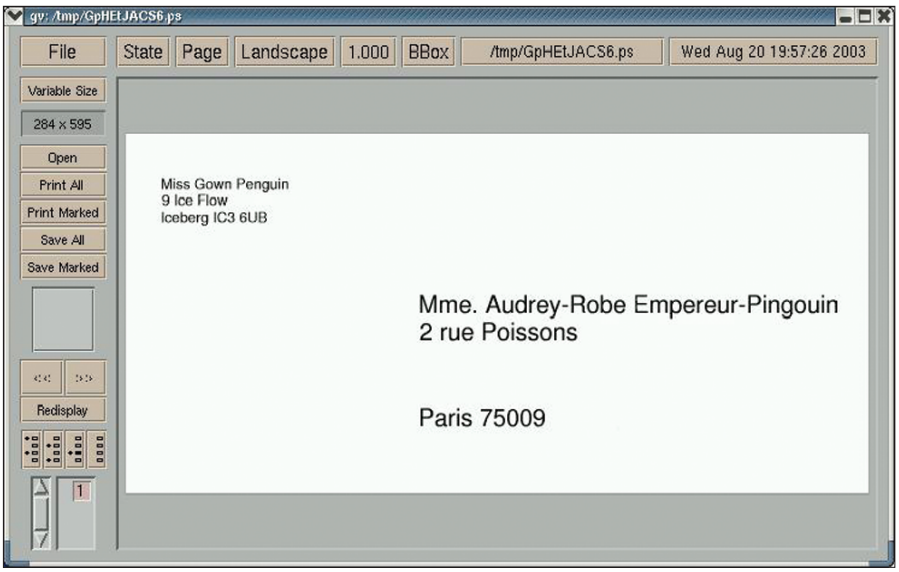

Figure 3: ... or long name, the offset stays the same

Short-lived

The *tempfile()* function from the *File::Temp* module creates a temporary file with the PostScript *.ps* suffix in line 34, and returns a writeable file handle and the file name.

The *output()* method which is called in line 56 writes the PostScript data to this file, but as it does not expect the *.ps* suffix, the suffix is first removed in line 55 and then the result is written to *$base*.

After calling the printer command in line 59, it is down to line 63 to remove the now obsolete temporary file.

textbox() in line 68 creates a new *PostScript::TextBlock* object and calls its *addText* method. It expects the font name, size, the line spacing value (*$leading*), and the text to be set.

To determine the size of the textbox that will be generated, it then calls *tb_width()* and *tb_height()* (*tb* for text block) defined further down.

Whereas *tb_height* simply needs to multiply the line spacing by the number of lines passed to it, calculating the amount of horizontal space used is more complicated, as the glyphs ot the proportional font are varying in width.

Font Metric Wizardry

Fortunately, there is a module called *PostScript::Metrics* with a function called *stringwidth()*, that uses embedded font tables to resolve this issue. The bad news is, that the module has never heard of *Helvetica-iso*. Removing the *-iso* suffix in line 96 provides a simple workaround. But unfortunately, this prevents special characters from working. This leads to another workaround in line 101, where a few special characters are simply replaced by the letter *A*. Although this will not produce precise results, it did the trick in our example. The width of the text block is derived from the longest line in the block.

Admittedly, I did have to resort to a few hacks this time. But my excuse is that I had to find a way of working around the slightly incomplete implementation of the *PostScript::** modules. It's a price I was more than happy to pay, as it allows me to choose the freedom of Linux and avoids all those superfluous Windows boot procedures. ■

INFO

[1] Printer installation how-to for Linux: *http://www.linuxprinting.org*

[2] Shawn Wallace, "Perl Graphics Programming": O'Reilly, 2002

[3] More PostScript info: *http://www.mathematik.uni-ulm.de/help/pstut/*