by Jeffreys Copeland and Haemer

Drawing on the Net

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS' R&D group in Boulder, CO. He's been a software consultant to the Hugo award administrators for several years. He spends his spare time raising children and cats.*

Jeffrey S. Haemer (`jsh@canary.com`) *now works for QMS, too, and is having a great time. Before he worked for QMS, he operated his own consultancy firm, and did a lot of other things, like everyone else in the software industry.*

Allow us to give you a bit of background before we begin, for those of you who didn't read our column in *RS/Magazine* and don't know us yet. Jeffreys Copeland and Haemer have known each other for more than a dozen years. We met when we were both working for the now defunct, original UNIX vendor Interactive Systems Corp., Copeland in Santa Monica, Haemer in Estes Park. We were building opposite ends of a very early object-oriented word processing and database package. It took us about 20 minutes to discover that we had lived in the same house at Caltech, separated by eight years.

Since then, Copeland worked for SHL Systemhouse after it absorbed part of Interactive Systems but before it became a subsidiary of MCI Communications Corp. Haemer spent several years consulting on issues of internationalization, standardization and software portability. Now we're both working at QMS Inc., building the innards of laser printers.

This column was originally written as our 56th column for *RS/Magazine.* We began with a 12-part series on internationalization, moved onto a 17-part series on POSIX, did a quick three-part filler on literate programming, and now you're reading our 24th "Work" column. In "Work," we have been exploring problems we trip over in our daily work. Oddly enough, because we're software nerds, many of these problems have solutions that require writing some software. If you are interested in the software we have developed to date for these columns, take a look at `http://www.alumni.caltech.edu/~copeland/work.html` for a quick review. Unfortunately, now that *RS/Magazine* has suspended publication, we have been left in the middle of a multipart discussion of maps and HTML. With that, you're up-to-date, and we can proceed with our regularly scheduled column.

Every Vote Counts

Last time, we developed a CGI form to help us count the results of the recent U.S. presidential election. It wasn't until after the column went to press that we realized we had left out some important details for our non-U.S. (and our U.S.-based, but high school civics course-deficient) readers.

U.S. presidential elections are not as simple as those in some other countries. (The good news is they aren't as complicated as Israeli parliamentary elections; but then

again, nothing is.) Each state is allocated a number of electoral votes based on its members of congress, which is, in turn, based on its population. There are 538 electoral votes. The winner of the plurality of the popular vote in any given state is awarded all the electoral votes for that state. Thus, when Bob Dole won 46% of the vote in Colorado (to Clinton's 44% and Perot's 7%) he got all eight of Colorado's electoral votes.

Not only did we ignore the mechanics of the election, but we completely ignored the issue of how to display the results–We both voted for Mickey Mouse, but who *won* the election? This month, we'll address that issue.

As you may recall, this all started when our eldest daughter needed to learn the names of the states, and we wrote a quick Web page containing a clickable map of the United States–you click on a state, and it tells you the name. ("I think that one's Montana!" (click!) "Oops! It's North Dakota!") The map was static, so we didn't need to worry about the computing power required to redraw it.

However, when we were updating the election map, we needed to be able to redraw it quickly, with states colored in red for Bill Clinton, the Democrat, blue for Bob Dole, the Republican, and green for the Ferengi, Ross Perot.

For quick drawing, we will turn to the `GD` package in Perl. But where do we get the map that we're going to color?

Space, the Final Frontier

We began with map data from the Massachusetts Institute of Technology. (We don't have the exact `ftp` reference. We've had the data for years and have converted it from line segments into PostScript, but we think the MIT folks derived the original data from the CIA geographic database. We'll be happy to pass the maps to you on request.) These are the outlines of the 48 continental states in longitude-latitude coordinates. So we have a bunch of files named things like `CA` that contain lines like the following:

```
% Begin: CA California
newpath
-120.0109 42.0125 M
-120.0090 41.2002 L
-120.0121 39.7082 L
-120.0020 39.4411 L
-120.0086 39.3135 L
-120.0033 39.1623 L
-120.0092 39.1152 L
```

The data in that form leaves us with two problems. First, how do we render the PostScript into a GIF or JPEG image suitable for display on a Web page? Second, how do we extract the map data so that we can handle the HTML inline `MAP` telling us what position on the screen corresponds to what state on the ground?

Our original solution to the first problem was the complicated and time-consuming pipeline we used to draw the map in the first case:

- PostScript data piped through a script to color the map.
- Output of the script read into GhostScript to convert it to Jef Poskanzer's portable pixel map (PPM) format. (Yes, we could have converted it directly to GIF, *if* our version of GhostScript had had the GIF driver installed.)
- Rotate the PPM file 90 degrees so north points up.
- Convert the PPM to a GIF.

Whew! This took about 45 seconds on the SPARC 5s we normally use on our desktops and for our Web server at the office, which is not nearly quick enough.

We solve the problem by producing several files for each state, naming them MA_1, MA_2 and so on, and sticking the set in a subdirectory named MA.

Next, we investigated the Perl package `GD`, which leads us to the solution to both problems we posed earlier. `GD` is a graphics package for Perl (versions also exist for C and Tcl) that allows us to render our output directly into GIF. If we convert the original map data from longitude-latitude space to the appropriate coordinates for GIF space, we can use the same map data to both draw the map and to give us the outlines of the clickable regions displayed on the map.

The program to do the conversion is pretty simple and is shown in Figure 1. Basically, we read each line in the PostScript map data, and each time we encounter a line with an `M` (move and begin a new polygon), we stop and produce output. We convert the coordinates as we read them in, preserving our original Mercator projection at 12.6-GIF pixels per degree, appropriately offset so that Kansas is in the center of our picture.

That's not the whole story, however. We need to do some small adjustments in the output. We want to eliminate points that are identical in GIF space, as well as points that we don't need because they're on the same horizontal or vertical line as points surrounding them. This serves to compress the data to a shorter list of points for each state outline.

This compression takes place in the `print_stuff` routine, which is called each time we need to output a self-contained polygon of the state. We could improve the compression yet again by eliminating all points that lie in the middle of a line segment of any orientation, but it's not worth the computational hassle.

But that's not the whole story, either. Some states, such as Massachusetts and Michigan, are made up of more than one polygon. We solve the problem by producing several files for each state, naming them `MA_1`, `MA_2` and so on, and sticking the set in a subdirectory named `MA`. Making the subdirectory and splitting up the output file is left as an exercise for the reader. An alternate exercise for the reader: Modify our conversion to preserve the notion of one state, one file. Warning: You'll need to modify the `drawmap` pro-

gram, which we'll look at next.

And *that's* not all, either. We've produced the data to draw the maps but still haven't produced the data for the `MAP` production in the HTML file. You'll notice some lines commented out in `gifmap` with double ##. Those are the lines that provide the HTML wrapper for the in-line map.

That's the whole story. Next, we have to figure out how to redraw the map from the data.

A Map, a Map, My Kingdom for a Map

Take a look at Figure 2. It contains the code for `drawmap`, our program to redraw and color the map. There are several interesting features that we need to note.

First, usage. We use the program by giving the directory containing the map data, and a results file. The file contains something like:

```
CA      Clinton   red
CO      Dole      blue
```

Notice that we've added the -T flag. 'T' is for 'Taint' and checks for possible security breaches.

Strictly speaking, the names of the candidates are redundant: `drawmap` needs only the state names and colors.

Second, we can explore some of the innards of `drawmap`. (For space reasons, we're going to do this quickly.) We begin the program with our usual "shebang" line specifying the path of the Perl executable. Notice, however, that we've added the `-T` flag. "T" is for "taint" and checks for possible security breaches. For example, the `PATH` is set explicitly because the taint checking feature is turned on–without the explicit set, the program won't run.

We turn on taint checking because we're going to use this routine as part of a CGI script, and we want code used by folks outside our local net to have a higher level of suspicion about its environment.

Given that, we can begin talking about the `GD` package, which is available from the CPAN archive at `http://www.perl.com/cpan/`. It has an excellent man page as part of the installation set.

In general, when using `GD`, we define an image object using a line like this:

Figure 1. Converting to GIF Space

```
#! /usr/local/bin/perl
# generate the cgi outline map
# from the PS outlines

$n = 0;

while( <> ) {
 if( /^% Begin: ([A-Z][A-Z]) (.*)/ ) {
    $state = $1;
    $full = $2;
    $full =~ s/ /_/g;
    print "# $1 $2 \
\$Id: gifmap,v 1.3 1997/01/06 23:38:01 jeff Exp $\n";
 }

 if( /([\d.-]*) ([\d.-]*) ([LM])/ ) {
    print_stuff() if( $3 eq 'M' && $n );
    $x[$n] = int(12.6 * $1 + 1610.7);
    $y[$n] = int((-12.6) * $2 + 762.7);
    $n++;
 }
}
&print_stuff();

sub print_stuff
{
    ## print " <AREA SHAPE=poly \
    ## HREF=/cgi-bin/state.cgi?state=$state";
    ## print " ALT=\ "$state\" COORDS=\"";
    print "$x[0] $y[0]\n";
    # now we go through the points in sequence,
    # eliminating those we can::
    #  begin with identical ones
    for( $i = 1; $i < $n-1; $i++ ) {
      $x[$i] = 0 if( $x[$i] == $x[$i-1]
         && $y[$i] == $y[$i-1] );
    }
    #  skip one of vertically co-linear pairs
    for( $i = 1; $i < $n-1; $i++ ) {
      $y[$i] = 0 if( $x[$i] == $x[$i-1]
         && $x[$i] == $x[$i+1] );

    }
    #  skip one of horizontal co-linear pairs
    for( $i = 1; $i < $n-1; $i++ ) {
      $y[$i] = 0 if( $y[$i] > 0 &&
         $y[$i] == $y[$i-1] &&
         $y[$i] == $y[$i+1] );
    }
    # now print those not eliminated
    for( $i = 1; $i < $n-1; $i++ ) {
      print "$x[$i] $y[$i]\n"
         if( $x[$i] > 0 && $y[$i] > 0 );
    }
    print "$x[$n-1] $y[$n-1]\n";
    # assume the last point and the first
    # are the same to close the polygon
    ## print "\">\n";
    $n = 0;
}
```

Figure 2. The Code for drawmap

```
#!/usr/local/bin/perl -Tw
# $Id: drawmap,v 1.7 1997/01/06 23:29:52 jeff Exp $

use GD;
use FileHandle;
use DirHandle;
use Getopt::Std;

use strict;      # Perl's lint
use subs qw(draw_state draw_chunk);
use vars qw($opt_h $opt_d $opt_c %color);
$ENV{PATH} = '/usr/bin';

my $usage = "$0 [-h] [-c color_file] \
 [-d states_directory] [state ...]";
getopt("cd");
die $usage if $opt_h;

# Create an image object
my $im = new GD::Image(800,500);

# Give the image's colors symbolic names
my $white= $im->colorAllocate(255,255,255);
my $black = $im->colorAllocate(0,0,0);
my $red = $im->colorAllocate(255,0,0);
my $green = $im->colorAllocate(0,255,0);
my $blue = $im->colorAllocate(0,0,255);

# Make the background transparent and interlaced
$im->transparent($white);
$im->interlaced('true');

# Frame the picture
$im->rectangle(0,0,1000,1000,$black);

# Read state colors
my %c;
if ($opt_c) {
   my $fh = new FileHandle $opt_c
       or die "Can't read $opt_c: $!";
   while (<$fh>) {
       # next if /^\s*\#?.*$/; # skip comment lines
       # Color file has format
       #  "state_name candidate color"
       my($state, $c);
       ($state, undef, $c) = split;
       $c{$state} = $c;
   }
   close ($fh);
}

# Get a list of the states
my @states;
$opt_d ||= ".";   # the directory of state outlines
unless (@ARGV) { # individually named states
   # get all the states from the named directory
   my $dh = new DirHandle $opt_d
       or die "Can't read directory $opt_d: $!";
    @states = grep /^[A-Z][A-Z]$/, $dh->read;
   $dh->close;
}
@states = map {"$opt_d/$_"} @states;

 # Draw each of them
 foreach (@states) {
   my $state = $_;
   s|$opt_d/||; # strip directory and suffixes
   my $cname = $c{$_};
   draw_state $state, $cname;
 }

 # Convert the image to GIF and print
 print $im->gif;

 # Draw a state
 sub draw_state {
   my($state, $cname) = @_;

   # Most states are single files.
   # If the state needs to be drawn in chunks
   # we make the state a subdirectory,
   # and the chunks individual files

   # draw the whole state at once
   unless (-d $state) {
       draw_chunk(@_);
       return;
   }

   my $dh = new DirHandle $state
       or die "Can't read directory $state: $!";
   my @chunks = grep /\w/, $dh->read;
   $dh->close;
   foreach (@chunks) {    # draw it in chunks
   draw_chunk("$state/$_", $cname);
   }
 }

# Draw a chunk (and maybe color it)
sub draw_chunk {
   my($chunk, $cname) = @_;
   my (@x, @y);
   my $i = 0;

   my %color = (           # for convenience
       'white'   => $white,
       'black'   => $black,
       'red'     => $red,
       'green'   => $green,
       'blue'    => $blue,
   );

   # A "chunk" contains X,Y coordinates of the
   # vertices. It can also contain comment lines,
   # which have a '#' as the first non-whitespace
   # character. We like to allow comments.

   # Read coordinates of vertices
   my $fh = new FileHandle $chunk
       or die "Can't read $chunk: $!";
   while (<$fh>) {
       next if /^\s*\#/;   # skip comment lines
       ($x[$i], $y[$i]) = split;
       $i++;
   }
   close $fh;

   # Now draw the polygon
   my $poly = new GD::Polygon;
   for ($i = 0; $i < @x; $i++) {
       $poly->addPt($x[$i], $y[$i]);
   }
   if ($cname) {
       $im->filledPolygon($poly, $color{$cname});
   } else {
       $im->polygon($poly,$black);
   }
}
```

```
my $im = new GD::Image(800,500);
```

for an image 800 pixels wide by 500 high. We can allocate colors and add items to the image with lines like the following:

```
my $red = $im->colorAllocate(255,0,0);
$im->rectangle(150,150,250,250,$red);
```

which draws a 100-pixel red square outline with its lower left corner at (150,150). We finish up and render the picture to standard output with the line

```
print $im->gif;
```

Caveat emptor: There's a bug in the Perl version of `GD` that leaves blank stripes in flood fills of complicated polygons like the state of California. We don't know what the fix is yet, but if we find it, we'll pass it on.

One final thing to notice is that we've used Perl's `FileHandle` and `DirHandle` packages. The former is used to safely read the map components, and the latter is employed to open the directories for states consisting of multiple polygons. Note that the `Handle` package, which obsoletes both `FileHandle` and `DirHandle`, has been announced, but is not yet available–though it may be by the time you read this.

In Summary

Here's where we've done: Earlier, we showed you a CGI script to allow the input of candidate/state pairs for the electoral college, and now we've just finished another CGI Perl program to turn the data we generated into a colorful map. We'll leave you with one last exercise: Combine the two programs into a single CGI script that allows you to click on the candidate and state, and then redraws the whole map, which, in turn, you can click on.

Next month, we'll return to a topic of long ago, building things, with implications for Web page development.

Until then, happy trails.