The 2D Graphics Library in Perl

Tutorial and Reference Manual

by

Malay K Basu <curiouser@ccmb.res.in>

Version 0.04, May 8, 2003

Contents

Chapter 1

Getting Started

COMPUTER GRAPHICS ARE EITHER bitmapped raster format or vector format. The former is a pixel by pixel definition of the graphics a rasterizer displays the graphics just by just representing the pixels on screen. Vector data on the other hand is represented by abstract object notation and a rasterizer has to calculate the pixels to render before actually displaying the graphics.

Unfortunately, graphics over the web was traditionally of bitmap raster format (*e.g.*, GIF, JPEG, PNG). These graphics formats are not difficult to create but they have two great big flaws–the lack of scalability (*i.e.*, they lose resolution when scaled to a bigger size), and the text is not text (*i.e.*, the text also becomes a bitmap). They are also very bulky in nature, in spite of the existence of efficient compression algorithms. Raster graphics of course has sometimes more advantages over vector for pixel by pixel rendering. If your require tons of points you are better off using a raster format.

1.1 From raster to vector

There was of course an alternative the traditional raster format–vector images. Vector images by definition are created as instruction sets to the display machine. The display machine computes all the points and their attributes on the fly and generates the bitmap to show it to the users.

The advantage of this method of display is that the image immediately becomes resolution independent. All the modern fonts are nothing but vector instructions to the display machine, thus, text is vector display remains text.

For quite sometime there was only one dominant player for transferring vector data over the Web. Created by Macromedia, Flash®, became the standard for vector data distribution over the web. The Flash file format, though open-source is controlled by Macromedia.

The situation changed with the invention of a new file format for vector data called

Scalable Vector Graphics (SVG) by W3C.

1.2 A bit more about SVG

SVG is a representation of graphics in plain text, more precisely, in XML. It has all the graphics primitives, like, line, rectangle, circle and much much more. To create an SVG file you just write instructions like this:

```
<rect x="30" y="40" width="190" height="10" style="fill:#ff0000;stroke:
    none"/>
```

The above code actually draws a rectangle 190 pixels wide starting from coordinate (30, 40) with height 10 pixels. It also says that the rectangle should be filled with color solid red and the outline of the rectangle should not be stroked.

To actually draw this on screen you need to wrap this line with several SVG specific lines and then open this file into a viewer. We will not discuss this in details any further.

1.3 What is Pastel?

Pastel is a Perl2D library written in of course, Perl. The main features of the Pastel are:

1. Unlike the other SVG library, Pastel treats graphics object as graphics. That means, SVG is not dealt as XML but as a collection of graphics primitives. That means the particular file format not important to Pastel.

2. One of the very well designed 2D graphics API is Java2D graphics. Instead of reimplementing a completely new set of API, Pastel implements Java2D API. Pastel takes the best of Java2D API but discards the complex part of it. Pastel API is, therefore, perlish but still well-accessible to Java programmers.

3. Programmatically generating graphics requires tons of other features; not merely creating strings (using DOM or such API). You require at least a bare minimum support for computational geometry. Pastel implements all required computational geometry algorithms.

4. One of the nightmares of vector graphics format designer is the font issue. Pastel takes the full advantage any arbitrary fonts within file. Pastel also has sophisticated font-handling features. That means Pastel is ideally suited for server-side on-the-fly generation of SVG image that are heavily text-oriented.

1.4 Requirements

Pastel will work only for Perl 5.6 ans above. It takes advantage of the unicode support of Perl.

The SVG data generated by BioSVG can be viewed by any SVG viewer. The most common viewers are: Adobe SVG viewer (ASV), available from:

http://www.adobe.com/svg/viewer/install/main.html.

ASV is just a browser plug-in. Once installed, SVG data can be viewed on your browser. ASV is actually installed automatically with the Windows version of Acrobat Reader® version 5.0 onward.

The second most common viewer is Batik SVG viewer, available from:

http://xml.apache.org/batik/index.html.

Batik is written in Java, therefore, can be run in any platform. Batik is a stand-alone SVG viewer.

1.5 Installation

Follow the standard Perl way:

```
make
make test
make install
```

1.6 Code-convention using Pastel

A very minimal Pastel file:

```
#!/usr/bin/perl;
use Pastel;
my $graphics = Pastel::Graphics->new(); # create the graphics context
$graphics->show(); # to dump the output to STDOUT
$graphics->get_svg(); # to get the SVG document as strings
```

Note that there are two ways to get the SVG. You can dump it on STDOUT, a method may be more preferable in case of CGI, or you can get the whole string, may be to embed the SVG document into an HTML file.

All constructors should be called with named parameters:

```
my $rect =
   Pastel::Geometry::Rectangle->new(
              -x=>20,
              -y=>30,
              -width=>250,
              -height=>300 );
```

Note that the constructor, in this case, `new()` is called with named parameters. The sequence of parameters could be interchanged.

The method calls does not require this named parameters:

Thus, you have to call this:

```
$object->some_method(5, 10);
```

1.7 Some design decisions

1.7.1 Exception handling

There are some framework in Perl where they go a roundabout way to make Perl as close as Java. They create an elaborate framework of exception handling. I humbly differ from their opinions and believe that Perl is not meant to be used that way. Most of the calls in Pastel actually die in case of an internal error. Although stalwarts might disagree but it's my belief (a belief can't be argued) that this is the best way. And I have an argument for it- it actually keeps the code simple!

But it is there if one requires it. Look at documentation of `Pastel::Exception` for details.

1.7.2 Graphics context

User should create an object of `Pastel::Graphics` class. This object is called Graphics context and stores all the necessary information for correctly drawing graphics. The context is created like this:

```
use Pastel;
my $graphics = Pastel::Graphics->new(-width=>800, -height=>600);
```

The width and the height parameters are the width and height of the SVG.

The two most important method is this class are `draw()` and `draw_string()`. Which are used to draw classes inherited from `Pastel::Shape` and `Pastel::String` class, respectively. All the classes in `Pastel::Geometry` are derived from `Pastel::Shape`. And two classes `Pastel::String` and `Pastel::Text::AttributedString` can be drawn by the `draw_string()` method.

Here, how you will draw some shapes:

```
my $rectangle = Pastel::Geometry::Rectangle->new(-x=>50,-y=>60,-width
    =>200,-height=>300);
$graphics->draw($rectangle);

# Now write the ubiquitous hello world
$graphics->draw_string("Hello world", 100, 150);

# Now show the graphics
$graphics->show();
```

Note that we have used straight way a Perl string instead of `Pastel::String` object as parameters to `draw_string()` method.

The graphics context also strore the color and the stroke. If you set the color and the stroke of the graphics context, all the subsequent drawing will used the color and the stroke currently set in the context.

Here we stroke a red-outlined and blue colored ellipse.

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

# create the graphics context
my $graphics = Pastel::Graphics->new();

#create the ellipse
my $ellipse = Pastel::Geometry::Ellipse->new(-x=>20,-y=>30,-width=>200,-
    height=>150);

# set color to blue
$graphics->set_paint(Pastel::Color->blue());

# fill the shape with blue color
$graphics->fill($ellipse);

# set the color for stroke
$graphics->set_paint(Pastel::Color->red());

# set the stroke
$graphics->set_stroke(Pastel::BasicStroke->new());

# Now draw the ellipse
$graphics->draw($ellipse);

# create something to write
my $text = Pastel::Text::AttributedString->new("Pastel␣is␣cool!");
$text->add_attribute("ANCHOR","MIDDLE");
my $font = Pastel::Font->new("Lucida","bold",14);
$text->add_attribute("FONT", $font);
$text->add_attribute("COLOR", Pastel::Color->white());

# get the centre of ellipse
my $x = $ellipse->get_x() + $ellipse->get_width() / 2;
my $y = $ellipse->get_y() + $ellipse->get_height() / 2;

# draw the string in the middle of the ellipse
$graphics->draw_string($text, $x, $y);

$graphics->show();
```

The output is shown in Figure 1.1.

FIGURE 1.1