CHAPTER 4

Drawing

Unless you already have all the graphics you will ever need, you have to create them; and if you already had everything you needed, you wouldn't be reading this book, so I assume that you do need to create some graphics yourself. This part of the book covers how to do this. Throughout the book you will find many ways in which to create graphics, such as charts, images for your web pages, or animations. In this particular chapter we'll have a look at the groundwork for most computer graphics generation: drawing.

In the most basic sense, drawing computer graphics consists of the manipulation of certain *drawing primitives*, such as circles, rectangles, lines and text. No matter which drawing package you use, you will find the same, or very similar, primitives, methods, and functions. All drawing packages also have in common a coordinate space in which to work, and the objects and primitives you work with are all expressed in terms of this coordinate space. For example, you draw a line from coordinates (12,12) to (30,0) or a circle with the center at (50,0) and a radius of 20. What these coordinates express depends on the drawing package: some use pixels, others points, centimeters, or inches. It is always possible to translate between these different coordinate systems, so it doesn't really matter what is used natively in a drawing package. When the result needs to be imported in another package, the relevant coordinate space transformations can be taken into account.

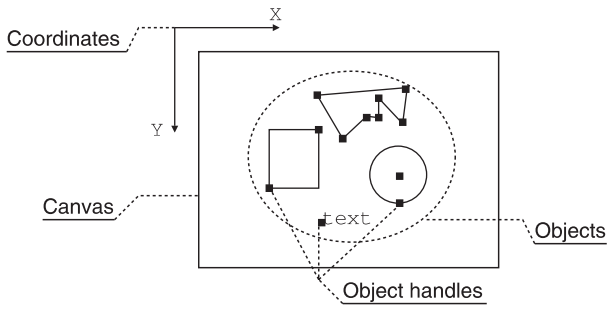


Figure 4.1
Some drawing primitives that can be created and manipulated with computer graphics programs and packages.

The drawing primitives for almost all graphics packages, Perl modules or otherwise, are limited to a fairly small set: lines, rectangles, circles, arcs, and polygons. One could argue that all that is really needed are a line and an arc, because all other primitives can be created from those. Nonetheless, the other primitives, such as the circle and rectangle, are sufficiently common to see them as primitives in their own right, especially when they need to be filled as well. Even though most drawing packages provide these primitives, each does it in a slightly different way.

One other previously mentioned primitive, which almost all packages define, is text. The handling of text is generally more complex than the other primitives, because it involves selections of fonts, calculations of bounding boxes and alignment concerns. These issues will be discussed in section 4.1.3 on page 43, section 4.2.4 on page 50, and in more detail in chapter 11.

The differences between drawing packages starts with the definition of a coordinate system. For some graphics applications, the coordinate origin is the bottom left of the canvas, and for others it's the top left. Some drawing packages work in real-life coordinates, such as inches or millimeters, and others work in pixels. I haven't seen any applications that have a coordinate system that is not Cartesian, but there is nothing that says it has to be, and in section 10.1.1, "Coordinate transformation," on page 180, we present a case in which a polar coordinate system is more convenient. Fortunately, the two main drawing packages for Perl, *GD* and *Image::Magick,* both use the same coordinate system: with the origin in the top left and the coordinates in pixels (see figure 4.1).

Many primitives can be specified in several ways. For example, the simple rectangle can be uniquely identified by providing the coordinates of one of its corners and the width and height of the rectangle. Alternatively, one can specify one of its corners and the corner on the other side of the diagonal. A circle can be specified as the coordinate of its center and a radius. It can also be specified as three points that are not on a straight line.[1] Another alternative is the coordinates of the center and the coordinates

[1] Euclidean geometry teaches us that a circle can always be drawn through any three points that are not on a straight line.

of one point on the circle itself. The only way to find out which of these specifications a drawing package uses, is to read its documentation.

In order to deal with all these ways of representing objects on a canvas, it helps if you are comfortable with geometric algebra, and can easily translate any of the specifications into something with which you are more comfortable.

In this chapter we'll look at how the various modules available for Perl deal with drawing primitives, what some of these primitives are, and how they can be used.

4.1 DRAWING WITH GD

GD provides a range of drawing primitives and some auxiliary methods, such as color allocation, color flood fill, and methods to set a brush. These methods are listed in table 4.1, and the text drawing methods are listed later in this chapter, in table 4.3 on page 45.

Table 4.1 The drawing primitives for the *GD* module which are available to the user of the module to create images. All of these methods should be called on a *GD::Image* object.

setPixel(x,y,color)	set the color of the pixel at the specified coordinates
line(x1,y1,x2,y2,color)	draw a solid line between the specified points
dashedLine(x1,y1,x2,y2,color)	draw a dashed line between the specified points
rectangle(x1,y1,x2,y2,color)	draw a rectangle with the specified corners
filledRectangle(x1,y1,x2,y2,color)	draw a filled rectangle with the specified corners
polygon(poly,color)	draw the polygon specified by poly, which is a polygon object created with GD::Polygon::new()
filledPolygon(poly,color)	draw the filled polygon specified by poly
arc(cx,cy,w,h,st,end,color)	draw an arc with the specified center, width and height, and start and end angle
fill(x,y,color)	flood-fill all pixels with the same color around the specified point
fillToBorder(x,y,bcolor,color)	flood-fill all pixels around the specified point, until a pixel with the specified bcolor is encountered
setBrush(brush)	Set the brush to be used to the specified brush, which is another *GD::Image* object
setStyle(color-list)	Set the line style to the specified color list. Each element of the list represents a single pixel on the line to be drawn.

Most drawing methods accept a color argument, which is either the index of a color previously allocated with one of the color allocation methods, or one of `gdBrushed`, `gdStyled` or `gdStyledBrush`. These three will cause the object to be drawn with the image's current brush, line style, or both.

NOTE Unfortunately, there is no filled form of the `arc()` drawing primitive, which can cause some hardship when trying to create filled partial circles or ellipses. There also is no variant of the arc primitive which allows you to specify a start and end point of the arc. Thus, it is almost impossible to reliably create a pie slice (or a pizza slice) that can be flood-filled. Because of little rounding errors it is possible that there are gaps between the arc that forms the outside bit of the slice, and the lines that form the wedge. When subsequently one of the fill methods is called, these little gaps will cause the flood-fill to leak out into the rest of your drawing. If there were an arc command that allowed one to specify start and end points, it would be possible to reliably close these gaps.

GD has its coordinate origin defined in the top left-hand corner of the canvas, at coordinate point (0,0). This means that all pixel coordinates are offset at 0, and the coordinates of the center point of a 100 by 100 pixel image are at (49,49).

In order to use colors in *GD* you first need to allocate them for your object. The total number of colors in a *GD::Image* object is limited to 256, which for most drawing purposes is fine (see also section 12.1, "GD and pixels," on page 211). You will find that you normally don't use more than about 10 colors anyway.

NOTE At the moment of writing, Thomas Boutell has a beta version of version 2 of libgd available for download. This new version supports true color images and a real alpha channel, and should become even more useful for all kinds of applications. Once this version of libgd stabilizes, I suspect that Lincoln Stein will release a new version of *GD*.

4.1.1 An example drawing

The best way to present how this works is to show some code:

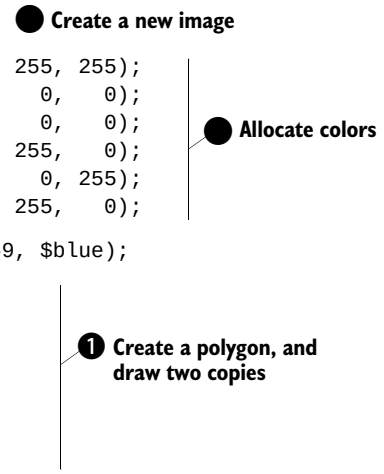
```
use GD;

my $gd = GD::Image->new(400,300);

my $white  = $gd->colorAllocate(255, 255, 255);
my $black  = $gd->colorAllocate(  0,   0,   0);
my $red    = $gd->colorAllocate(255,   0,   0);
my $green  = $gd->colorAllocate(  0, 255,   0);
my $blue   = $gd->colorAllocate(  0,   0, 255);
my $yellow = $gd->colorAllocate(255, 255,   0);

$gd->filledRectangle(0, 129, 199, 169, $blue);

my $poly = GD::Polygon->new();
$poly->addPt(199, 149);
$poly->addPt(399,  74);
$poly->addPt(324, 149);
$poly->addPt(399, 224);
$gd->filledPolygon($poly, $yellow);
$gd->polygon      ($poly, $black);
```



```
$gd->arc(199, 149, 250, 250, 0, 360, $red);
$gd->arc(199, 149, 100, 200, 0, 360, $red);
$gd->fillToBorder(99, 149, $red, $green);
```

❷ **Create a circle, cut out an ellipse, and fill**

```
$gd->rectangle(0, 0, 399, 299, $red);
$gd->line(199, 0, 199, 299, $red);
$gd->line(0, 149, 399, 149, $red);
```

● **Frame with a red border, and draw a red cross**

The output of this program can be seen in figure 4.2.

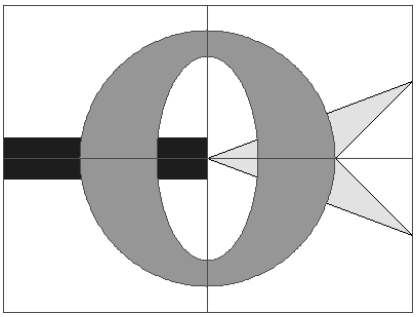


Figure 4.2
A simple drawing created with *GD*, demonstrating how to allocate color and use the `arc()`, `line()`, `rectangle()`, `filledRectangle()`, `filledPolygon()`, and `fillToBorder()` drawing primitives and instructions.

A new image is created and a blue-filled rectangle is drawn around its horizontal center in the left half of the image.

❶ To draw a polygon, one first needs to create one with *GD::Polygon*'s `new()` method, and all the vertices need to be added to this polygon in order. In this case the polygon takes the shape of an arrowhead, pointing left, so we add the necessary points to the object. In general, it is not a bad idea to wrap this in a subroutine, one that possibly also takes a scaling parameter that can be used in the `map()` method which is provided for *GD::Polygon* objects. When all points have been added, the object can be used with the `polygon()` and `filledPolygon()` methods to actually draw it.

❷ The first call to the `arc()` method draws a red circle with a diameter of 250 pixels around the center of the image. Once that is done, an ellipse with a width of 100 pixels and a height of 200 pixels is drawn inside this circle. The circle and the ellipse now form the outlines of the big letter O that can be seen in figure 4.2. To fill the space between these two, a suitable point is picked in that area, and the `fillToBorder()` method is called, with the instruction to flood-fill until it encounters pixels that are red.

This example shows how to use most of the primitives provided by the *GD* module. The documentation that comes with *GD* is quite clear and extensive, and if you spend the minimum amount of time reading through it, you'll see how simple it is to use this module to create your drawings.

Real-life programs that need drawing capabilities will be only superficially more complex than the one presented here. The hardest job in graphics programming is defining what needs to be done, and subsequently mapping these requirements to the various primitives. In other words, the real complexity in graphics programming isn't

writing the programs, but coming up with a clear way in which, with a limited set of tools, an objective can be reached.

4.1.2 Filling objects

It could prove useful to discuss filling of odd shapes, or even overlapping shapes, correctly with *GD*. The `fill()` method will color all the pixels that have the same color and that are adjacent to the target pixel. In other words, it will fill an area of uniform color that contains the target pixel. This works fine provided your drawing is not complex, and is relatively free of overlapping shapes. If however, the area that you want to fill isn't entirely uniform, but contains lines or edges in another color, this method cannot be used. Instead, you can use the `fillToBorder()` method.

The `fillToBorder()` method can be used to change the color of all adjacent pixels until another specified color is encountered. This last method can be used to create some interesting shapes, and fill them correctly, regardless of the other colors already present. But what if the image already has lines or blobs in the area that you need to fill, and these lines are of the identical color you wish to use for your border? You solve that problem by keeping one special color around, which you use only to create fill borders. Since colors in *GD* are always specified as an index, it doesn't even matter what RGB values you use for this.

Let's demonstrate this: suppose you need to add some more elements to the image from figure 4.2, with the same colors already present in the image. You decide to use the `colorResolve()` method to get at those colors, so that you don't accidentally run out of room in the palette. The element you want to add is a partial ellipse, and you want it to have a black border and a green inside.

You draw the partial ellipse and the boundary lines with the black color that you obtained from the `colorResolve()` method. Then you use `fillToBorder()` to fill the resulting figure with green, only to find out that the fill stops at the edges of the yellow polygon. You can solve this by creating a temporary color that you use as a boundary.

```
use GD;

my $gd = GD::Image->newFromPng('GDExample.png');

my $black  = $gd->colorResolve(  0,   0,   0);
my $green  = $gd->colorResolve(  0, 196,   0);

my $border = $gd->colorAllocate(0,    0,    0);

$gd->arc (199, 149, 300, 100, 0, 270, $border);
$gd->line(199, 149, 199,  99, $border);
$gd->line(199, 149, 349, 149, $border);
$gd->fillToBorder(149, 149, $border, $green);

$gd->arc (199, 149, 300, 100, 0, 270, $black);
$gd->line(199, 149, 199,  99, $black);
$gd->line(199, 149, 349, 149, $black);

$gd->colorDeallocate($border);
```

Allocate colors, and reuse ones that exist

Allocate a color for the border

Draw the arc's border, and fill

1 Redraw in the wanted color

2 Remove the border color

The result of this code can be seen in figure 4.3. Note that the `$border` color is exactly the same color, by RGB value, as `$black`. However, since each color in *GD* is uniquely identified by a palette index, they are in effect different colors, their appearance notwithstanding. It matters little which RGB values you choose for the temporary border color.

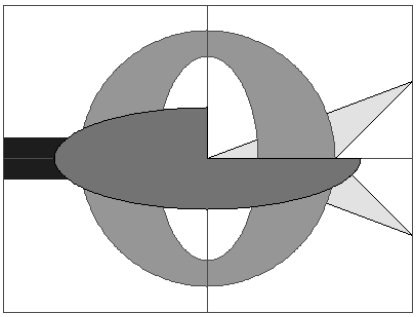


Figure 4.3
Adding a filled shape to an image with *GD*, using fillToBorder() with a temporary color for the border to make sure that the resulting filled shape is correct.

❶ After the flood fill has stopped, the color you used as a border color is still visible as a line around the object just filled. To resolve this, you can redraw the original shape, but this time in the desired color. For fully solid shapes, this would be the same color as the one used to fill the object.

❷ When the border color is no longer needed, it should be deallocated. This serves mainly to prevent repeated reads and writes of the same image, together with the operations discussed here, resulting in a palette which is filled with useless duplicate border colors. If this were not done, you would have to clean up the palette every time you detected that you were running out of index entries. (See section 12.1.2, "Removing duplicate color palette entries," on page 212, for a way to do this.)

Using this technique you can create shapes as complex as you wish. As long as these shapes can be limited on all sides by a unique color (and this should be a temporary color), you can fill the shape with any other color you want, after which you can redraw the boundaries in the colors you intended initially. In fact, it is not a bad idea to always use this method when you use `fillToBorder()`. It requires a few more lines of code, but is also much more portable and safe. If you ever move your code fragment that draws this complex shape into a subroutine, and start using that subroutine in all kinds of programs, you will benefit mightily from making certain that the border color you use will be unique in the whole image.

4.1.3 Drawing text with GD

The GD module has various ways to draw text. First, you can use one of the set of compiled-in fonts. These fonts are all of a fixed width and height, and are *GD::Font* objects. There are several methods available for these objects that allow the programmer to

extract some information about them. The built-in fonts can only be drawn horizontally or vertically.

In later versions of GD, you can also use TrueType fonts. These can be scaled to virtually any size and drawn at any angle. The TrueType fonts in *GD* are not wrapped in an object, but enough information about the text to be drawn can be obtained through the use of the `stringTTF()` method.

4.1.4 GD's built-in fonts

The built-in font types in GD can be used by specifying their symbolic short name, or by invoking the *GD::Font* package method associated with them. The fonts and their character sizes in pixels are:

Table 4.2 The names of the built-in fonts in *GD* as aliases, and as fully specified *GD::Font* objects. Either of these names can be used anywhere a *GD::Image* method requires a font name.

short name	font object	size
`gdTinyFont`	GD::Font->Tiny	5 x 8
`gdSmallFont`	GD::Font->Small	6 x 13
`gdMediumBoldFont`	GD::Font->MediumBold	7 x 13
`gdLargeFont`	GD::Font->Large	8 x 16
`gdGiantFont`	GD::Font->Giant	9 x 15

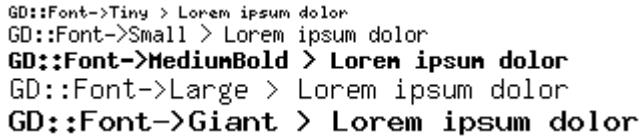


Figure 4.4
The built-in fonts for *GD*.

These fonts are not pretty, but they do a reasonably good job for most low-resolution graphics, such as the ones you would display on a web page. Additionally, because they are compiled in and of a fixed size, they are really fast to use.

For the built-in fonts, you can use the `string()`, `stringUp()`, `char()` and `charUp()` object methods. The last two methods are an inheritance from the C interface of `libgd`; in Perl, of course, there is no formal distinction between characters and strings.

```
$im->string  (gdMediumBoldFont, 0, 0, 'Black Medium Bold', $black);
$im->stringUp(gdGiantFont,  0, 100, 'Red Giant', $red);
$im->char    (gdSmallFont, 50,  50, 'A', $black);
$im->charUp  (GD::Font->Tiny , 60,  50, 'B', $black);
```

This example uses both the short name and the full *GD::Font* object name as arguments to the methods; they are interchangeable. The coordinates that these functions expect denote the upper left corner for `string()` and `char()`, and the lower left corner for `stringUp()` and `charUp()`. Of course, from the string's perspective, these are both the upper left corner, and maybe that's a better way to look at it.

There are some methods available for the *GD::Font* objects that can be used to retrieve information.

```
$nchars = gdGiantFont->nchars;
$offset = GD::Font->Tiny->offset;
$width  = gdSmallFont->width;
$height = gdSmallFont->height;
```

The `nchars()` method tells us how many characters this font provides, and the `offset()` method gives us the ASCII value of the first character in the font. The `width()` and `height()` methods return the dimensions of a single character in the font, which is useful, since that means we don't have to hard-code font sizes in our programs.

Table 4.3 The text drawing primitives for *GD*. The top part of the table shows all methods that work on the built-in fonts, and the bottom part of the table shows the method that can be used for TrueType fonts. Note that the possibilities of drawing TrueType strings are more flexible and advanced than for built-in fonts.

string(font,x,y,string,color)	draw a horizontal string starting at the specified point
stringUp(font,x,y,string,color)	draw a vertical string starting at the specified point
char(font,x,y,char,color)	draw a single character starting at the specified point
charUp(font,x,y,char,color)	draw a vertical character starting at the specified point
stringTTF(color,font,size, angle,x,y,string)	draw a string with a TrueType font of the specified size starting at the specified point

4.1.5 TrueType fonts and GD

Versions of GD since 1.20 also support TrueType fonts, which can be drawn with the `stringTTF()` method. This method can be called as an object method (`$gd_object->stringTTF()`), in which case it draws the string on the calling object, or as a class method (`GD::Image->stringTTF()`), in which case it returns the bounding box of the text to be drawn, without actually drawing it. The list returned from this method consists of eight elements, denoting four coordinate pairs: the *x* and *y* coordinates of the lower left, lower right, upper right and upper left corner, in that order. The `stringTTF()` method draws the string aligned to the left side of the baseline of the font.

The following code first requests the bounding box for a string, and then adapts the coordinates by subtracting the horizontal and vertical offset from the baseline. This ensures that the string is aligned to the requested coordinates with its upper left corner.

```
$text     = 'String';
$font     = '/usr/share/fonts/ttfonts/arialbd.ttf';
$fontsize = 12;
$angle    = 0;
($x, $y)  = (20, 25);

@bb = GD::Image->stringTTF(0, $font, $fontsize, $angle, $x, $y, $text);
$x += $x - $bb[6];
$y += $y - $bb[7];
$im->stringTTF($black, $font, $fontsize, $angle, $x, $y, $text);
```

You might wonder why we bother recalculating the *x* coordinate as well as the *y* coordinate. That is because sometimes the bounding box returned by `stringTTF()` is slightly offset to the right; in other words, the baseline used to draw the string is not always exactly vertically aligned with the requested coordinates. If you don't need such precision, you can dispense with this check.

SEE ALSO More manipulations of strings with the *GD* module can be found in chapter 11, as well as examples on how to align and position strings precisely.

4.2 DRAWING WITH IMAGE::MAGICK

Image::Magick possesses a wealth of methods, but few are actually designed to help create drawings. In fact, only one method is really useful, and that is the appropriately named `Draw()` method. There are quite a large number of options that can be passed to the method, and there are quite a number of primitives.

4.2.1 An example drawing

Let's see how the code would look in creating the same drawing as we did with *GD* in the previous section.

```
use Image::Magick;

my $im = Image::Magick->new(size => '400x300');

$im->Read('xc:white');
$im->Set(stroke => 'red');

$im->Draw(primitive => 'rectangle',
          points    => '0,129 199,169',
          fill      => 'blue',
          stroke    => 'blue');

$im->Draw(primitive => 'polygon',
          points    =>
            '199,149 399,74 324,149 399,224',
          fill      => 'yellow',
          stroke    => 'black');

$im->Set(antialias => 0, fuzz => 15);
```

Create a new empty image object

Add a white image, and set the default color

Draw a rectangle

Draw a polygon

2 Tweak some parameters

```
$im->Draw(primitive => 'circle',
        strokewidth => 3,
          points    => '199,149 74,149');
$im->Draw(primitive => 'ellipse',
        strokewidth => 3,
          points    => '199,149 50,100 0,360');
$im->Draw(primitive     => 'color',
          method        => 'filltoborder',
          points        => '99,149',
          bordercolor   => 'red',
          fill          => 'green1');
$im->Draw(primitive => 'rectangle',
          points    => '0,0 399,299');
$im->Draw(primitive => 'line',
          points    => '199,0 199,299');
$im->Draw(primitive => 'line',
          points    => '0,149 399,149');
```

❶ **Draw a circle and cut out an ellipse**

Flood-fill the gap

Frame with a red border and cross

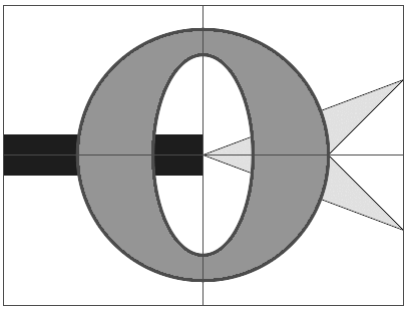


Figure 4.5
An example drawing created with *Image::Magick*, demonstrating the use of the `rectangle`, `polygon`, `circle`, `ellipse`, `filltoborder`, and `line` drawing instructions. The border lines need to be wide to avoid the fill color bleeding out of the circles, due to antialiasing.

As you can see, the code for *Image::Magick* is much more verbose than that for *GD*, even if we make use of a default drawing color. It is, however, not difficult to follow. To understand what the various parameters to the `Draw()` method mean, and how the points attribute changes its meaning for the various primitives, read the description of `Draw()` on page 257 in appendix A.

❶ The output of this example can be seen in figure 4.5. You immediately notice that the lines in this image are much wider than the lines in figure 4.2. You might have also noticed that these wide lines have been explicitly requested in the code. We will look at why, in this particular example, such wide lines were necessary.

4.2.2 Anti-alias and fuzz

❷ The use of the anti-alias and fuzz attributes warrants more explanation. By default, *Image::Magick* will anti-alias all objects it draws. However, since we want to use the red ellipses as boundaries to fill to, we need them to be of a reasonably uniform

color, and anti-aliasing removes that uniformity. Even with anti-aliasing turned off, we still need fuzzy color matching, because of the way these algorithms work in *Image::Magick*. The fuzz attribute allows the color matching algorithms to match a color *near* the one we have specified, instead of matching only the exact color. The higher the value specified for fuzz, the more lenient the color matching will be.

Another solution could have been to draw the circle with an anti-aliased line width of 5, and to use a higher fuzziness. As it is, we already need to use a line width of 3 to prevent the fill color to bleed through. Either of the two methods can be used, and both are equally hard to get right. If at all possible, it is much better to stroke and fill in one go with *Image::Magick*.

NOTE While playing with the anti-alias and fuzz attributes allows us to create shapes that we can successfully fill, it does require wide lines, and that is just not always desirable. Apart from that, it is not really possible to come up with a good set of rules that can be used to pick a decent value for the fuzz attribute. Instead, we have to rely on manually tweaking this value until the drawing looks as close to what we want as possible. We'll explore a few alternative ways of achieving our goal in the next sections.

4.2.3 Drawing by combining images

Another way to achieve the drawing we want is to create two images and combine them. In the code in the previous section, everything between the drawing of the polygon and the drawing of the red box can be replaced by the creation of a new image, on which the green O is drawn:

```
my $im2 = Image::Magick->new(size => '400x300');
$im2->Read('xc:white');

$im2->Draw(primitive => 'circle',
           stroke    => 'red',
           fill      => 'green1',
           points    => '199,149 74,149');
$im2->Draw(primitive => 'ellipse',
           stroke    => 'red',
           fill      => 'white',
           points    => '199,149 50,100 0,360');
$im2->Transparent(color => 'white');

$im->Composite(image   => $im2,
               compose => 'Over');
```

Create a new white image

Make the background transparent

Combine with other image

The output of this code can be seen in figure 4.6. It is very similar to figure 4.5, except that the outlines of the O are thinner, and more smooth. However, the combination of the two images has introduced some slight artifacts where the O overlaps the polygon and the rectangle. This is due to the fact that *Image::Magick* has anti-aliased the edge of the circle it drew against the background of the secondary image, which was white. The call to the `Transparent()` method makes all white pixels in

the image transparent, but the determination of which pixels are white is, once again, subject to the fuzzy matching that *Image::Magick* employs. The artifacts that show up are the pixels that weren't considered to be white.

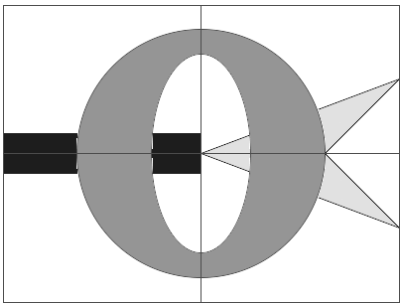


Figure 4.6
Creating a drawing with *Image::Magick* by overlaying several separate images which each represent part of the complete drawing.

The artifacts on the outside can be removed by starting with a transparent image, instead of a white one:

```
$im2->Read('xc:none');
```

This leaves the pixels on the inside of the O. Of course, we could simply increase the fuzziness of the matching algorithm before calling `Transparent()`. In this particular case that would probably work well, but it is not a universally workable solution. If we could fill the inner ellipse with a fully transparent color, there would be no artifacts; but unfortunately, setting the fill color to *none* or another color with full transparency (such as #000000ff) doesn't help, and the transparency will be silently ignored.[2]

There is one other solution that combines images, and that is to create a transparency mask for the secondary image:

```
my $im2 = Image::Magick->new(size => '400x300');
$im2->Read('xc:white');
$im2->Draw(primitive => 'circle',
           stroke    => 'red',
           fill      => 'green1',
           points    => '199,149 74,149');
$im2->Draw(primitive => 'ellipse',
           stroke    => 'red',
           fill      => 'white',
           points    => '199,149 50,100 0,360');

my $mask = Image::Magick->new(size => '400x300');
$mask->Read('xc:white');
$mask->Draw(primitive => 'circle',
            stroke    => 'black',
            fill      => 'black',
```

[2] It is anticipated that this will be fixed in a future release of *Image::Magick*

```
            points    => '199,149 74,149');
$mask->Draw(primitive => 'ellipse',
            stroke    => 'black',
            fill      => 'white',
            points    => '199,149 50,100 0,360');

$im2->Composite(image   => $mask,
                compose => 'ReplaceMatte');
$im ->Composite(image   => $im2,
                compose => 'Over');
```

Calls to `Transparent()` are eliminated because the transparency is created by the mask. The mask is created by repeating the drawing instructions on the new image, but with the colors set to either black or white, depending on what you want to be transparent and what opaque. A fully black pixel in the mask will result in an opaque pixel after the composition with the `ReplaceMatte` method, and a fully white pixel will result in a transparent pixel.

SEE ALSO More discussion on how image composition works in *Image::Magick* can be found in section 8.2.2, "Combining Image::Magick images," on page 142, more on transparency and alpha masks in section 12.4, "Alpha channels and transparency," on page 229, and a description of the `Draw()` method and all its arguments in appendix A on page 257.

While the result of this method is reasonably acceptable, it is a lot to deal with, and the code to create images like these often becomes quite verbose. Overlaying images can be a very useful tool (see for example sections 8.2, "Combining Images," on page 140 and 12.4, "Alpha channels and transparency," on page 229), but it does not entirely provide us with the desired outcome for this particular task. Let's have a look at another method to achieve the drawing we want.

4.2.4 Drawing with paths

The two previous examples required several commands, and either some artful manipulation of fuzzy color matching or extra images to draw the filled O shape. There is a third way to draw this figure: recent versions of *Image::Magick* have introduced the *path* drawing primitive, based on the drawing paths in the Scalable Graphics Vector format.[3] These paths are formed by positioning a virtual pen and drawing strokes with this pen, based on a string of commands. The letters in the string define a drawing instruction, and the numbers following those letters are the arguments and coordinates for the command. Lowercase letters take relative coordinates to the current point, and uppercase letters take absolute coordinates; for example, a capital *M* means *move to* the coordinates that follow, and a lowercase *a* indicates an *arc* drawn relative to the current coordinates.

[3] For more information on the SVG format, see page 281. The *path* primitive is explained in more detail in appendix A.

With paths, all the work needed in the previous examples can be done with one single operation. The code to draw just the O shape is:

```
$im->Draw(
    primitive   => 'path',
    stroke      => 'black',
    fill        => 'green1',
    points      => 'M 74,149
                    a 125,125 0 0,1 250,0
                    a 125,125 0 0,1 -250,0
                    M 149,149
                    a 50,100 0 0,1 100,0
                    a 50,100 0 0,1 -100,0 z');
```

and the result of that code can be seen in figure 4.7. The remainder of the code stays exactly the same. As you can see, this avoids both the fat lines from the first example, as well as the introduced artifacts of the second example.

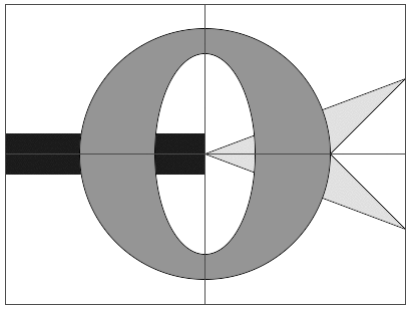


Figure 4.7
Drawing with *Image::Magick*'s path primitive, which allows complex filled shapes without artifacts.

The first line of the path specified in the example positions the pen at the coordinates (74,149). The arguments to the arc command *a* are the horizontal and vertical radius, the rotation of the arc, two flags that indicate the direction in which the arc will be drawn (see figure A.9 on page 283), and the second point through which to draw the arc. The coordinates of this second point are specified relative to the current position of the pen which is, of course, also the first point through which the arc runs. The path is closed by ending the command string with a *z*.

Which technique to use depends on your needs and the complexity of what you want to achieve. Most drawing tasks won't require the combining of images, but knowing that it can be done allows you to produce a drawing that most closely approximates what you intended to create.

4.2.5 Drawing text with Image::Magick

With *Image::Magick* you have a choice of several font specifications and sources (see also "Fonts," on page 278). You can use a qualified X11 font name such as -*-times-medium-r-*-*-12-*-*-*-p-*-*-*, a PostScript name like Helvetica, or a TrueType font from a file with @arialbd.

The following are possible ways in which we can draw text with *Image::Magick*:

```
$im->Read('xc:white')

$rc = $im->Annotate(
    x    => 10,
    y    => 20,
    text => 'Some String');

$rc = $im->Annotate(
    text      => 'Green Text',
    font      => '@timesbd.ttf',
    fill      => 'green',
    geometry  => '+20+30',
    pointsize => 8);

$rc = $im->Draw(
    primitive => 'Text',
    points    => '0,40 "Other String"');

$rc = $im->Read('label:New Image');
```

❶ **Using Annotate() to draw text**

❷ **Using Draw() with the Text primitive**

❸ **Using the built in LABEL image type**

❶ `Annotate()` is used in two ways in this code. The first call uses the default font and colors to draw a string, and the coordinates are specified separately. The second call customizes almost everything that can be customized, and uses the geometry attribute to specify the coordinates. `Annotate()` is described in more detail on page 252.

❷ As you can see, drawing text with `Draw()` is quite awkward. The coordinates and the contents of the string have to be passed as a single parameter as the value of the points attribute. This can easily cause your code to look ugly when the coordinates and the text are all variables, and the quotes and escaping of quotes become messy. Splitting up your string and concatenating it, or using the `qq()` operator instead of quotes can help to keep it all readable. The `Draw()` method is described in further detail on page 257.

❸ The last line demonstrates the use of `Read()` with the built-in LABEL image type, to create an image with text only. This can be used to approximately predict the size of the string to be drawn, before we actually draw it on an image. We will see more on this in section 11.1, "Determining text size," on page 191.

SEE ALSO *Image::Magick* is described in quite some detail in appendix A, and it would probably be a good idea to browse through it to familiarize yourself with its functionality. The `Draw()` method is discussed in the section, "Draw()," on page 257, and the path primitive in section A.7, "The path drawing primitive," on page 281. The global image options are discussed in section A.4, "Image attributes," on page 246, and acceptable color and font names are described in "Common method arguments," on page 276. More information on combining images with *Image::Magick* can be found in section 8.2.2, "Combining Image::Magick images," on page 142.

4.3 COMBINING GD AND IMAGE::MAGICK

There might be a time when you would like to use *GD* to draw some objects, then transport your image to *Image::Magick*, manipulate it a bit, maybe draw some more, and then transport it back. One good reason to do this is to make use of the many image filters and special effects that *Image::Magick* provides, while benefitting from the higher speed of drawing in *GD*. Also, it might be that you have all this legacy code lying around that uses *GD*, and now you need to do something that this module doesn't allow, but *Image::Magick* does. Instead of rewriting all your code, you can use the techniques presented in this section to allow you to keep using all your legacy code, at least until you find the time to fully rewrite it all.[4] Of course, there is a computational cost associated with transporting the image back and forth between the two modules, so you will have to take that into consideration before you decide to go this way.

One way of transporting the image data between the modules is to save the image to a file with one module, and read it with the other. However, this is terribly inefficient and unnecessary. Since version 1.30, the *GD::Image*'s `new()` method accepts more arguments, and it is now possible to pass in the raw data of the image file as a scalar. Together with the `ImageToBlob()` and `BlobToImage()` methods that *Image::Magick* provides, we have a perfect mechanism to transport images back and forth between the two modules. The following example code illustrates how to transport an image in both directions:

```
use GD;
use Image::Magick;

my $gd = GD::Image->new(400,100);
my $white  = $gd->colorAllocate(255, 255, 255);
my $red    = $gd->colorAllocate(255,   0,   0);
my $blue   = $gd->colorAllocate(  0,   0, 255);
my $yellow = $gd->colorAllocate(255, 255,   0);

$gd->filledRectangle(49, 2, 349, 97, $yellow);
$gd->rectangle      (49, 2, 349, 97, $blue);
$gd->stringTTF($red,
    '/usr/share/fonts/ttfonts/arialbd.ttf',
    30, 0, 74, 64, 'This is a flag');

my $rc;
my $im = Image::Magick->new();
$rc = $im->BlobToImage($gd->png);
die $rc if $rc;

$rc = $im->Wave(amplitude => 12);
warn $rc if $rc;
```

❶ **Transport the image from GD to Image::Magick**

● **Make the rectangle look like a wavy flag**

[4] As if *that* is ever going to happen...

```
$rc = $im->Quantize(colors => 256);
warn $rc if $rc;
$gd = GD::Image->new($im->ImageToBlob());

$red = $gd->colorClosest(255, 0, 0);
my ($w, $h) = $gd->getBounds();
$gd->rectangle(0, 0, $w - 1, $h - 1, $red);
```

❷ **Transport the image back to GD**

❸ **Draw a rectangle around the edge of the image**

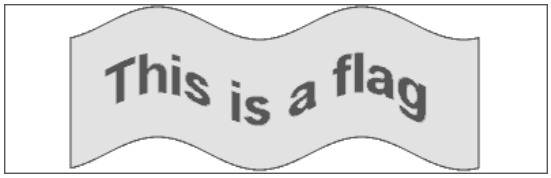


Figure 4.8
An image created with a combination of *GD* and *Image::Magick*. The flag was drawn with *GD*, the wave effect applied with *Image::Magick*, and the border drawn again with *GD*.

In this case, the whole figure could have been created with the *Image::Magick* object directly. But the point is that there might be situations, as discussed earlier, in which you have good reason to do it this way.

❶ To export the image from *GD* and import it into *Image::Magick*, the output from the `png()` method is directly fed into the `BlobToImage()` method. In real production code it might not be a bad idea to make this into two separate steps, and to verify in between that everything went as planned.

NOTE While *Image::Magick* is designed to preserve as much as possible from any image that gets imported, this is certainly not true for *GD*. The latter is limited to working with a 256 color palette (but also see the note on page 40), of which only one can be marked as transparent. *Image::Magick* supports full RGB color as well as an 8-bit alpha channel for transparency; hence, if you export an image from *Image::Magick* to *GD,* you run the risk of losing some information. Whether that is acceptable depends on the image, and on what you are prepared to sacrifice.

❷ *GD*'s limited palette is the reason the `Quantize()` method is called before exporting the image data back to *GD* from *Image::Magick*. We make certain that the number of colors introduced in the `Wave()` operation due to anti-aliasing doesn't exceed the 256 with which *GD* can comfortably deal. If you don't do this, the output of `BlobToImage` might be a truecolor PNG format, and *GD* doesn't deal with that very well. We hand the output of `ImageToBlob()` directly to *GD*'s `new()` method. This is doable, as the image type of the `$im` object will default to PNG, since that is also what it read in. Thus we know that *GD* will be able to read it.

❸ Once the image is back in *GD* we get the closest color to a pure red, and draw a rectangular frame around the edges of the image. This is done mainly to show that it is actually possible. There is, of course, no guarantee that the color we will be using is actually pure red, since we use the `colorClosest()` call, but that is the price we pay for working with a small palette.

Once we're finished working with `$gd`, we can save the image, print it, or even pass it back to *Image::Magick* again. The point is that it is easy to do this, although it is also computationally slightly expensive. Use this only if you have an absolutely genuine need to do so.

4.4 DRAWING WITH TERM::GNUPLOT

The Perl module *Term::Gnuplot* provides an interface to the drawing routines and terminal drivers that are used internally by gnuplot (see section 5.4, "Interfacing with gnuplot," on page 85). Because these routines have been specifically designed and written for gnuplot, they are also strictly limited to the functionality required by this charting program, which means they are *very* limited. But even with the few primitives that *Term::Gnuplot* provides, one can do surprisingly nice things.

One feature that makes *Term::Gnuplot* worth looking at is the large number of output devices that it supports, many of which are vector based, making it feasible to create graphics with this module that scales a lot better than do images. As said, the main reason for these terminal drivers is to supply gnuplot internally with a drawing interface that is consistent for all output. However, paraphrasing an old saying: all output terminals are equal, but some are more equal than others; i.e., that, while all terminal drivers provide the identical interface, not all terminals have the same capabilities. The actual results cannot always be as finely controlled as one would like to see in a generic drawing package, as the interface is written to a lowest common denominator, and sometimes at a fairly abstract high level.

For example, not all terminals support color, so there is no direct way to manipulate the color of the line you're drawing. Instead there are generic line types (set with `line_type()`). How exactly a particular line type gets drawn is up to the driver, and the driver alone. Font support is another feature that differs markedly between the various terminals. Some, such as the PostScript one, support several built-in fonts, others support only a very small list.

In practice, unless you can make your drawing very generic, you shouldn't be using this module.

In part, to steer past some of the problems that a totally generic drawing interface introduces, each terminal device in *Term::Gnuplot* has its own characteristics and options that can be set with the `set_options()` function. What exactly those options do, and how they affect the outcome of the drawing, is up to the driver. All terminal drivers, and their options, are documented in the *Term::GnuplotTerminals* documentation, as well as in the internal help of the gnuplot program.

Let's write something that makes *Term::Gnuplot* do what it does best: a program that plots a function.[5] The skeleton of the program looks like this:

[5] This example borrows heavily from Ilya Zakharevich's example in the *Term::Gnuplot* documentation.

```
use Term::Gnuplot ':ALL';

plot_outfile_set('TermGnuplot.eps');
change_term('postscript');
set_options('eps', 'enhanced', '"Helvetica"',  24);
term_init();
term_start_plot();

draw_axes(-6, 6, -1, 1, 0.1);
plot_function(sub {sin $_[0]/2});
plot_function(sub {cos $_[0] * sin $_[0]});
plot_function(sub {-0.2 * $_[0]}, 2);

term_end_plot();
Term::Gnuplot::reset;
```

❶ Set up and initialize the terminal

❶ Plot some functions

❷ And end the output

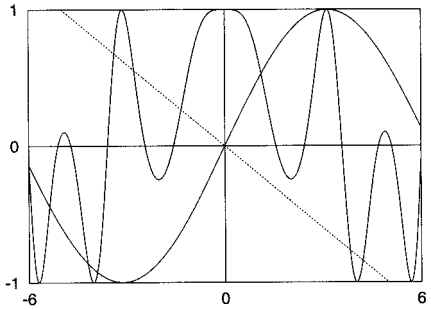


Figure 4.9
The drawing capabilities of *Term::Gnuplot* lend themselves well for plotting functions, which is not surprising, given that the module is an interface to the drawing library of gnuplot.

❶ There are a few things that need to be set up before we can start writing the code that actually creates the drawing. First, we need to initiate and open an output device with `plot_outfile_set()`, `change_term()` and `set_options()`. These three lines are the only output-specific code in the program. For this particular example we'll create an EPS output file, and initiate the font to a 24-point Helvetica. Once that is done, the terminal can be initialized and the plot started.

❷ After all the drawing is completed, we need to call `term_end_plot()` to end our plot, and `Term::Gnuplot::reset()`[6] to free all resources that were allocated with `term_init()`. Calling `reset()` is optional if you are going to exit from the program anyway, but it's a good idea to get into the habit of making it part of your code.

We'll now have a look at the function that draws the axes of the plot, and sets up some variables needed for the plotting functions.

```
my ($h_offset, $v_offset);
my ($delta_x,  $delta_y);
my ($xmin, $xmax, $ymin, $ymax);
```

❶ Some variables used by draw_axes() and plot_function()

[6] We use the fully qualified name of the subroutine here, because the name `reset()` is already taken by a Perl built-in function.

```
sub draw_axes
{
    my ($margin);
    ($xmin, $xmax, $ymin, $ymax, $margin) = @_;
    my ($h_points, $v_points);

    $h_offset = $v_offset = 0;
    $h_points = scaled_xmax() - 1;
    $v_points = scaled_ymax() - 1;

    if ($margin)
    {
        $h_offset  = $margin * $h_points;
        $v_offset  = $margin * $v_points;
        $h_points -= 2 * $h_offset;
        $v_points -= 2 * $v_offset;
    }

    linetype -2;
    linewidth 2;
    move  ($h_offset,             $v_offset);
    vector($h_offset + $h_points, $v_offset);
    vector($h_offset + $h_points, $v_offset + $v_points);
    vector($h_offset            , $v_offset + $v_points);
    vector($h_offset            , $v_offset);

    justify_text(RIGHT);
    put_text($h_offset - h_char(), $v_offset, $ymin);
    put_text($h_offset - h_char(), $v_offset + $v_points, $ymax);
    justify_text(CENTRE);
    put_text($h_offset, $v_offset - v_char(), $xmin);
    put_text($h_offset + $h_points, $v_offset - v_char(), $xmax);

    $delta_x  = ($xmax - $xmin)/$h_points;
    $delta_y  = ($ymax - $ymin)/$v_points;

    if ($xmin < 0 && $xmax > 0)
    {
        my $h_zero = $h_offset - $xmin/$delta_x;
        move  ($h_zero, $v_offset);
        vector($h_zero, $v_offset + $v_points);
        justify_text(CENTRE);
        put_text($h_zero, $v_offset - v_char(), "0");
    }
    if ($ymin < 0 && $ymax > 0)
    {
        my $v_zero = $v_offset - $ymin/$delta_y;
        move  ($h_offset,             $v_zero);
        vector($h_offset + $h_points, $v_zero);
        justify_text(RIGHT);
        put_text($h_offset - h_char(), $v_zero, "0");
    }
}
```

❷ **Set up the boundaries of the plot**

❸ **Draw the box**

❹ **Draw the axes**

❶ The `draw_axes()` subroutine initializes the coordinates for the plot relative to the actual coordinates of the medium we're working with, and draws the axes. It accepts 5 arguments: The minimum and maximum *x* value to plot, the minimum and maximum *y* values to plot and the amount of space to leave open for margins. This last parameter is relative to the total size, i.e., a value of 0.1 will leave a 10 percent margin all around the plot.

❷ The subroutine starts off by asking the terminal it is going to write to how many horizontal and vertical points it has. The plotting code will use those numbers for the resolution with which the functions are going to be plotted.

❸ Once we have calculated the resolution of the plot, it is time to draw a box. We set the line type and width, and move the pen to the lower left corner of the box in which the plot will appear. We then draw the box with the four `vector()` calls. These calls are equivalent to a `lineto` statement in PostScript or SVG paths; they draw a straight line. To complete the box, we put the requested minimum and maximum values along the axes. If the *x* or *y* values to be plotted include 0, we also plot the zero point axes, and label them as such.

❹ The axes inside the plot are only drawn when necessary; if the zero point doesn't fall between the minimum and maximum axis on the value, we don't need to draw anything.

The subroutine that plots the functions is:

```
sub plot_function
{
    my $function   = shift;
    my $line_type  = shift || 0;
    my $line_width = shift || 4;

    linetype $line_type;
    my $moved = 0;

    for (my $x = $xmin; $x <= $xmax; $x += $delta_x)
    {
        my $y = $function->($x);
        next unless $y >= $ymin && $y <= $ymax;
        my $hor = $h_offset + ($x - $xmin)/$delta_x;
        my $ver = $v_offset + ($y - $ymin)/$delta_y;
        ($moved) &&
            vector($hor, $ver) ||
            move  ($hor, $ver), $moved = 1;
    }
}
```

The `plot_function()` subroutine is fairly simple. It accepts as its first argument a reference to a subroutine that can be used to calculate the *y* value, given an *x* value, and a line type and width as the optional second and third arguments. To determine the resolution and boundaries of the plot, it uses the values that were previously calculated and set by `draw_axes()`.

The output of this code can be seen in figure 4.9. As you can see, the result is surprisingly good-looking for a relatively simple program.

Using *Term::Gnuplot* can seem daunting at first, but once you have built yourself a few subroutines such as the ones just described, it becomes easier to deal with. All you need to do is write some wrapper functions that allow you to draw some circles, squares, and other shapes, and maybe some fill patterns. *Term::Gnuplot* will never take the place of *GD* or *Image::Magick*, but it does provide a low-level drawing library that is capable of creating various vector format graphics.

SEE ALSO The documentation for *Term::Gnuplot* is fairly minimal at this time; however, it provides enough information to work with the modules with sufficient study. The terminals and output devices that are supported by this module are documented in a separate manual page: *Term::GnuplotTerminals*.

4.5 POSTSCRIPT AND SVG

Most of this chapter has focused on creating image graphics with *GD* and *Image::Magick*. *Term::Gnuplot* offers drivers that allow you to create some vector formats, but its set of drawing primitives is rather limited. There are, however, other possibilities to create vector graphics.

Perl is a text processing language. Postscript and SVG, for example, are both text-based formats. Even better, SVG is based on XML. Thus you can easily create output files in these formats. Both PostScript and SVG are vector formats, or at least, both support vector graphics as part of their native format.

To describe the PostScript language or Scalable Vector Graphics format in this book is not really possible, since they both are quite large and extensive. However, if you want to bone up on PostScript, you could read the PostScript language reference [21], and for SVG I'd recommend the specifications on the Web Consortium website [22]. Additionally, take a look at the PostScript and SVG implementations of a module discussed in chapter 10, and which are listed in appendix C.

4.6 SUMMARY

In this chapter we've looked at the various modules that are available for Perl to create two-dimensional computer graphics. We've seen that *GD* is simple to use and reasonably featureful, but is limited in the number of colors it supports, and lacks some higher-level functionality. *Image::Magick* offers many more features, but is more difficult to use, and the code needed to create drawings can become quite verbose. However, it supports a wide variety of output formats and a good set of drawing commands. We've also seen that *GD* and *Image::Magick* can be combined to attain the best of both worlds.

If you need to produce graphics that are not bitmaps, Term::Gnuplot is a possibility, although its interface is limited. Alternatively, you can directly create PostScript or SVG files.