Ordering Information:

Perl How to Program
The Complete Perl Training Course

- View the complete **Table of Contents**
- Read the **Preface**
- Download the **Code Examples**

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at **www.informIT.com/deitel**.

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ONLINE* e-mail newsletter at **www.deitel.com/newsletter/subscribeinformIT.html**.

To learn more about our **Perl programming courses** or any other Deitel instructor-led corporate training courses that can be delivered at your location, visit **www.deitel.com/training**, contact our Director of Corporate Training Programs at (978) 461-5880 or e-mail: **christi.kelsey@deitel.com.**

Note from the Authors: This article is an excerpt from Chapter 21, Sections 21.1 and 21.2 of *Perl How to Program*. This article introduces basic graphics processing in Perl, used to create GUIs, charts and images. We provide an example of creating an image with several shapes.Readers should be familiar with Perl syntax, as well as objects and modules in Perl. The code examples included in this article show readers programming examples using the DEITEL™ signature LIVE-CODE™ Approach, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

21.1 Introduction

Graphics convey information and make programs visually appealing. Everywhere we look, we see graphics—video games, billboards, movies, etc. One of the best examples is World Wide Web pages. The majority of Web pages have some form of graphics. Pictures are one form of graphics commonly found on Web pages. Pictures provide an enormous quantity of information. Graphics are more than just pictures; they are the elements of a picture. Colors, lines, rectangles, patterns, text, etc. are all graphics. Graphics are visual images.

Perl does not have any built-in graphics manipulation functions, yet we can use Perl to generate complex graphics and *graphical user interfaces* (also called *GUIs*). Many modules have been developed to offer these capabilities. We will look at three of the most popular modules in this chapter. We begin to understand basic graphics concepts by discussing the **GD** module. From there, we move on to discuss modules that create more high-end graphics, charts and sophisticated GUIs.

21.2 GD Module: Creating Simple Shapes

The ***GD*** module is used to create and manipulate images, similar to a basic drawing or painting application. The major difference between the **GD** module and drawing applications is that the programmer must specify in text commands, rather than physically drawing, what it is that he or she wishes to accomplish. Figure 21.1 demonstrates a simple program that creates some shapes.

```
1  #!/usr/bin/perl
2  # Fig 21.1: fig21_01.pl
3  # Using the GD module to create shapes.
4
5  use strict;
6  use warnings;
7  use GD;
8
9  my $image = new GD::Image( 320, 320 );
10
11 my $white = $image->colorAllocate( 255, 255, 255 );
12 my $red = $image->colorAllocate( 255, 0, 0 );
13 my $green = $image->colorAllocate( 0, 255, 0 );
14 my $blue = $image->colorAllocate( 0, 0, 255 );
15 my $black = $image->colorAllocate( 0, 0, 0 );
16 my $purple = $image->colorAllocate( 255, 0, 255 );
17
18 $image->filledRectangle( 15, 15, 150, 150, $red );
19 $image->arc( 200, 200, 50, 50, 0, 360, $black );
20 $image->fill( 200, 200, $blue );
21 $image->rectangle( 100, 100, 200, 125, $green );
22 $image->fillToBorder( 150, 110, $green, $green );
23
24 my $polygon = new GD::Polygon();
25 $polygon->addPt( 20, 300 );
26 $polygon->addPt( 20, 175 );
```

Fig. 21.1 Using the **GD** module. (part 1 of 2)

```
27  $polygon->addPt( 100, 175 );
28
29  $image->polygon( $polygon, $blue );
30  $image->fill( 50, 200, $purple );
31
32  $polygon->setPt( 0, 30, 300 );
33  $polygon->setPt( 1, 110, 300 );
34  $polygon->setPt( 2, 110, 175 );
35
36  $image->filledPolygon( $polygon, $black );
37
38  open( PICT, ">fig21_02.png" ) or
39     die( "Can not open picture: $!" );
40
41  binmode( PICT );
42  print( PICT $image->png() );
43  close( PICT ) or die( "Can not close file: $!" );
```

Fig. 21.1 Using the **GD** module. (part 2 of 2)

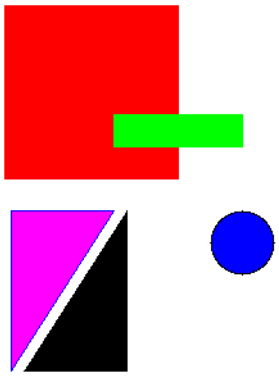

Fig. 21.2 Contents of **fig21_02.png**.

The program uses many of the **GD** module's methods, so let us go through it line by line. Line 7 imports the **GD** module. Line 9 creates a new **GD** image object. We pass to the constructor the *width* and *height* of the image we wish to create (in this case, 320 pixels wide by 320 pixels high). The upper-left corner of this area has an *x coordinate* of 0 and a *y coordinate* of 0. The *x* coordinate increases from left to right across the image and the *y* coordinate increases from top to bottom in the image. Then we define some colors (lines 11–16). The color **$white** becomes the background color for our picture. We define this color using the ***colorAllocate*** method, passing it the red, green and blue values that compose the color. The **GD** module uses the *RGB* (*red, green, blue*) *color scheme*, in which the red, green and blue parts of the color are specified as integers in the range 0–255. We also define colors red, green, blue, black and purple and set them to the corresponding variables. We now have a palette and we can refer to any of those six colors by their variable names.

Next we create some predefined polygons. We use the **$image** object's ***filledRectangle*** method to create a filled rectangle (line 18). The first two arguments define one corner of the rectangle. The third and fourth arguments define the diagonally opposite corner. The fifth argument is the color. In this case, the whole rectangle will be red.

We create a circle using method ***arc*** (line 19). The first two arguments passed to **arc** are the coordinates of the center of the circle. The next two arguments are the horizontal and vertical radius. The next two arguments provide degree bounds for the sweep of the arc. In this case, the bounds are 0 and 360 so this will sweep out the full circle. The final argument is the color. Notice that this method will not fill in the arc. We are forced to do this manually on line 20 by calling **GD**'s ***fill*** method. The first two arguments of this method specify a pixel. The paint will spread out from this pixel until a pixel is found that is not the same color as the starting pixel. The third argument is the color of the paint.

On line 21, we use **GD**'s ***rectangle*** method. This method to creates a hollow rectangle. We would like to fill this rectangle with color manually. We could use our **fill** method to do so, but half of the rectangle we just created is red (it overlaps with the previous rectangle) and the other half is white (blank canvas). The **fill** method fills only until it hits a pixel of a different color than the previous one. If we started filling in the white portion, only that area would get colored. Likewise, if we started filling in the red portion, only that part would be colored. **GD** provides a method to work around this problem called ***fillToBorder***. This method takes the same arguments as **fill**, but adds one more argument before the fill color. The third argument in this method is the border-color argument. The method will fill in the designated area until it reaches a pixel that is colored with the specified border color. In this case, we fill in the inside of the rectangle in green until we hit a border of green, the outside of the rectangle.

Next, we explore **GD**'s ***Polygon*** class. On line 24, we create an instance of the **Polygon** class. The polygon is null—it has no points. Then, we create some points in this polygon using method ***addPt***. This method takes two arguments—the *x* coordinate and the *y* coordinate of the point with respect to the upper-left *(0, 0)* coordinate of the image. We add three points to the polygon, then we add the polygon to the image using method ***polygon*** (line 29). This method takes a polygon object and a border color and adds the polygon to the image. Next, we change the points in our polygon so that we can draw the polygon again in a different position. We use **Polygon**'s ***setPt*** method to do this. This method takes three arguments. The last two arguments are the *x* and *y* coordinates of the new point. The first argument is the number of the point we wish to change. It is important to note that this numbering starts at zero. Once we have changed our polygon, we add it to our image, this time using method ***fillPolygon***. This method is draws a polygon filled with a color.

Finally, we would like to output our picture object to a file for later viewing. So, in line 38, we open a file. Line 41 sets the output mode of the file to binary with function ***binmode***, so that when the image is stored, the bytes of memory containing the image are written in the image's native format, rather than a platform-specific format. Then, we output the image after converting it to a ***.png*** (*Portable Network Graphics*) *file* with method ***png***. Figure 21.2 contains the image that was created in Fig. 21.1.