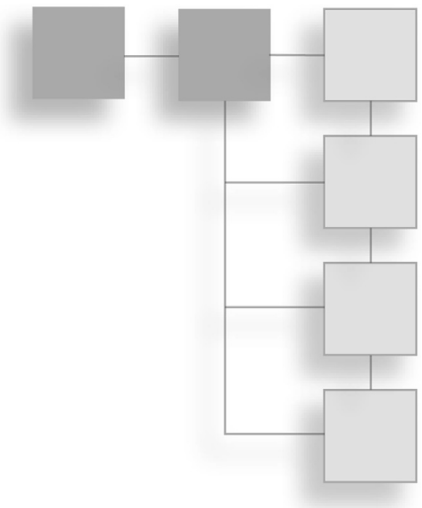
CHAPTER 8



GD Graphics Overview

- What Is GD?
- Installing GD
- Creating an Image
- Drawing Shapes on the Canvas
- Manipulating Color Information
- Adding Text to Images
- Resizing Images

Boutell's GD library is a great tool that provides you with ways to manipulate graphics on the fly. In this chapter you will be learning just that. You will start off by learning how to create images and draw simple shapes on the canvas. Then you will learn how to manipulate the color information of an image, add text to an image, and, finally, how to resize an image. This knowledge will allow you to add cool graphics on the fly to your on-line game.

What Is GD?

GD is a C graphics library that allows you to create and manipulate .jpegs, .pngs, and .wbmps. What's that you say? No support for .gifs? That is correct. There used to be a GD library out there at one time that supported the .gif file format, but since Unisys (the company that owns the patent on LWZ compression that .gifs use) changed their licensing agreements, GD has stopped supporting the .gif file format. This could change in the next

year, because Boutell (the company that makes the GD library) is planning to support .gifs as soon as the patent expires. Meanwhile, have no fear because .pngs support all of the features of a .gif (as was discussed in Chapter 3).

Installing GD

Just in case you missed installing the GD library when you were setting up PHP, I'll quickly go over it again. I will also show you a great little chunk of code that you can use to load the library dynamically.

To install GD on a Linux or UNIX installation you will need to recompile PHP with the --with-gd option. You can also compile the GD library as a shared object so you can load the GD library dynamically whenever you need it. To do this you would use the --with-gd=shared option. Then, to load in the library dynamically at run time, you would use the following line of code:

```
dl('gd.so');
```

To install GD on a Windows platform you need to copy the php_gd2.dll that came with the installation package into the extensions directory specified in the php.ini file. Then you will need to edit the php.ini file and uncomment the following line:

```
extension=php_gd2.dll
```

This will enable access to the GD library. If you would like to load the extension dynamically with script you may do so with the following line of code:

```
dl('php_gd2.dll');
```

Caution

In order to dynamically load an extension, the shared object file (UNIX) or the dll file (Windows) must be in the extensions directory specified in the php.ini file. If it is not, then the loading of the extension will fail and none of the functions will be available to your game.

Let's say you don't know if the server you are hosting your files on is a UNIX or a Windows server. You can create a dynamically loading script that will work on either platform very easily. Just take a look at the following code:

```
<?php
if(!extension_loaded('gd'))
{
    if(strtoupper(substr(PHP_OS, 3)) == "WIN")
    {
        dl('php_gd2.dll');
```

```
    }
    else
    {
        dl('gd.so');
    }
}
?>
```

This handy little chunk of code will work for any extension—this example just happens to be for the GD library. The first step is to see if the extension is already loaded. If the extension is loaded then why would you want to do anything else? If, however, the extension is not loaded then you will need to determine what platform you are on and load the proper file. To determine what OS the PHP script is living on you will need to use the PHP_OS constant. This returns the information about the current OS. If you are on a Windows platform the first three characters of the string will be "WIN". If you do not find the string "WIN" in the PHP_OS constant it is safe to assume that you are on a UNIX platform. Once you know what platform you are on you can dynamically load the extension that you want by using the `dl()` function.

Now that you have the extension properly loaded it is time to test to see if it is actually there. The easiest way to do this is to use the `phpinfo()` function. Remember when you used this in Chapter 2? You will need to run that same test script again and look for a section in the page that looks like Figure 8.1.

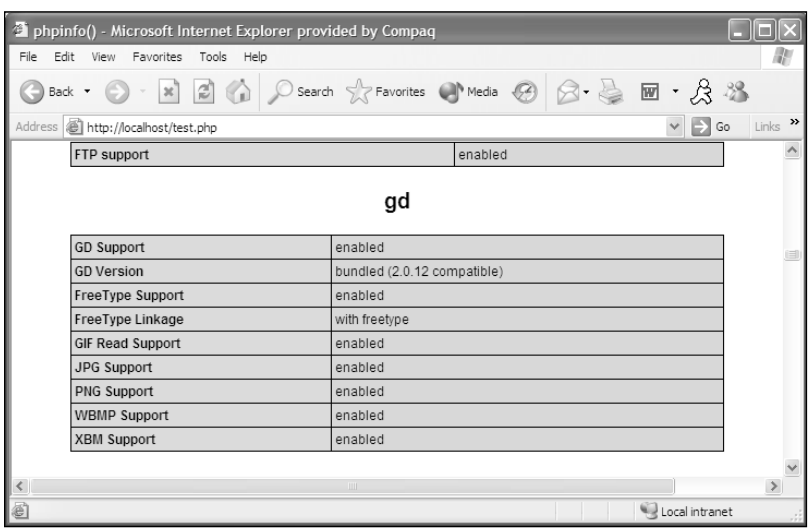


Figure 8.1 Results of the `phpinfo()` function.

Creating and Using a New Image

To create new images, GD offers you two functions. Both of these functions take two arguments, width and height, and they both return a resource to the image.

- `ImageCreate(width, height)`
- `ImageCreateTrueColor(width, height)`

GD also offers functions to create images from existing files, but you will learn about these later in this chapter. First take a look at the `ImageCreate()` function.

Note

A resource to an image is just the chunk of memory in which the image is being stored. In PHP you work in memory to manipulate images instead of a GUI like Paint Shop Pro.

`ImageCreate()` creates a new blank image with width number of pixels across, and height number of pixels tall. The color palette that `ImageCreate()` uses when creating an image is an indexed-color palette, meaning that the number of colors in the palette will be 256.

Remember in Chapter 3 when image types and compressions were discussed? The indexed-color palette uses a color lookup table to determine the specific color. These types of images are really good for images that use flat colors or text and they are especially good with simple shapes. You can save these images as .png files, and if GD ever supports the .gif file format again you will be able to save them as .gif files.

Caution

If you save an 8-bit image (also referred to as an indexed-color palette image) as a .jpeg you will end up with quite large file sizes and blotchy images.

Take a look at an example for creating a blank image:

```
<?php
if(!extension_loaded('gd'))
{
    if(strtoupper(substr(PHP_OS, 3)) == "WIN")
    {
        dl('php_gd2.dll');
    }
    else
    {
        dl('gd.so');
```

```
    }
}
$imageResource = ImageCreate(100, 100);
?>
```

You now have an image in memory that is 100 pixels wide and 100 pixels tall. You can now operate on this image using various functions provided by GD. Before you get into these functions, though, take a very quick look at the `ImageCreateTrueColor()` function.

The `ImageCreateTrueColor()` function also takes a width and height and it also returns a resource to an image in memory. However, it has one key difference: it creates an image with a true-color palette instead of an indexed-color palette. This makes this function ideal for creating complex graphics with lots of different elements in them. You would use this function to create .jpeg images. You can also save the image as a .png, but it will be saved as a true-color .png file instead of an indexed-color .png file.

Now that you know how to create an image resource, you will want to know how to edit this resource.

How to Use Colors

Before you can actually start drawing geometric shapes all over the place, you need the ability to select and use color. There are eight basic functions for color, as follows:

- `ImageColorAllocate(resource image, int red, int green, int blue)`
- `ImageFill(resource image, int x, int y, int color)`
- `ImageColorTransparent(resource image, int color)`
- `ImageTrueColorToPalette(resource image, bool dither, int colors)`
- `ImageColorsTotal(resource image)`
- `ImageColorAt(resource image, int x, int y)`
- `ImageColorsForIndex(resource image, int index)`
- `ImageColorSet(resource image, int index, int red, int green, int blue)`

Each of these functions takes an image resource as its first argument. Now take a more in-depth look at each of these functions to see how you will be able to use them to manipulate the colors on your canvas.

Allocating Colors to an Image

`ImageColorAllocate()` takes four arguments: an image resource, a value for the amount of red that should be in the color, a value for the amount of green that should be in the color,

and a value for the amount of blue that should be in the color. If you have never used RGB (red, green, blue) values before, each element of the color has a range of 0 to 255; 0 being the lowest level and 255 being the highest level. Let's say you wanted to allocate a color that was completely black. You would write a line of code that looks like the following:

```
$black = ImageColorAllocate($someImageResource, 0, 0, 0);
```

What this essentially means is that you will be able to use the variable `$black` as a color in the specified image resource. If you were working with two images simultaneously, then you would not be able to use the variable `$black` in both images. You can use only the colors that you allocate to a specific image. If this seems a little confusing, take a look at the following code example and hopefully it will make things a little clearer:

```
<?php
// Create two images
$image1 = ImageCreate(100, 100);
$image2 = ImageCreate(100, 200);
// Allocate the colors to be used
$black = ImageColorAllocate($image1, 0, 0, 0);
// This is legal to do
ImageRectangle($image1, 6, 6, 66, 42, $black);
// This is illegal to do because black is not allocated to the image
ImageRectangle($image2, 6, 6, 66, 42, $black);
?>
```

Filling the Image

The `ImageFill()` function also takes four arguments: an image resource, a starting x point, a starting y point, and the color to fill. `ImageFill()` does a flood fill of the entire image starting at the specified coordinates. Since this function fills the entire image there is no need to specify any other coordinate besides (0, 0). Take a look at the following example:

```
<?php
$image = ImageCreate(100, 100);
$red = ImageColorAllocate($image, 255, 0, 0);
ImageFill($image, 0, 0, $red);

// Show our Image
header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

Take a look at Figure 8.2 to see the results of using the `ImageFill()` function.

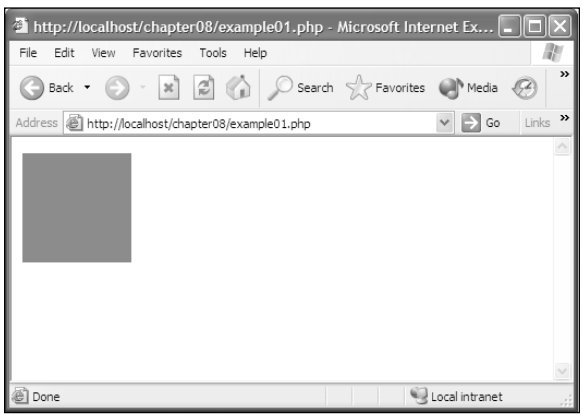


Figure 8.2 Results of using the `ImageFill()` function.

You could have specified any (x, y) coordinate and you would have received the same exact results. Don't worry too much about the display code right now; it will be covered later in this chapter.

Setting Your Transparent Color

If you are using an image format that supports transparencies, such as .png, you can set the color you want to show up as transparent by using the `ImageColorTransparent()` function (Figure 8.3). `ImageColorTransparent()` takes two arguments: the image resource, and the color that you want to make transparent. To allocate a color to be transparent you still use the `ImageColorAllocate()` function. To understand this better, take a look at the code below. In the following example a red circle is drawn over a dark background and then the color red is made transparent and the same image is drawn again.

```
<?php
$image = ImageCreate(128, 128);
$black = ImageColorAllocate($image, 0, 0, 0);
$red = ImageColorAllocate($image, 255, 0, 0);
// Make the background black
ImageFill($image, 0, 0, $black);
// Draw the circle
ImageFilledArc($image, 64, 64, 110, 110, 0, 360, $red, IMG_ARC_PIE);

// Show our Image Filled Image
ImagePng($image, "redcircle.png");

// Make the red transparent
ImageColorTransparent($image, $red);
```

```
// Show our Image Transparent Image
ImagePng($image, "transparentcircle.png");

ImageDestroy($image);
?>
<HTML>
<HEAD>
    <TITLE>Image Test</TITLE>
</HEAD>
<BODY>
<img src="redcircle.png">
<img src="transparentcircle.png">
</BODY>
</HTML>
```

Caution

You will need to change the permissions on the folder containing redcircle.png in order to use the example. Simply add read/write permissions to the IUS_(MACHINENAME) user under the security tab for the folder.

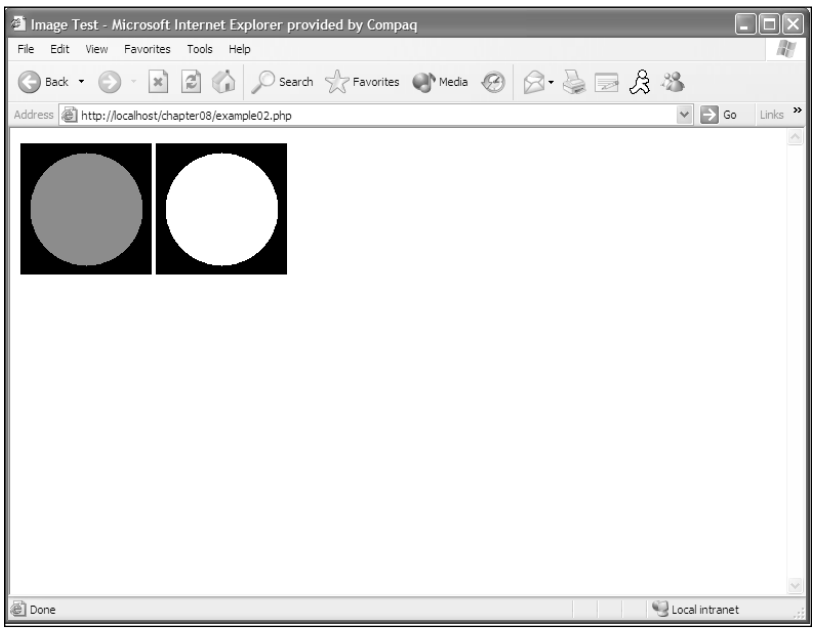


Figure 8.3 Using transparent colors.

How cool is that!? You are quickly becoming a graphic expert in PHP. Again, don't worry about the display code yet; I promise you will get to it later in this chapter.

Converting a True-Color Image to a Palette Image

Converting an image is not as complicated as it sounds. As a matter of fact, this is very easy using the `ImageTrueColorToPalette()` function. `ImageTrueColorToPalette()` takes three arguments, the first of which is the image resource. The second is a Boolean value—if it is set to true, then dithering will be used; otherwise dithering will not be used. The third argument is the number of colors that should appear in the color palette of the new image.

You could use this function to convert a .jpeg image into an indexed-color image, but the decline in quality of the image would be very noticeable. The most logical reason to use this function would be if you have a .png image that uses only, say, 16 colors, but it was saved as a true-color image for some odd reason. Take a look at the following example:

```
<?php
// Get the image that needs to be converted
$image = ImageCreateFromPng('truecolorimage.png');
// Now convert the image, since it is only using 16 colors you only need to specify 16
colors
ImageTrueColorToPalette($image, true, 16);
// save the image
ImagePng($image, 'convertedimage.png');
?>
```

See, you converted an existing image from a true-color image to an indexed-color image and saved it as a new image all in three lines of code. I told you it wasn't as complicated as it sounded. Do some experimenting with this function on several different images to see the effects. Convert a .jpeg photo using dithering, and then do the same thing without using dithering to see the dramatic differences.

Counting Colors in an Image

To get the total number of colors in an image, you can use `ImageColorsTotal()`. `ImageColorsTotal()` takes one argument: the image resource for which you want to count the number of colors.

```
<?php
// get an Image
$image = ImageCreateFromPng('somegraphic.png');
// now count the colors in this Image
```

```
$totalColors = ImageColorsTotal($image);
echo("There are " . $totalColors . " colors in this image");
?>
```

Caution

The `ImageColorsTotal()` function works only with indexed-color images.

Retrieving a Color at a Point

The `ImageColorAt()` function takes three arguments: the image resource, an x point, and a y point. `ImageColorAt()` will return the index of the color at the specified (x, y) pixel. Once you have this index you can use the `ImageColorsForIndex()` function to retrieve the red, green, and blue components of the specific color.

```
<?php
$image = ImageCreateFromJpeg('someimage.jpg');
 $colorIndex = ImageColorAt($image, 100, 20);
$colorArray = ImageColorsForIndex($image, $colorIndex);
$red = $colorArray['red'];
$green = $colorArray['green'];
$blue = $colorArray['blue'];

echo("Red: " . $red . "<br>Green: " . $green . "<br>Blue:" . $blue);
?>
```

Caution

The `ImageColorAt()` function works only with true-color images.

The `ImageColorsForIndex()` function takes two arguments: an image resource and a color index. It then returns an array with the individual red, green, and blue components in each index. Once you have these components you could use the `ImageColorAllocate()` function to allocate a color for an image, or you could use the `ImageColorSet()` function to change a color in an image.

The `ImageColorSet()` function takes five arguments. The first is the image resource that you want to operate on, the second is the index of the color you want to change, the third argument is the red component, the fourth is the green component, and the fifth and final argument is the blue component.

Drawing Basic Shapes on Your Empty Canvas

You now know how to create an image canvas that you can draw on, and you also know how to allocate, manipulate, and count colors in an image. Now you'll learn how to draw geometric shapes on your empty canvas.

Tip

If you have ever worked with DirectX you can think of the image resource as a buffer that you draw on. When you display the image you can think of that as blitting the buffer to the screen. The only difference is that you don't do this per frame.

To draw anything on your images you need a coordinate system. GD uses a Cartesian coordinate system where the point (0, 0) is in the upper left-hand corner of the image. Moving to the right along the x axis will move you in the positive x direction. Moving down along the y axis will move you in the positive y direction. You will never use a negative x or y coordinate when working with images.

Each of the eight geometric functions available in GD takes one or more sets of coordinates. You will be reviewing each of the eight functions in detail, so dust off all of those old geometry lessons and get ready. Here are the eight geometric functions, in order from easiest to hardest:

- `ImageSetPixel(resource image, int x, int y, int color)`
- `ImageLine(resource image, int x1, int y1, int x2, int y2, int color)`
- `ImageRectangle(resource image, int x1, int y1, int x2, int y2, int color)`
- `ImageFilledRectangle(resource image, int x1, int y1, int x2, int y2, int color)`
- `ImagePolygon(resource image, array points, int numberOfPoints, int color)`
- `ImageFilledPolygon(resource image, array points, int numberOfPoints, int color)`
- `ImageArc(resource image, int centerXPoint, int centerYPoint, int width, int height, int startDegree, int endDegree, int color)`
- `ImageFilledArc(resource image, int centerXPoint, int centerYPoint, int width, int height, int startDegree, int endDegree, int color, int style)`

Pixels and Lines

`ImageSetPixel()` takes four arguments: the first is the image resource, the second is the x point, the third is the y point, and the fourth argument is the color. The `ImageSetPixel()` function draws a single pixel at point (x, y) in the specified color. Take a look at the following example:

```
<?php
$image = ImageCreate(320, 200);
```

```
$color = ImageColorAllocate($image, 254, 254, 254);
ImageFill($image, $color);

for($iLoop = 0; $iLoop < 1000; $iLoop++)
{
   $color = ImageColorAllocate($image, rand() % 256, rand() % 256, rand() % 256);
ImageSetPixel($image, rand() % 320, rand() % 200, $color);
}

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

This example generates a 320 × 200 image, setting 1000 random pixels in a random color throughout the canvas. You use the random function to generate a random number and then perform a modulus on that number to keep it between the specified limits of the canvas size. Take a look at Figure 8.4 to see the results of the code.

Now that you can draw pixels to the screen, let's deal with a function that is a little more practical. The `ImageLine()` function draws a line to the canvas. It takes six arguments. The first is the resource to the image to which you are drawing the line. The second and third arguments are the starting coordinates of the line. The fourth and fifth arguments are the ending coordinates of the line. The final argument is the color that you want the line to be drawn in.

```
<?php
$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, $white);

ImageLine($image, 5, 5, 40, 100, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

In this example, you first create an image that is 320 × 200 pixels wide. Then you allocate two colors, white and black. You fill the canvas with the white color so your canvas is now all white. Then you draw a line from point (5, 5) to point (40, 100) in black and output the image to the browser.

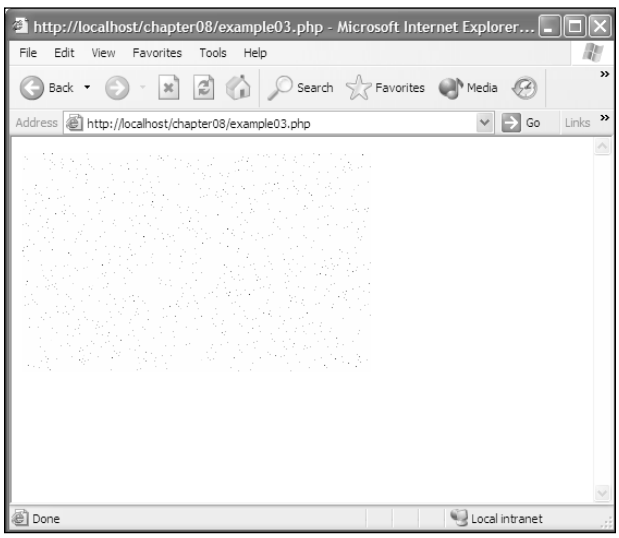


Figure 8.4 Generating 1000 random pixels.

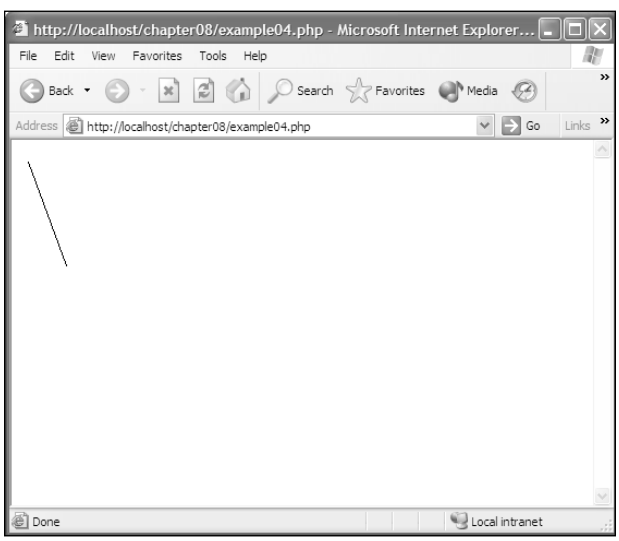


Figure 8.5 Drawing a line.

From Lines to Rectangles

`ImageRectangle()` takes the same four parameters as the `ImageLine()` function, except instead of the starting point of a line and an ending point for a line, it is the upper left-hand corner of the rectangle and the lower right-hand corner of the rectangle.

```
<?php
$image = ImageCreate(320, 200);
```

```
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, $white);

ImageRectangle($image, 5, 5, 100, 100, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The results of this example are shown in Figure 8.5.

But wouldn't it be better to give the rectangle a starting (x, y) coordinate and a width and a height that you would like the rectangle to be? You can create a function that does exactly that.

```
<?php
function MyRectangle($image, $x1, $y1, $width, $height, $color)
{
    // First you need to calculate the values for x2 and y2
    $x2 = $x1 + $width;
    $y2 = $y1 + $height;

    // Now draw the rectangle
    ImageRectangle($image, $x1, $y1, $x2, $y2, $color);
}

$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, $white);

MyRectangle($image, 5, 5, 95, 95, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The results of this example look like Figure 8.6, but the function is much more intuitive. I recommend that anytime you see a way to make a function simpler by creating your own, then do it. Anytime that you can make code that makes more sense when you read it, you should do it.

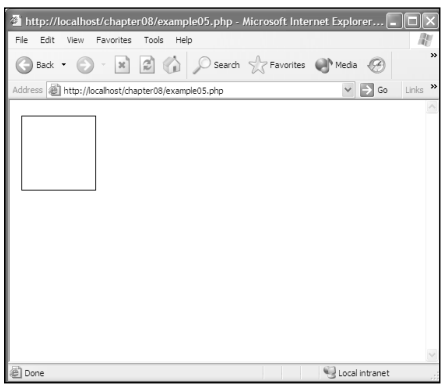


Figure 8.6 The `ImageRectangle()` function.

`ImageFilledRectangle()` functions exactly like `ImageRectangle()`, except it creates a filled rectangle. `ImageFilledRectangle()` takes the same six parameters as `ImageRectangle()`. Take a look at the following code example to see the `MyFilledRectangle()` function:

```
<?php
function MyFilledRectangle($image, $x1, $y1, $width, $height, $color)
{
    // First you need to calculate the values for x2 and y2
    $x2 = $x1 + $width;
    $y2 = $y1 + $height;

    // Now draw the rectangle
    ImageFilledRectangle($image, $x1, $y1, $x2, $y2, $color);
}

$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, 0, 0, $white);

MyFilledRectangle($image, 5, 5, 100, 100, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The results of the code above are shown in Figure 8.7.

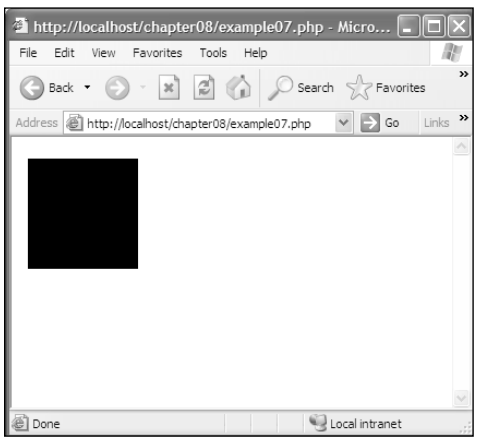


Figure 8.7 The `MyFilledRectangle()` function.

From Rectangles to Polygons

As mentioned earlier, the concepts of each geometric shape will get progressively more difficult. However, drawing a polygon isn't that complicated. The major difference between drawing a polygon and drawing a rectangle is the number of coordinates the function will take. So far you have dealt only with functions that take one coordinate (x, y). The `ImagePolygon()` function takes multiple coordinates. It also takes the number of points that are in your polygon. When the `ImagePolygon()` function draws your polygon it connects each point with a line.

`ImagePolygon()` takes four arguments. The first is, of course, the resource to the image. The next argument is an array that contains all of your point coordinates for the polygon. The third argument is an integer telling `ImagePolygon()` how many vertices are in your polygon. The fourth and final argument is the color in which you want your polygon to be drawn.

Let's say you wanted to draw a triangle on your canvas. You would need three sets of coordinates. You can store these coordinates in an array, for example:

```
$points = array(50, 10,
                20, 40,
                80, 40);
```

This gives you each corner of the triangle. Now you need to calculate how many points there are in the given polygon. With a triangle it is simple; there are obviously three points. But what would you do for a more complex shape? Well, since there are two elements in every point you can divide the size of the array by 2 to get the number of points in the polygon.

```
$vertices = sizeof($points) / 2;
```

Remember in the last section that I said anytime you see a way to make the code a little simpler, do it. Well, what if you created a function that took three arguments instead of four to create a polygon? That would be great, wouldn't it? You could save doing the number of vertices calculation every time you wanted to draw a polygon.

```
<?php
function MyPolygon($image, $points, $color)
{
    // Calculate the number of vertices in the polygon
    $vertices = sizeof($points) / 2;
    // Draw the polygon to the canvas
    ImagePolygon($image, $points, $vertices, $color);
}
?>
```

I know it doesn't seem like much now, but what if you were constantly drawing polygons and you had to redo that stupid division calculation every single time? Take a look at the following example to see how to draw a polygon:

```
<?php
function MyPolygon($image, $points, $color)
{
    // Calculate the number of vertices in the polygon
    $vertices = sizeof($points) / 2;
    // Draw the polygon to the canvas
    ImagePolygon($image, $points, $vertices, $color);
}
$points = array(50, 10,
                20, 40,
                80, 40);
$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, 0, 0, $white);

MyPolygon($image, $points, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The results of this example can be seen in Figure 8.8.

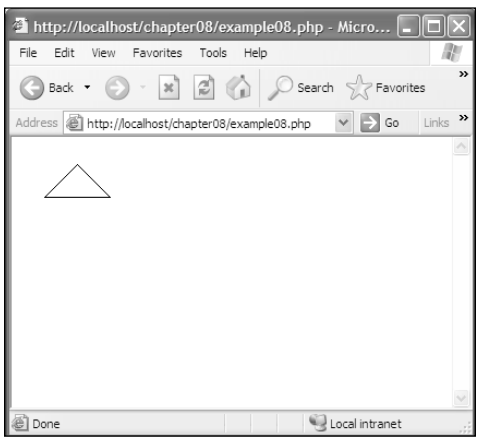


Figure 8.8 Drawing a polygon with the `ImagePolygon()` function.

The `ImageFilledPolygon()` function takes the same parameters as the `ImagePolygon()` function, but instead of drawing an outline of the polygon it draws a filled-in version of the polygon. Here is a modified function for the `ImageFilledPolygon()` function:

```
<?php
function MyFilledPolygon($image, $points, $color)
{
    // Calculate the number of vertices in the polygon
    $vertices = sizeof($points) / 2;
    // Draw the polygon to the canvas
    ImageFilledPolygon($image, $points, $vertices, $color);
}
?>
```

From Polygons to Arcs and Ellipses

Wow, you are quickly becoming a master at creating on-the-fly images with GD. After you learn how to create arcs and ellipses, you will move on to putting dynamic text in your graphics, and finally you will take a more in-depth look at how to display and save your on-the-fly images.

GD provides you with a function called `ImageArc()` to create, well, arcs. This function takes a whopping eight arguments. The first is the resource to the image. The next two arguments are the center x point for the arc and the center y point for the arc. The fourth and fifth arguments are the desired width of the arc, and then the desired height of the arc. The sixth argument is the starting degree for the arc. The seventh argument is the ending degree for the arc. The eighth and final argument is the color that you would like the arc to appear in.

Note

The `ImageArc()` function draws in a clockwise direction.

The `ImageArc()` function puts 0 degrees at three o'clock, 90 degrees at six o'clock, 180 degrees at nine o'clock, and 270 degrees at twelve o'clock. Although documentation on www.php.net says that the `ImageArc()` function draws counter-clockwise, it actually draws clockwise.

Take a look at how you would use the `ImageArc()` function and the results.

```
<?php
$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, 0, 0, $white);

ImageArc($image, 50, 30, 90, 90, 0, 220, $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The code example above draws an arc that is 90 pixels wide and 90 pixels high, with its center point at (50, 30). It starts at the 0 degree mark and ends at the 220 degree mark. Take a look at Figure 8.9 to see the results of the above code.

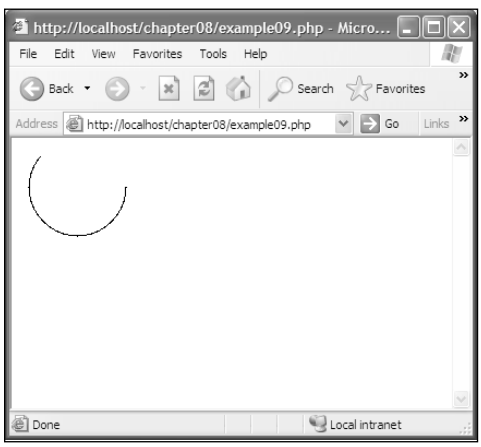


Figure 8.9 The `ImageArc()` function.

The `ImageFilledArc()` function behaves in the same way as the `ImageArc()` function. It also draws its arc starting from three o'clock and going in a clockwise direction. But the `ImageFilledArc()` function takes an additional argument. The ninth argument, after color, is the style in which the arc should be drawn. You can also use a bitwise OR to combine styles. Take a look at what the styles are and how they affect the arc.

- **IMG_ARC_PIE**. Draws a pie chart-styled arc with solid lines connecting the center point to the edges of the arc.
- **IMG_ARC_CHORD**. Draws a triangle that connects the beginning and end points of the arc.
- **IMG_ARC_NOFILL**. If this option is used, it behaves like the `ImageArc()` function.
- **IMG_ARC_EDGED**. Connects the end points of the arc to its center.

Let's use each of these individually in a code example and take a look at the results so you know how each style affects the outcome of your arc.

```
<?php
$image1 = ImageCreate(320, 200);
$image2 = ImageCreate(320, 200);
$image3 = ImageCreate(320, 200);
$image4 = ImageCreate(320, 200);

$white1 = ImageColorAllocate($image1, 255, 255, 255);
$white2 = ImageColorAllocate($image2, 255, 255, 255);
$white3 = ImageColorAllocate($image3, 255, 255, 255);
$white4 = ImageColorAllocate($image4, 255, 255, 255);
$black1 = ImageColorAllocate($image1, 0, 0, 0);
$black2 = ImageColorAllocate($image2, 0, 0, 0);
$black3 = ImageColorAllocate($image3, 0, 0, 0);
$black4 = ImageColorAllocate($image4, 0, 0, 0);

ImageFill($image1, $white1);
ImageFilledArc($image1, 50, 30, 90, 90, 0, 220, $black1, IMG_ARC_PIE);
ImagePng($image1, "pie.png");
ImageDestroy($image1);

ImageFill($image2, $white2);
ImageFilledArc($image2, 50, 30, 90, 90, 0, 220, $black1, IMG_ARC_CHORD);
ImagePng($image2, "chord.png");
ImageDestroy($image2);
```

```
ImageFill($image3, $white3);
ImageFilledArc($image3, 50, 30, 90, 90, 0, 220, $black3, IMG_ARC_NOFILL);
ImagePng($image3, "nofill.png");
ImageDestroy($image3);

ImageFill($image4, $white4);
ImageFilledArc($image4, 50, 30, 90, 90, 0, 220, $black4, IMG_ARC_EDGED);
ImagePng($image4, "edged.png");
ImageDestroy($image4);
?>
<table border="1" cellpadding="2" cellspacing="2">
    <tr>
        <td align="left" valign="top"><img src="pie.png"></td>
        <td align="left" valign="top"><img src="chord.png"></td>
    </tr>
    <tr>
        <td align="left" valign="top">IMG_ARC_PIE</td>
        <td align="left" valign="top">IMG_ARC_CHORD</td>
    </tr>
    <tr>
        <td align="left" valign="top"><img src="nofill.png"></td>
        <td align="left" valign="top"><img src="edged.png"></td>
    </tr>
    <tr>
        <td align="left" valign="top">IMG_ARC_NOFILL</td>
        <td align="left" valign="top">IMG_ARC_EDGED</td>
    </tr>
</table>
```

This produces the screen that you see in Figure 8.10.

The one obvious geometric shape missing from GD is a function to create an ellipse. Or is it? You can use the `ImageArc()` and `ImageFilledArc()` functions to create an ellipse. All you have to do is specify a starting degree of 0 and an ending degree of 360. This will create a full ellipse. For example:

```
<?php
$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, $white);

ImageArc($image, 50, 40, 70, 40, 0, 360, $black);
```

```
header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

This generates an ellipse that looks like Figure 8.11.

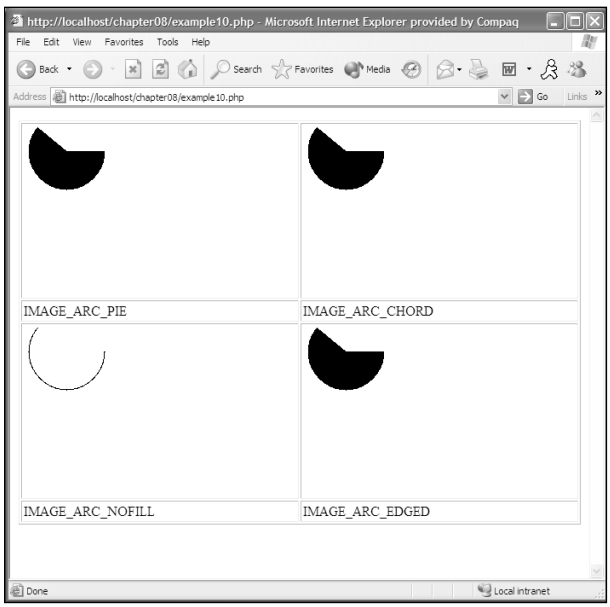


Figure 8.10 The `ImageFilledArc()` function.

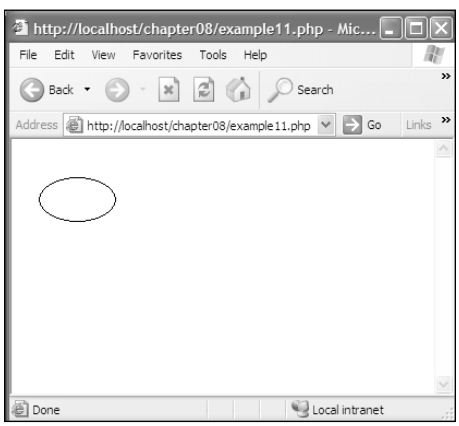


Figure 8.11 Creating an ellipse.

Creating Images with Text

You have seen how GD can handle drawing geometry on the canvas, but what if you have some text that you want to dynamically put into your graphic? GD provides you with three main functions for doing just that. They are:

- `ImageString(resource image, int fontNumber, int x, int y, string text, int color)`
- `ImageTTFText(resource image, int size, int angle, int x, int y, int color, string fontFile, string text)`
- `ImageTTFBBox(int size, int angle, string fontFile, string text);`

The `ImageString()` function is fairly straightforward. It takes six parameters. The first is the resource to the image. The second is a font number, from 1 to 5, that uses a built-in font to write out your text. The third and fourth arguments are the location that you want the text to start at. The fifth argument is the text that you want displayed. The final argument is the color in which you want the text to be displayed. This is the simplest of all three font functions and gives you the least amount of flexibility.

```
<?php
$image = ImageCreate(320, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, $white);

ImageString($image, 1, 10, 10, "Text In Font 1", $black);
ImageString($image, 2, 10, 30, "Text In Font 2", $black);
ImageString($image, 3, 10, 50, "Text In Font 3", $black);
ImageString($image, 4, 10, 70, "Text In Font 4", $black);
ImageString($image, 5, 10, 90, "Text In Font 5", $black);

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The code example above goes through each font size that can be rendered by the `ImageString()` function. Figure 8.12 shows the results of the code. As you can see, it isn't a very pretty font.

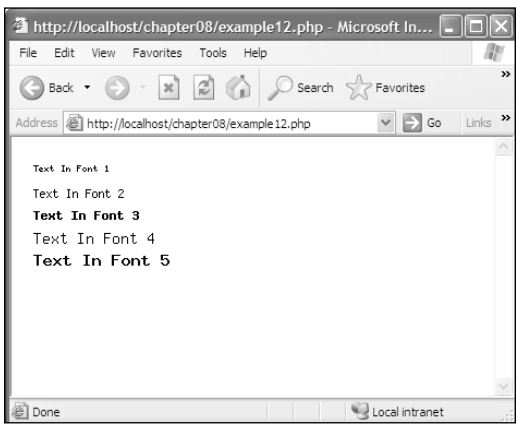


Figure 8.12 Adding text with the `ImageString()` function.

A more useful function for rendering fonts to your canvas is `ImageTTFText()`. The `ImageTTFText()` function takes eight arguments. The first is the resource to the image; the second argument is the size in which you want to render the text. The third argument is the angle at which you want to render the text. This will allow you to render text sideways, diagonally, and even upside down—it really is a fun argument to play with. The fourth and fifth arguments are the (x, y) coordinate that you want the text to start at. The sixth argument is the color you want to render the font in. The seventh argument is the True Type Font you want to use, and the final argument is the text you want to render.

The `ImageTTFText()` function returns an array with eight elements that represent the four points that make the bounding box of the text. Take a look at Table 8.1 to see what index points to which point.

Table 8.1 Bounding Box Coordinates

Index in the Array	Description
0	Lower left x position.
1	Lower left y position.
2	Lower right x position.
3	Lower right y position.
4	Upper right x position.
5	Upper right y position.
6	Upper left x position.
7	Upper left y position.

* You start in the lower left corner and work your way counter-clockwise around the box.

Tip

By default, `ImageTTFText()` renders anti-aliased text. If you would like the font to be rendered as aliased text you need to put a "-" (minus) sign in front of the color.

Now try rendering some text in a cool True Type Font. I am using Matisse ITC for this example.

```
<?php
$image = ImageCreate(500, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
ImageFill($image, 0, 0, $white);

ImageTTFText($image, 36, 0, 0, 80, $black, "matisse_.ttf", "Cool Anti-Aliased Text!!!");
ImageTTFText($image, 36, 0, 0, 150, -$black, "matisse_.ttf", "Cool Aliased Text!!!");

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

Caution

To run properly, the font file must be located in the same directory as your PHP file.

In this example, I rendered both anti-aliased text and aliased text to show you the difference. The results are shown in Figure 8.13. As you can see, the aliased text is very jagged around the edges and not very pretty. I imagine there will be very few cases where you'll use the aliased font.

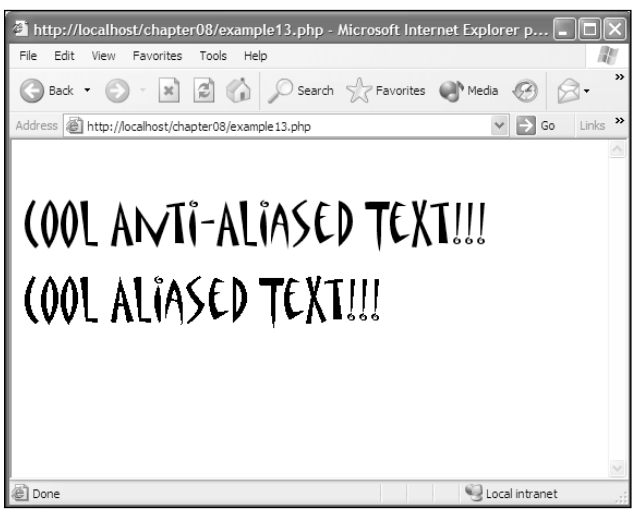


Figure 8.13 Rendering text with the `ImageTTFText()` function.

Caution

There is a bug in the GD library in PHP 4.2. The bug will cause `ImageTTFText()` to throw an error. This shouldn't be a problem since PHP 5 has been released. If you are still using PHP 4.2, upgrade!

I know you are saying, "Cool, but what if I want to do some font effects, like embossing?" You can do that with `ImageTTFText()` too. It just requires some careful placement of colors and text. Take the same example you just did but add a few more lines of code to it.

```
<?php
$image = ImageCreate(500, 200);
$white = ImageColorAllocate($image, 255, 255, 255);
$black = ImageColorAllocate($image, 0, 0, 0);
$gray = ImageColorAllocate($image, 155, 155, 155);
ImageFill($image, $white);

ImageTTFText($image, 36, 0, 2, 81, $black, "matisse_.ttf", "Cool Embossed Text!!!");
ImageTTFText($image, 36, 0, 0, 79, $white, "matisse_.ttf", "Cool Embossed Text!!!");
ImageTTFText($image, 36, 0, 0, 80, $gray, "matisse_.ttf", "Cool Embossed Text!!!");

header("Content-type: image/png");
ImagePng($image);
ImageDestroy($image);
?>
```

The first step in creating the effect that you see in Figure 8.14 is to render the black border around the text. To do this you just add 1 to the (x, y) coordinate. Next you want to render the text again with white to create a separation between the gray and the black. To do this you subtract 1 from the (x, y) coordinate. The final step is to render the text in gray at the specified (x, y) coordinate.

The final text function to learn is the `ImageTTFBBox()` function. This function takes four arguments. The first is the size of the font, then the angle, followed by the True Type Font file, and, lastly, the text you want to render. `ImageTTFBBox()` returns an array with the bounding box of the text. This is an extremely useful function when you need to make sure that the text you are going to render is within the image, unless you don't mind cutting your text off in mid-sentence.

The returned array is ordered exactly the same as the `ImageTTFText()` function's array. It starts at the lower left-hand corner of the box and works its way around in a counterclockwise fashion.

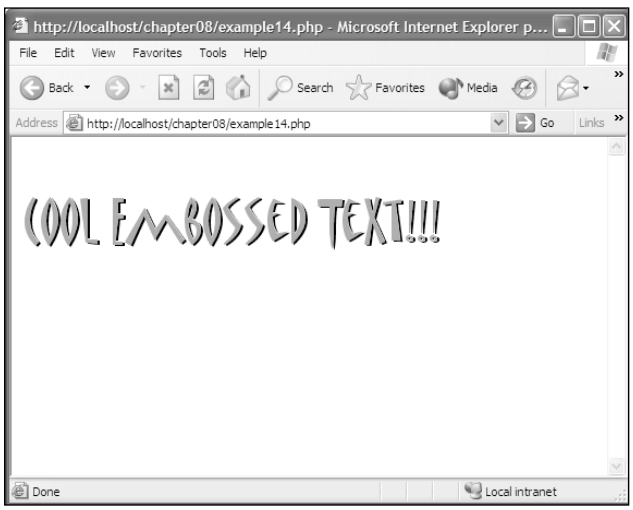


Figure 8.14 Creating cool text effects with the `ImageTTFText()` function.

Now create a function to put in our common.php file that will calculate the width and height of the bounding box for the text. Your function will also need to take four arguments, but you will want to return two integers. So your function will actually need to take six arguments.

```
<?php
function MyTTFBox($size, $angle, $fontfile, $text, &$width, &$height)
{
    // Get the bounding box
    $arrBox = ImageTTFBBox($size, $angle, $fontfile, $text);

    // Now calculate the width and the height of the box
    $width = abs($arrBox[6] - $arrBox[2]);
    $height = abs($arrBox[7] - $arrBox[3]);
}

$myWidth = 0;
$myHeight = 0;

MyTTFBox(36, 0, "matisse_.ttf", "Cool Embossed Text!!!", $myWidth, $myHeight);


echo("The Bounding box is $myWidth pixels by $myHeight pixels.");
?>
```

The `MyTTFBox()` function takes the same four arguments as the `ImageTTFBBox()` function plus two more arguments—the width and the height. The width and the height are passed by reference, meaning that anything that changes these variables in the function will also affect the results of the variables that were passed in. Take a look at Figure 8.15 to see the results of your new function.

Saving Your Images

Throughout the examples you have seen the use of `ImagePng()` and `ImageJpeg()`. These two functions are used to save images to a file and render images to the browser. There is a third function called `ImageWbmp()` that renders and saves WBMP files. All three of these functions take two arguments. The first is required: the resource to the image. The second argument is optional; you can specify a filename that you would like to save the image data to. Just make sure that if you save a file you use the proper extension. Don't try to save a .png file while using the `ImageJpeg()` function and vice versa.

```
// This saves a PNG file
ImagePng($image, "myimage.png");
// This saves a JPEG file
ImageJpeg($image, "myimage.jpg");
// This saves a WBMP file
ImageWbmp($image, "myimage.wbmp");
```

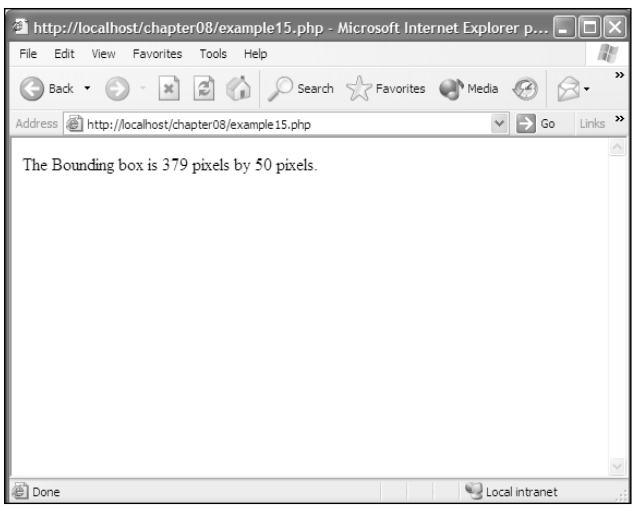


Figure 8.15 The `MyTTFBox()` function.

If you want to render the image directly to the browser you must specify a content type. You can do this by using the built in PHP `header()` function. To render a .png you have to specify a content type of image/png. To render a .jpeg you have to specify a content type of image/jpg. To render a WBMP you have to specify a content type of image/wbmp.

```
// This renders a PNG to the browser
header("Content-type: image/png");
ImagePng($image);
// This renders a JPEG to the browser
header("Content-type: image/jpg");
ImageJpeg($image);
// This renders a WBMP to the browser
header("Content-type: image/wbmp");
ImageWbmp($image);
```

Using Existing Images

Throughout this chapter you have seen how to create images, use colors, draw geometrical shapes to the canvas, and render text to your images. Now you will learn how to use existing images to create new images. There are six basic functions that you will use. The first two are the basis of using existing images to make new images.

- `ImageCreateFromJpeg(string filename)`
- `ImageCreateFromPng(string filename)`

Each of these functions returns a resource to an image. You cannot create an image from a WBMP, only from .pngs and .jpegs.

```
// Get a resource to an existing image
$image = ImageCreateFromPng("someimage.png");
```

After you have opened a file and retrieved a resource you can copy, resize, resample, or merge the image with another image with the following functions:

- `ImageCopy(resource destinationImage, resource sourceImage, int desinationX, int destinationY, int sourceX, int sourceY, int sourceWidth, int sourceHeight)`
- `ImageCopyResized(resource destinationImage, resource sourceImage, int desinationX, int destinationY, int sourceX, int sourceY, int destinationWidth, int destinationHeight, int sourceWidth, int sourceHeight)`
- `ImageCopyResampled(resource destinationImage, resource sourceImage, int desinationX, int destinationY, int sourceX, int sourceY, int destinationWidth, int destinationHeight, int sourceWidth, int sourceHeight)`

- `ImageCopyMerge(resource destinationImage, resource sourceImage, int destinationX, int destinationY, int sourceX, int sourceY, int sourceWidth, int sourceHeight, int percent)`

I have included two images on the CD called dragon.jpg and frame.jpg. You will use these images to create a new image using the `ImageCopy()` function. The images appear in Figure 8.16.

Now you will copy dragon.jpg onto frame.jpg using the `ImageCopy()` function.

```
<?php
// Get our image resources
$dragon = ImageCreateFromJpeg("dragon.jpg");
$frame  = ImageCreateFromJpeg("frame.jpg");

// Copy dragon onto frame
ImageCopy($frame, $dragon, 21, 31, 0, 0, 477, 462);

// Save the image and show it
ImageJpeg($frame, "frameDragon.jpg");

echo("<img src=frameDragon.jpg>");
?>
```

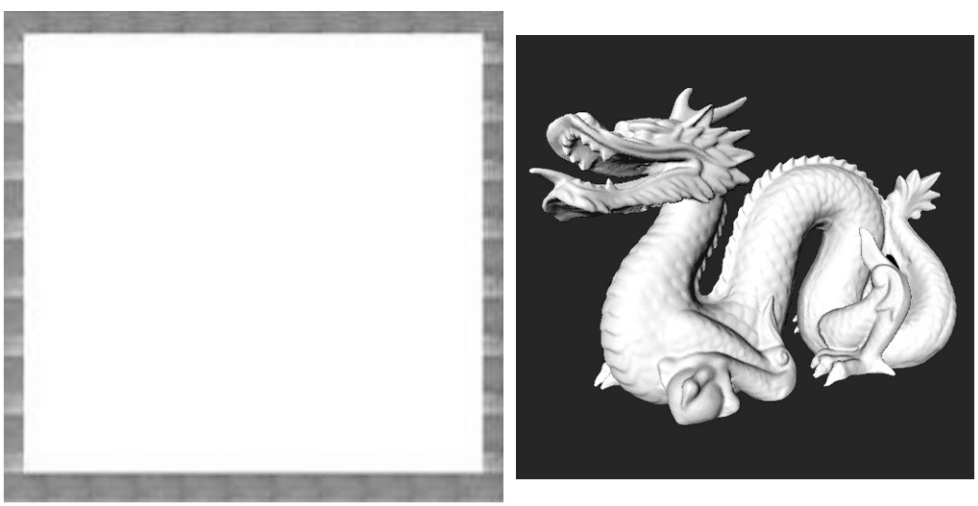


Figure 8.16 Images for use in the `ImageCopy()` function.

I'll break down each element of the `ImageCopy()` function in this example. The first argument is the destination image, which in this case is $frame. The second argument is the source image, which is $dragon. The next argument is what pixel you want to put the source image in in the destination image. Since the frame ends at the coordinate (21, 31), that is where you want to put the source image. The next argument is what pixel you want to grab the image at. Since the dragon image fits perfectly inside the frame, you just grab it starting at the coordinate (0, 0). The final arguments are the width and height of the source image. Since the dragon does fit perfectly within the frame, you can specify the actual width and height of the image. But what if the dragon was the same size as the frame?

If both the images were the same size, you would have to do a little math to get the same results as Figure 8.17. You would want to first grab the source image from the same point you were placing the image. In the example above, it would be (0, 0). Then you would want to subtract that amount from the overall width and height of the image. In the example above, it would be (456, 431). So the new `ImageCopy()` line would look like this:

```
ImageCopy($frame, $dragon, 21, 31, 0, 0, 456, 431);
```

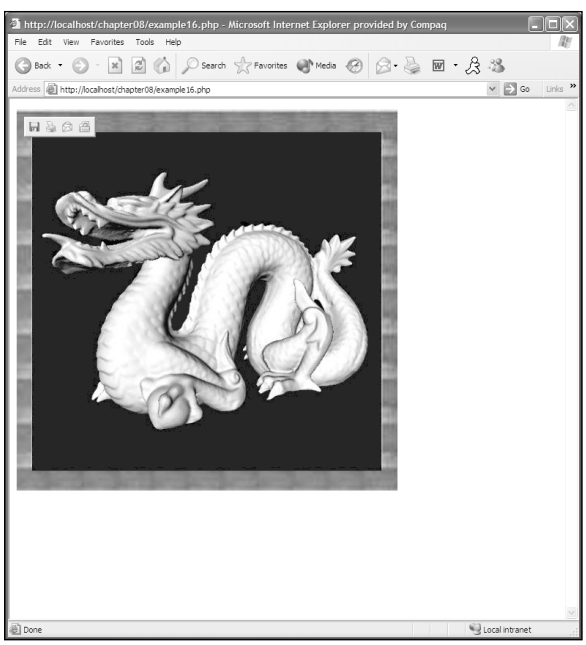


Figure 8.17 Copying an image with the `ImageCopy()` function.

So now you know how to copy an image that is either the same size or smaller than the destination image. But what if you want to copy an image that is larger than the destination image? You would use the `ImageCopyResized()` or the `ImageCopyResampled()` functions. A good example would be to take the framedDragon.jpg image that you just created and create a 100 × 100 thumbnail of the image.

```
<?php
// Get our image resources
$image = ImageCreateFromJpeg("frameDragon.jpg");

// Create a true color image 100x100
$thumb = ImageCreateTrueColor(100, 100);

// Copy dragon onto frame
ImageCopyResized($thumb, $image, 0, 0, 0, 0, 100, 100, ImagesX($image),
ImagesY($image));

// Save the image and show it
ImageJpeg($thumb, "thumbFrameDragon.jpg");

echo("<img src=thumbFrameDragon.jpg>");
?>
```

This starts off the same way as the last example, by getting the resource to an existing image. Then you create a new true-color image that is 100 × 100 pixels. Now you copy the image $image to the destination image $thumb. The seventh and eighth arguments are the destination image's width and height. The ninth and tenth arguments are the source image's width and height. In this example you use the `ImagesX()` and `ImagesY()` functions that return the width and height of the image resource. A full listing of all image functions can be found in Appendix D. Take a look at Figure 8.18 to see the results of the example above.

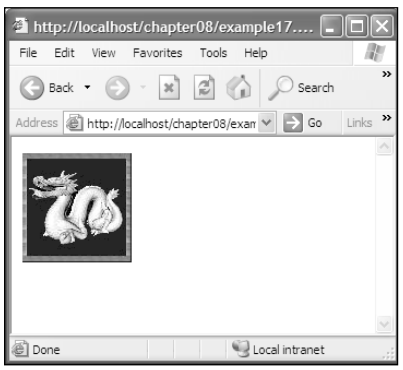


Figure 8.18 Using the `ImageCopyResized()` function.

I mentioned earlier in this chapter that the `ImageCopyResized()` function and the `ImageCopyResampled()` function do the same thing. Well, technically they do the same thing, but with one minor difference. After the `ImageCopyResampled()` function copies and resizes the image, it also resamples it. The results are a crisper, cleaner image. In the upcoming code example you will resize the image using `ImageCopyResized()` and then `ImageCopyResampled()`. After they are resized you will save them and compare the results.

```
<?php
// Get our image resources
$image = ImageCreateFromJpeg("frameDragon.jpg");

// Create a true color image 100x100
$thumb = ImageCreateTrueColor(250, 250);
$thumb2 = ImageCreateTrueColor(250, 250);

// Copy dragon onto frame
ImageCopyResized($thumb, $image, 0, 0, 0, 0, 250, 250, ImagesX($image),
ImagesY($image));
ImageCopyResampled($thumb2, $image, 0, 0, 0, 0, 250, 250, ImagesX($image),
ImagesY($image));

// Save the image and show it
ImageJpeg($thumb, "thumbFrameDragon1.jpg");
ImageJpeg($thumb2, "thumbFrameDragon2.jpg");

echo("<img src=thumbFrameDragon1.jpg>   <img src=thumbFrameDragon2.jpg>");
?>
```

The results of the code example are shown in Figure 8.19.

The difference is noticeable. The top image is the one made with `ImageCopyResized()`. It has quite a bit of artifacting and looks quite grainy compared to the bottom image, which was created with `ImageCopyResampled()`.

So why would you ever use `ImageCopyResized()`? Well, the only reason I can think of is speed. But since you aren't batching a ton of images at once, you might as well use the `ImageCopyResampled()` function whenever speed isn't a concern. After all, you do want to get the best results you possibly can.

The final function to look at is the `ImageCopyMerge()` function. This function behaves exactly like the `ImageCopy()` function with one very key difference. The extra parameter (int percent) tells the image how opaque the image should be on the destination image. This allows you to create some cool backgrounds or translucent images.

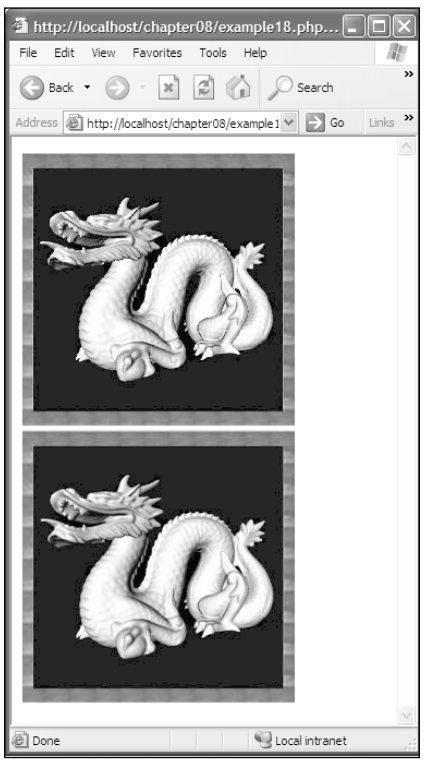


Figure 8.19 Results of the `ImageCopyResized()` [top] and `ImageCopyResampled()` [bottom] functions.

Let's merge our dragon image onto a white image with 20% opacity. This will allow most of the white to show through while displaying a faint image of the dragon.

```
<?php
// Get our image resources
$dragon = ImageCreateFromJpeg("dragon.jpg");

// Create a true color white image
$whiteImage = ImageCreateTrueColor(512, 512);
$white = ImageColorAllocate($whiteImage, 255, 255, 255);
$black = ImageColorAllocate($whiteImage, 0, 0, 0);
$gray = ImageColorAllocate($whiteImage, 155, 155, 155);
ImageFill($whiteImage, 0, 0, $white);
```

```
// Copy dragon onto frame
ImageCopyMerge($whiteImage, $dragon, 0, 0, 0, 0, ImagesX($dragon), ImagesY($dragon),
20);

ImageTTFText($whiteImage, 36, 45, 201, 257, $black, "matisse_.ttf", "Dragon!!!");
ImageTTFText($whiteImage, 36, 45, 199, 254, $white, "matisse_.ttf", "Dragon!!!");
ImageTTFText($whiteImage, 36, 45, 200, 256, $gray, "matisse_.ttf", "Dragon!!!");

// Show the new image
header("Content-type: image/jpeg");
ImageJpeg($whiteImage);
ImageDestroy($whiteImage);
?>
```

The results of all of your hard work are shown in Figure 8.20.

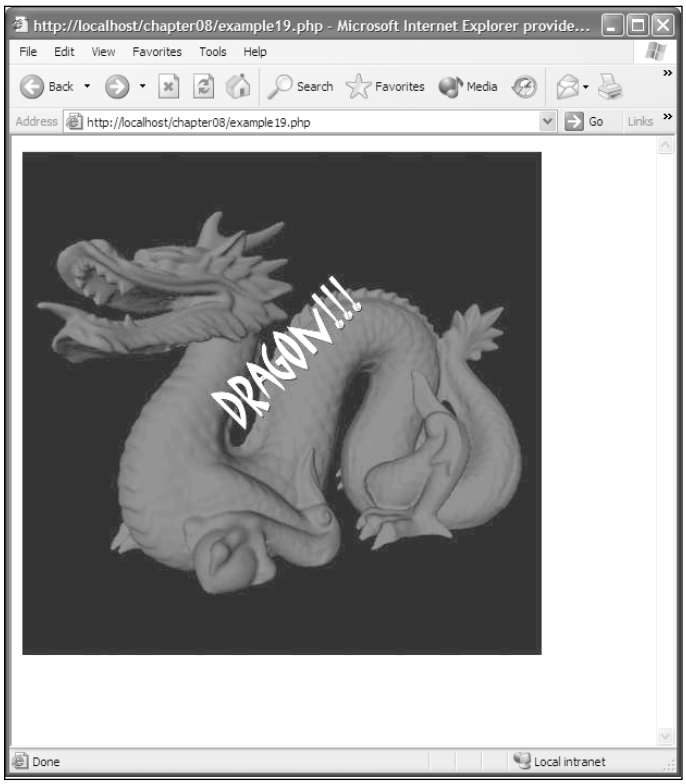


Figure 8.20 The results of your hard work!

Conclusion

That's it! You are now officially a dynamic image master in PHP. You learned how to create indexed and true-color images. You also learned how to allocate colors to an image. You became a god at drawing dynamic shapes onto your empty canvases. You also learned how to generate text from a True Type Font and use it in your image. Finally, you learned how to use existing images to your advantage to create some really cool-looking graphics.

Next up: creating a game called Battle Tank and creating dynamic terrain for your new tank game!