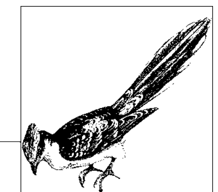
PHP

IN A NUTSHELL

A Desktop Quick Reference

Paul Hudson

16



Manipulating Images

Lots of people stereotype PHP as only being suitable for outputting text, but that's not true—you can use PHP to create complex and dynamic pictures using the GD image extension. This chapter covers many of the GD functions that will allow you to make your own images for your site, either from scratch or by using existing images.

For image manipulation purposes, PHP ships with its own copy of the popular GD library. You used to have to get your own copy of GD and hope it was compatible with your PHP version. This is no longer the case. The copy of GD that ships with PHP will work with that version of PHP.

Getting Started

An important PHP function when working with images is `header()`. This outputs a HTTP header of your choice; in this situation, we will be sending the content-type header, which tells web browsers what kind of content they can expect through the connection. Popular content types include text/plain for plain text documents; text/html for most web pages; and image/*, where the * is *png*, *jpeg*, *gif*, or MIME types for other picture formats.

As `header()` sends HTTP headers, it must be used before you send any content through. This is a core HTTP rule—no headers can be sent after content. This is the same thing that stops you from using cookies after you have sent content. The `header()` function is covered in more detail in Chapter 20, but for now, we will just work with this one aspect of it.

Creating a new image is done with the `imagecreate()` function, which has two parameters: the height and width of the image you wish to create. This will return `false` if it failed to create an image, which is usually the result of a lack of memory; otherwise, it will return the image as a resource for you to use in other image functions. To free up this image's memory, pass that resource into `imagedestroy()` as its only parameter.

Once you have your image resource, it is yours to play with all you want. PHP provides a selection of functions for you to use to manipulate the image. When you are done, you just choose your output format and the picture is finished.

To output the picture, you call one of several functions. If you want to convert it to PNG format, you call `imagepng()`. This function takes two parameters, which are the image resource to use and a filename to save the picture as (optional). If you don't provide the second parameter, `imagepng()` sends the PNG-formatted picture straight to output, which is usually a visitor to your site.

To choose JPEG, you call the `imagejpeg()` function, which takes three parameters—the same two as `imagepng()`, plus the quality you wish to use for the picture. The quality, a number between 0 (lowest quality, smallest file) and 100 (highest quality, largest file), is optional, as is the filename parameter. If you want to set the quality without specifying a filename, just provide an empty string ('') as the filename.

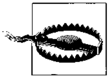
The most basic image script looks like this:

```
$image = imagecreate(400,300);
// do stuff to the image
imagejpeg($image, '', 75);
imagedestroy($image);
```

Save that as *picture1.php*. As most of your pictures will probably be referenced from a web page, we will also make a companion web page. Save this as *phppicture.html*:

```
<html>
<title>PHP Art</title>
<body>
PHP woz 'ere:
<img src="picture1.php" />
</body>
</html>
```

Open up your web browser and load in *phppicture.html*—you should see a large black box for the image, as shown in Figure 16-1.



Be sure not to have anything outside the PHP code block, not even an empty line or a space. Everything outside the PHP block is sent to the browser as part of the picture, and even having a single space character at the end of the file will cause problems.

Manipulating Images

The next step is to add a little color in place of the "do stuff to the image" comment, so we need `imagecolorallocate()` (note that you must use U.S. spellings for these function names). This new function takes four parameters: the image resource you are choosing a color for, then three integers between 0 and 255—one each for the red value, then green value, and the blue value of the color. You can also specify these colors in hexadecimal format (e.g., 0xff) rather than decimal.

Figure 16-1. Our first picture using PHP is a big square colored entirely black—not exactly a stunner, but a good start

The first color you allocate is automatically used as the background color for your image, so this next piece of code is a minor modification of the last script to include color information:

```
$image = imagecreate(400,300);
$gold = imagecolorallocate($image, 255, 240, 00);
imagepng($image);
imagedestroy($image);
```

Save that over *picture1.php*, and refresh *phppicture.html*—you should see the black square replaced by a yellow square.

Don't worry about deallocating colors, as they are just numbers and not resources, meaning they don't use up any special memory. If you really want to deallocate a color (perhaps if you're working with a paletted image), use the `imagecolordeallocate( )` function.

Choosing a Format

For high-quality images with many colors or a lot of detail, the JPEG format is preferred. JPEG saves in true color and allows you to set the compression ratio in order to get the best trade-off between size and quality. PNGs, on the other hand, work best as a replacement for GIFs, and as such, work well using limited colors. They also offer alpha transparency and quite small file sizes.

So, put as simply as possible: for photographs, prefer JPEGs, and for everything else, prefer PNGs. Just as an aside, and at the risk of starting a flame war, the colorcorrect pronunciations are "ping," "jay-peg," and "jif." Note that WBMP is not Windows Bitmap, as you might have first thought—it stands for Wireless Bitmap and is designed for use in limited bandwidth situations.

Getting Arty

The `imagefilledrectangle()` function takes six parameters in total, which are, in order: an image resource to draw on, the top-left X coordinate, the top-left Y coordinate, the bottom-right X coordinate, the bottom-right Y coordinate, and a color to use. There is a similar function called `imagerectangle()`, which takes the same parameters but only draws the *outline* of the rectangle, whereas `imagefilledrectangle()` fills the shape with color.

In order to draw a rectangle in such a way as to make it stand out, we need to allocate another color and then draw the rectangle. Here is how that is done:

```
$white = imagecolorallocate($image, 255, 255, 255);
imagefilledrectangle($image, 10, 10, 390, 290, $white);
```

Put those two lines just after the definition of `$gold`, then save the modified script and refresh *phppicture.html*.

This function becomes more interesting when used in a loop, like this:

```
$image = imagecreate(400,300);
$gold = imagecolorallocate($image, 255, 240, 00);
$white = imagecolorallocate($image, 255, 255, 255);
$color = $white;

for ($i = 400, $j = 300; $i > 0; $i -= 4, $j -= 3) {
        if ($color == $white) {
                $color = $gold;
        } else {
                $color = $white;
        }

        imagefilledrectangle($image, 400 - $i, 300 - $j, $i, $j, $color);
}

imagepng($image);
imagedestroy($image);
```

That script calls `imagefilledrectangle()` each iteration of the loop, slowly making the rectangle smaller and smaller as `$i` and `$j` decrease in value. Your output should look like Figure 16-2.

In place of a plain color, it is possible to fill your shapes with a tiled image using the `imagesettile()` function.

Manipulating Images

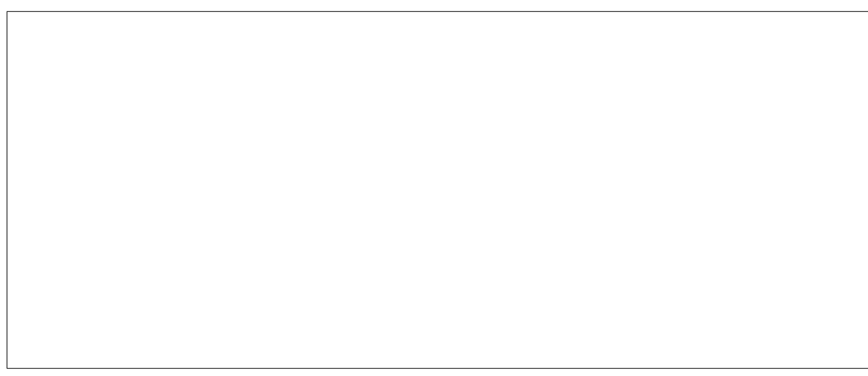


Figure 16-2. Using a simple loop, we've turned our simple rectangle into a series of concentric rectangles

More Shapes

Using three new functions, we can make a much more complicated image. These are: `imagecreatetruecolor( )`, `imagefilledellipse( )`, and `imagefilledarc( )`.

Here is a script using these new functions:

```
header("content-type: image/png");

$image = imagecreatetruecolor(400,300);
$blue = imagecolorallocate($image, 0, 0, 255);
$green = imagecolorallocate($image, 0, 255, 0);
$red = imagecolorallocate($image, 255, 0, 0);

imagefilledellipse($image, 200, 150, 200, 200, $red);
imagefilledellipse($image, 200, 150, 180, 180, $blue);
imagefilledellipse($image, 200, 150, 50, 50, $red);
imagefilledarc($image, 200, 150, 200, 200, 345, 15, $green, IMG_ARC_PIE);
imagefilledarc($image, 200, 150, 200, 200, 255, 285, $green, IMG_ARC_PIE);
imagefilledarc($image, 200, 150, 200, 200, 165, 195, $green, IMG_ARC_PIE);
imagefilledarc($image, 200, 150, 200, 200, 75, 105, $green, IMG_ARC_PIE);

imagepng($image);
imagedestroy($image);
```

The output from that script is shown in Figure 16-3.

Figure 16-3. Ellipses and circles

Using imagecreatetruecolor() is the same as imagecreate()—it takes the same two parameters, and returns an image resource that is freed using imagedestroy(). The difference between the two is that imagecreatetruecolor() returns an image with a true-color palette, whereas an image made by imagecreate() cannot contain more than 256 colors. Furthermore, the image resource returned by imagecreatetruecolor() automatically has a black background, so you needn't worry about the first allocated color being used as the image background color.

The two new shape functions take several parameters, so you may need to keep the list at hand when working with them. The parameters for imagefilledellipse() are: image resource, center of ellipse (X coordinate), center of ellipse (Y coordinate), height, width, and color. As there are more parameters required to draw an arc, imagefilledarc() is more complicated again: image resource, center X, center Y, height, width, then the start and end points of the arc specified in degrees, followed by color and, finally, the type of arc to draw.

The start and end points for arcs are specified from 0 to 359 degrees, with 0 pointing directly to the right, or 3 o'clock if you think in clock faces. To draw a complete circle rather than just a section, as in the example, you would specify 0 and 359 as the start and end points; although, in this case, it is easier just to use imagefilledellipse(). The final parameter to imagefilledarc() is the type of arc to draw, and you have the choice of the following:

- IMG_ARC_PIE, as in the previous example, which draws a filled wedge shape with a curved edge
- IMG_ARC_CHORD, which draws a straight line between the starting and ending angles
- IMG_ARC_NOFILL, which draws the outside edge line without drawing the two lines toward the center of the arc
- IMG_ARC_EDGED, which draws an unfilled wedge shape with a curved edge

You can combine these four together in various ways to make your own style of arc, with the exception of IMG_ARC_CHORD and IMG_ARC_PIE, which cannot be combined together because they conflict geometrically. Some examples:

```
imagefilledarc($image, 200, 150, 200, 200, 345, 15, $green,
        IMG_ARC_CHORD | IMG_ARC_NOFILL);
imagefilledarc($image, 200, 150, 200, 200, 345, 15, $green,
        IMG_ARC_EDGED | IMG_ARC_NOFILL);
```

If we use those to replace the first and third calls from the previous script, they should make the righthand arc become a straight line on the outside edge of the arc, and make the lefthand arc become an unfilled wedge. This is pictured in Figure 16-4.

So far, we've only been looking at the filled shapes, but there are unfilled varieties too: imageellipse() complements imagefilledellipse(), imagearc() complements imagefilledarc(), and imagerectangle() complements imagefilledrectangle(). The first and last of these work the same, whether they are filled or otherwise, but imagefilledarc() is slightly different—you don't need the last parameter, because the arc is always the equivalent of IMG_ARC_NOFILL.

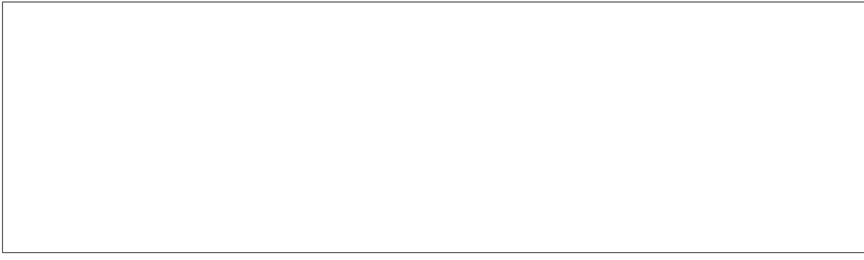


Figure 16-4. Now we've tweaked the last parameter to imagefilledarc() for the first and third calls

Complex Shapes

Rectangles, ellipses, and arcs are inherently easy to use because they have predefined shapes, whereas polygons are multisided shapes of arbitrary geometry and are more complicated to define.

The parameter list is straightforward and the same for both `imagefilledpolygon()` and `imagepolygon()`: the image resource to draw on, an array of points to draw, the number of total points, and the color. The array is made up of pairs of X,Y pixel positions. PHP uses these coordinates sequentially, drawing lines from the first (X,Y) to the second, to the third, etc., until drawing a line back from the last one to the first.

The easiest thing to draw is a square, and we can emulate the functionality of `imagefilledrectangle()` like this:

```
$points = array(
        20, // x1, top-left
        20, // y1

        230, // x2, top-right
        20, // y2

        230, // x3, bottom-right
        230, // y3

        20, // x4, bottom-left
        230 // y4
);

$image = imagecreatetruecolor(250, 250);
$green = imagecolorallocate($image, 0, 255, 0);
imagefilledpolygon($image, $points, 4, $green );

header('Content-type: image/png');
imagepng($image);
imagedestroy($image);
```

I have added extra whitespace in there to make it quite clear how the points work in the `$points` array—see Figure 16-5 for how this code looks in action. For more advanced polygons, try writing a function that generates the points for you.

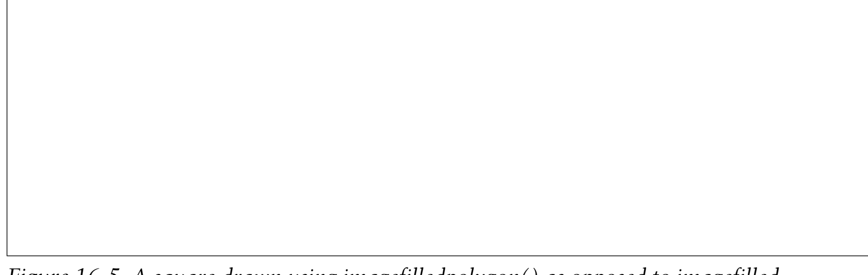


Figure 16-5. A square drawn using imagefilledpolygon() as opposed to imagefilled-rectangle()—as long as you get the numbers right, it should look exactly the same

PHP draws the polygon by iterating sequentially through the points array, and if your shape crosses itself, it is interpreted as a hole in the polygon. If you re-cross the hole, it becomes filled again, and so on.

Outputting Text

To output text using PHP, you first need fonts. PHP allows you to use TrueType (TTF) fonts, PostScript Type 1 (PS) fonts, or FreeType 2 fonts, with TTF tending to be the most popular, due to the availability of fonts. If you are running Windows, you probably have at least 20 TTF fonts already installed that you can use—check in the "Fonts" subdirectory of your Windows directory to see what is available. Many Unix distributions come with TTF fonts installed also—either check in */usr/share/fonts/truetype*, or run a search for them. Alternatively, if you have a Windows CD around, you can borrow some from there. Some distributions (including Debian and SUSE) allow you to install Microsoft's Core Fonts for the Web. The Free Software Foundation has a set of free fonts that you can grab from its web site.

For this next example, I used the font Arial, which is stored in the same directory as my PHP script. Save this code as *addingtext.php*:

```
$image = imagecreate(400,300);
$blue = imagecolorallocate($image, 0, 0, 255);
$white = ImageColorAllocate($image, 255,255,255);

if(!isset($_GET['size'])) $_GET['size'] = 44;
if(!isset($_GET['text'])) $_GET['text'] = "Hello, world!";

imagettftext($image, $_GET['size'], 15, 50, 200, $white,
        "ARIAL", $_GET['text']);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

The two `isset()` lines in that example are there to make sure there is a default font size, 44, and default text, "Hello, world!" for our image. These are set only if you do not pass values using *addingtext.php?size=26&text=Foobarbaz*.

Next comes the important function, `imagettftext()`, which takes eight parameters in total: the image resource to draw on, font size to use, angle to draw at, X coordinate, Y coordinate, color, font file, and the text to write. A few of those parameters are the same as parameters we've used in other functions, but font size in points, angle, name of font, and the text to print are all new. The X and Y coordinates might fool you at first, because they should be set to the position in which you want the lower-left corner of the first character to appear.

The angle parameter works almost in the same manner as the angle parameters used in `imagefilledarc()`, with the difference being that it works in the opposite direction—the angles in `imagefilledarc()` work in a clockwise direction from 3 o'clock, whereas `imagettftext()` works counter-clockwise. That is, specifying 15 as the angle will make the text rotate 15 degrees so that it slants upward.

The font name parameter needs to point to the TTF file you want to use. If this filename does not begin with /, PHP will automatically add *.ttf* to the end and search locally. On Unix machines, you may find that PHP searches in */usr/share/fonts/truetype*. As you can see in the example, "ARIAL" is specified, so *ARIAL.TTF* will be loaded and used for printing the text.

The final parameter for the function is the text to print, and you should be sure to specify any new lines as `\n\r`, not one or the other. You may find that certain fonts do not have various special characters—in this situation, you will see empty boxes drawn rather than the special characters.

The output from this script is shown in Figure 16-6.

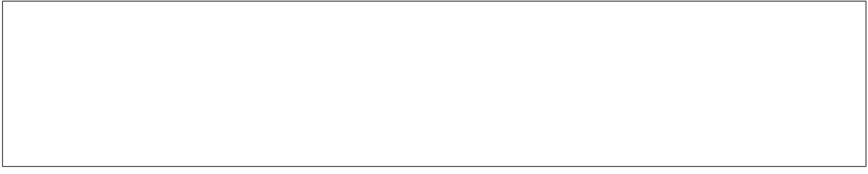


Figure 16-6. Any TrueType font at any size, any angle, and any color—all through one easy function

If you do not want your text to be anti-aliased (smooth-edged), put a minus sign before your color, e.g., `-$white`.

Fitting text into an exact space is a complex art, particularly when you rotate the text too. PHP makes the job easier with the function `imagettfbbox()`, which will return an array containing the coordinates of a bounding box around the text—literally, how big it is in each of its dimensions. The complication here is that it is tricky to get the coordinate system right, as the numbers returned seem easier to use than they actually are.

To call `imagettfbbox()`, you need to pass in four parameters: font point size, rotation angle, font name, and text string to measure. This is essentially a cut-down version of `imagettftext()`, so you can just copy your existing call to that and remove the unnecessary parameters.

What you will get back is an array of eight elements, which are shown in Table 16-1.

Table 16-1. The eight elements in the array returned by imagettfbox()

0	Lower-left corner, X coordinate
1	Lower-left corner, Y coordinate
2	Lower-right corner, X coordinate
3	Lower-right corner, Y coordinate
4	Upper-right corner, X coordinate
5	Upper-right corner, Y coordinate
6	Upper-left corner, X coordinate
7	Upper-left corner, Y coordinate

Each of those coordinates are relative to the text itself, viewed horizontally. That is, although 0 should be the lower-left corner of our first letter, it's unlikely that either the lower-left X or the lower-left Y will be 0, particularly if your text is rotated. For example, in our previous example we rotated text 15 degrees counter-clockwise, which would put the lower-left corner of our rotated text to the right and above the lower-left corner of the horizontal text. Add to that the fact that the numbers are frequently a little off, especially if you use large fonts, and you should be ready for problems!

However, if you are not rotating your text, or if you are rotating only a little (under about 20 degrees), you are not likely to encounter any problems, and you can use a fairly simple script like this next one to get your image fitting your text closely:

```
if(!isset($_GET['size'])) $_GET['size'] = 44;
if(!isset($_GET['text'])) $_GET['text'] = "Hello, world!";

$size = imagettfbbox($_GET['size'], 0, "ARIAL", $_GET['text']);
$xsize = abs($size[0]) + abs($size[2]);
$ysize = abs($size[5]) + abs($size[1]);

$image = imagecreate($xsize, $ysize);
$blue = imagecolorallocate($image, 0, 0, 255);
$white = ImageColorAllocate($image, 255,255,255);
imagettftext($image, $_GET['size'], 0, abs($size[0]), abs($size[5]), $white,
        "ARIAL", $_GET['text']);

header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

Note the use of the `abs( )` function to convert negative numbers to positive. The value `abs($size['5'])` is used as the Y coordinate for the text because `imagettfbbox( )` returns its values from the lower-left corner of the baseline of the text string, not the absolute lower-left corner. The baseline of a letter is where it

would sit if you were handwriting it on lined paper—for example, the letter “a” sits on the line, whereas the letter “y” sits below the line, with the “v” part of the letter resting on the baseline. The baseline problem is illustrated in Figures 16-7 and 16-8.

Figure 16-7. This text uses the image height to align the text to the bottom of the picture; note how the “g” in “sitting” is cut off because it falls below the baseline

Figure 16-8. This text aligns to the top of the picture, as our code does, so that the baseline is no longer right at the bottom and the “g” is fully visible

Loading Existing Images

Some of the best ways to use the image functions in PHP are with existing images. For example, you can write a script to dynamically create buttons by first loading a blank button image from your hard drive and overlaying text on top. Loading images takes the form of a call to imagecreatefrom*(), where the * is *png*, *jpeg*, or various other formats. These functions take just one parameter, which is the file to load, and return an image resource for use as we’ve been doing already.

The first step in creating a customizable button script is to create a blank button (as in Figure 16-9) using the art package of your choice.

Figure 16-9. A blank button saved in PNG format is easy to load into PHP for dynamic modification

Adding text to this button is largely the same as our existing text code, with a few minor changes:

- The `$blue` color is no longer needed, and we will not be using `imagecreate()`.
- We need to center the text in the middle of the button.
- The font size needs to come down a little in order to fit the button.

With that in mind, here’s the new script:

```
if(!isset($_GET['size'])) $_GET['size'] = 26;
if(!isset($_GET['text'])) $_GET['text'] = "Button text";

$size = imagettfbbox($_GET['size'], 0, "ARIAL", $_GET['text']);
$xsize = abs($size[0]) + abs($size[2]);
$ysize = abs($size[5]) + abs($size[1]);
```

```
$image = imagecreatefrompng("button.png");
$imagesize = getimagesize("button.png");
$textleftpos = round(($imagesize[0] - $xsize) / 2);
$texttoppos = round(($imagesize[1] + $ysize) / 2);
$white = ImageColorAllocate($image, 255,255,255);

imagettftext($image, $_GET['size'], 0, $textleftpos, $texttoppos, $white,
"ARIAL", $_GET['text']);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

The new function in that script is `getimagesize()`, which returns the width and height of the image specified in its parameter as an array, with elements 0 and 1 being the width and height, respectively. In addition, element 2 is the type of the picture, and will be set to either `IMAGETYPE_BMP`, `IMAGETYPE_GIF`, `IMAGETYPE_JPEG`, `IMAGETYPE_PNG`, `IMAGETYPE_PSD`, `IMAGETYPE_SWF`, among other values. This element is particularly helpful when used with `image_type_to_mime_type()`.

Running that script without any parameters generates the picture shown in Figure 16-10, although you can send "text" and "size" if you want to play around. With this script in place, you can generate a whole toolbar of buttons for a web site using this one script, simply by changing the "text" value you pass in. Of course, it is not very efficient to keep regenerating the same buttons each time a page is loaded, so if I were you, I would save each generated picture as a file named after the text used—that way, you can use `file_exists()` to attempt to load the existing picture and save the extra work.

Figure 16-10. An empty button overlaid with rendered text

With just a little work, we can even add a simple shadow to the text, as shown in Figure 16-11. To do this, allocate a new color for the shadow (such as black), then call `imagettftext()` twice—once for the shadow, and again for the text itself. Offset the shadow by +1 on X and Y, and the text by -1 on X and Y, completing the effect.

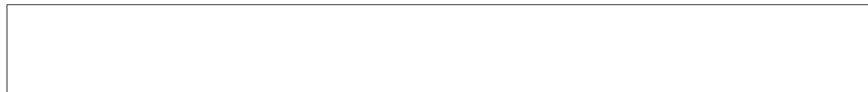


Figure 16-11. Drawing text twice to get a shadow

Color and Image Fills

The function `imagefill()` takes four parameters: an image resource, the X and Y coordinates to start the fill at, and the color with which to fill. The fill will automatically flood your image with color outward from the point specified by your X and Y parameters until it encounters any other color.

Put this `imagefill()` function call into your *addingtext.php* script, just after `imagettftext()`:

```
$red = imagecolorallocate($image, 255, 0, 0);
imagefill($image, 0, 0, $red);
```

With that function, our red color is used to fill in the image starting from (0,0), which is the top-left corner. If you load the script into your web browser, you will see the fill has left some parts of the blue behind—the parts it couldn't "reach" inside the text. Also, you will notice there is a bluish fringe around the text, where the white text was anti-aliased (smoothed) against the blue background, producing a blue-white edge to the text. Figure 16-12 shows how the fill looks with the blue areas that could not be reached inside letters. Figure 16-13 shows a close-up of the letter "o," where you can see the anti-aliasing in action. As our fill starts on blue, it will not fill over any other shade of blue, which is why this fringe has been left there.

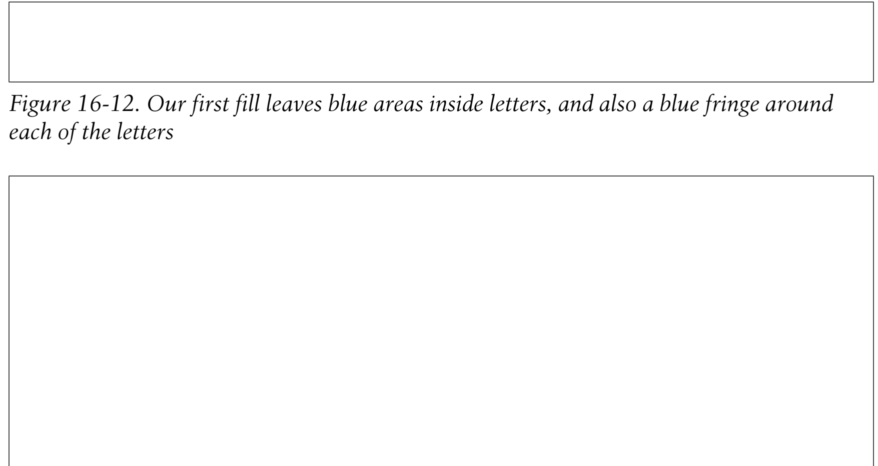


Figure 16-12. Our first fill leaves blue areas inside letters, and also a blue fringe around each of the letters

Figure 16-13. Anti-aliasing has made PHP blend the blue and white together on the edges of the letters to get a smooth effect—our fill leaves these intact

There is a similar function, `imagefilltoborder()`, where the color to fill is the fifth parameter, and the new fourth parameter is the color at which the fill should stop "flowing." That is, the fill will keep flooding outward until it hits the border color. If we change our `imagefill()` call to `imagefilltoborder()` and specify `$white` as the color at which to stop, it should eliminate the anti-aliasing fringe around the letters. Replace the `imagefill()` call with this:

```
imagefilltoborder($image, 0, 0, $white, $red);
```

Whereas the `imagefill()` function will fill the image with color until it encounters any other color, the `imagefilltoborder()` function call shown above will fill the image with color and continue until it finds pixels colored with `$white`. When you look at it in your browser, you will notice the text has become very jagged, because our red fill has taken away all the blue-white smoothing.

The `imagesettile()` function allows you to use an existing image as the picture for your fill in place of a color, which PHP will tile across your image as it fills. This function takes just two parameters: the image you want to change and the image to use as a tile fill.

In order to use a tiled image for your fills rather than a color, pass the constant `IMG_COLOR_TILED` where you would usually pass a color. Thus, we can alter the *addingtext.php* script to look like this:

```
if(!isset($_GET['size'])) $_GET['size'] = 44;
if(!isset($_GET['text'])) $_GET['text'] = "Hello, world!";
$size = imagettfbbox($_GET['size'], 0, "ARIAL", $_GET['text']);
$xsize = abs($size[0]) + abs($size[2]);
$ysize = abs($size[5]) + abs($size[1]);

$image = imagecreate($xsize, $ysize);
$blue = imagecolorallocate($image, 0, 0, 255);
$white = ImageColorAllocate($image, 255,255,255);
imagettftext($image, $_GET['size'], 0, abs($size[0]), $ysize, $white,
"ARIAL", $_GET['text']);

$bg = imagecreatefrompng("button_mini.png");
imagesettile($image, $bg);
imagefill($image, 0, 0, IMG_COLOR_TILED);
header("content-type: image/png");

imagepng($image);
imagedestroy($image);
imagedestroy($bg);
```

You can use `imagesettile()` as many times as you need in order to do several fills using different images. As an added bonus, once you have used `imagesettile()`, you can also use `IMG_COLOR_TILED` wherever you create filled shapes—just use it in place of the color and you can create tiled polygons, ellipses, and other shapes.

Adding Transparency

Specifying the part of an image that should be transparent is as simple as picking the color to use as transparent and passing it into the `imagecolortransparent()` function. As the support for transparency in some browsers (notably with Internet Explorer and PNG transparency) is limited, this function is most useful when the transparent image is used as part of a larger image so that the transparency can be seen.

```
$image = imagecreatetruecolor(400,400);

$black = imagecolorallocate($image, 0, 0, 0);
imagecolortransparent($image, $black);

/// rest of picture here
```

Why JPEGs Don't Support Transparency

JPEGs do not support transparency and will likely never do so. This is because both methods of transparency—color selection and alpha channels—are unsuitable for the JPEG format.

The first is impossible because JPEGs do not guarantee exact color matching, which means that a color you expect to be transparent may end up not. The second is because alpha channels usually have large blocks of transparency followed by a quick change to non-transparency—something that JPEG handles very badly, because it relies on smooth changes in colors to compress well.

Using Brushes

In the same way that `imagesettile()` allows you to use a picture for filling, `imagesetbrush()` allows you to use a picture for an outline. While this could be a premade picture you've just loaded, you can get nice effects by using handmade pictures that are swept around basic shapes.

Figure 16-14 shows a picture of a lot of dots ranging in color from red to yellow—not very interesting, but great for using as a brush.

Figure 16-14. The picture we'll be using as our brush

Those dots were created with this script:

```
$brush = imagecreate(100,100);

$brushtrans = imagecolorallocate($brush, 0, 0, 0);
imagecolortransparent($brush, $brushtrans);

for ($k = 1; $k < 18; ++$k) {
        $color = imagecolorallocate($brush, 255, $k * 15, 0);
        imagefilledellipse($brush, $k * 5, $k * 5, 5, 5, $color);
}

imagepng($brush);
imagedestroy($brush);
```

The next step is to create a larger image, recreate that brush, and use it as the outline for a shape. Here's the code:

```
$pic = imagecreatetruecolor(600,600);
$brush = imagecreate(100,100);

$brushtrans = imagecolorallocate($brush, 0, 0, 0);
imagecolortransparent($brush, $brushtrans);
```

```
for ($k = 1; $k < 18; ++$k) {
        $color = imagecolorallocate($brush, 255, $k * 15, 0);
        imagefilledellipse($brush, $k * 5, $k * 5, 5, 5, $color);
}

imagesetbrush($pic, $brush);
imageellipse($pic, 300, 300, 350, 350, IMG_COLOR_BRUSHED);

imagepng($pic);
imagedestroy($pic);
imagedestroy($brush);
```

The new line in there is the call to `imagesetbrush( )`—note that it takes the image you're changing as the first parameter, and the brush to use as the second. To actually use the brush that has been set, we need to pass the special constant `IMG_COLOR_BRUSHED` as the color parameter for our shape.

That's pretty much it. The only other thing is the call to `imagecolortransparent( )`, which is there so that the black part of the brush (most of it!) doesn't overlay itself.

The result of that script is shown Figure 16-15—not bad for such a simple script, particularly as only one ellipse is actually drawn in the code.

Figure 16-15. Drawing an ellipse with our dots gives us a brightly colored Mobius strip

Once you've used your brush, you can change it for something else, and do so as many times as you want. Figure 16-16 shows the output of this next script, which uses ellipses drawn several times in different colors by re-creating the brush as necessary:

```
$pic = imagecreatetruecolor(400,400);

$bluecol = 0;
```

```
for ($i = -10; $i < 410; $i += 80) {
        for ($j = -10; $j < 410; $j += 80) {
                $brush = imagecreate(100,100);

                $brushtrans = imagecolorallocate($brush, 0, 0, 0);
                imagecolortransparent($brush, $brushtrans);

                for ($k = 1; $k < 18; ++$k) {
                        $color = imagecolorallocate($brush, 255,
                                $k * 15, $bluecol);
                        imagefilledellipse($brush, $k * 2, $k * 2,
                                1, 1, $color);
                }

                imagesetbrush($pic, $brush);
                imageellipse($pic, $i, $j, 50, 50, IMG_COLOR_BRUSHED);

                imagedestroy($brush);
        }

        $bluecol += 40;
}

imagepng($pic);
imagedestroy($pic);
```

Figure 16-16. Many dots, many ellipses, and many colors: iteration in action

Basic Image Copying

The two functions `imagecopy()` and `imagecopymerge()` are similar in that they copy one picture into another. Both of their first eight parameters are identical:

- The destination image you're copying to
- The source image you're copying from

- The X coordinate you want to copy to
- The Y coordinate you want to copy to
- The X coordinate you want to copy from
- The Y coordinate you want to copy from
- The width in pixels of the source image you want to copy
- The height in pixels of the source image you want to copy

Parameters three and four allow you to position the source image where you want it on the destination image, and parameters five, six, seven, and eight allow you to define the rectangular area of the source image that you want to copy. Most of the time, you will want to leave parameters five and six at 0 (copy from the top-left corner of the image), and parameters seven and eight at the width of the source image (the bottom-right corner of it) so that it copies the entire source image.

The way these functions differ is in the last parameter: `imagecopy( )` always overwrites all the pixels in the destination with those of the source, whereas `imagecopymerge( )` merges the destination pixels with the source pixels by the amount specified in the extra parameter: 0 means "Keep the source picture fully," 100 means "Overwrite with the source picture fully," and 50 means "Mix the source and destination pixel colors equally." The `imagecopy( )` function is therefore equivalent to calling `imagecopymerge( )` and passing in 100 as the last parameter.

Figures 16-17 and 16-18 show two input images that will be used to test these functions.

Figure 16-17. Our source picture: some stars

Now, to get those two to merge, we need a script like this one:

```
$stars = imagecreatefrompng("stars.png");
$gradient = imagecreatefrompng("gradient.png");
imagecopymerge($stars, $gradient, 0, 0, 0, 0, 256, 256, 60);
header('Content-type: image/png');
imagepng($stars);
imagedestroy($stars);
imagedestroy($gradient);
```

That merges the two at 60%, which gives slightly more prominence to the gradient. The result is shown in Figure 16-19.

Figure 16-18. Our destination picture: a smooth, blue gradient

Figure 16-19. Stars + gradient + some imagination = the night sky

Scaling and Rotating

PHP offers you two different ways to resize an image, and you should choose the right one for your needs. The first option, `imagecopyresized()`, allows you to change the size of an image quickly but has the downside of producing fairly low-quality pictures. When an image with detail is resized, aliasing ("jaggies") is usually visible, which makes the resized version hard to read, particularly if the resizing was to an unusual size. The other option is `imagecopyresampled()`, which takes the same parameters as `imagecopyresized()` and works in the same way, with the exception that the resized image is smoothed so that it is still visible. The downside here is that the smoothing takes more CPU effort, so the image takes longer to produce.

Here is an example of `imagecopyresized()` in action— save it as *specialeffects.php*:

```
header("content-type: image/png");
$src_img = imagecreatefrompng("complicated.png");
$srcsize = getimagesize("complicated.png");
$dest_x = $srcsize[0] / 1.5;
$dest_y = $srcsize[1] / 1.5;
$dst_img = imagecreatetruecolor($dest_x, $dest_y);

imagecopyresized($dst_img, $src_img, 0, 0, 0, 0,
        $dest_x, $dest_y, $srcsize[0], $srcsize[1]);
imagepng($dst_img);
```

```
imagedestroy($src_img);
imagedestroy($dst_img);
```

There are two images being used in there. The first one, `$src_img`, is created from a PNG screenshot of the online PHP manual—this contains lots of text, which highlights the aliasing problem with `imagecopyresized( )` nicely. The variables `$dest_x` and `$dest_y` are set to be the width and height of *complicated.png* divided by 1.5, which will set the destination size to be 66% of the source size. Resizing "exact" values such as 10%, 50%, etc., usually looks better than resizing unusual values such as 66%, 79%, etc.

The second image is then created using `imagecreatetruecolor( )` and our destination sizes, and is stored in `$dst_img`. Now comes the key part: `imagecopyresized( )` takes quite a few variables, and you needn't bother memorizing them. They are, in order, the image to copy to, image to copy from, destination X coordinate, destination Y coordinate, source X coordinate, source Y coordinate, destination width, destination height, source width, and source height. Parameters three to six, the coordinates, allow you to copy regions of the picture as opposed to the whole picture—PHP will copy from the specified coordinate to the end of the picture, so by passing in 0, we're using the entire picture. You probably will not ever want to copy regions using these parameters, so just leave them as 0.

Take a screenshot of a web site of your choosing and save it as *complicated.png* in the same directory as your PHP script, then load up *specialeffects.php* in your browser. All being well, you should see something similar to Figure 16-20—the web site picture has been resized down, but as a result, all the text is hard—if not impossible—to read.

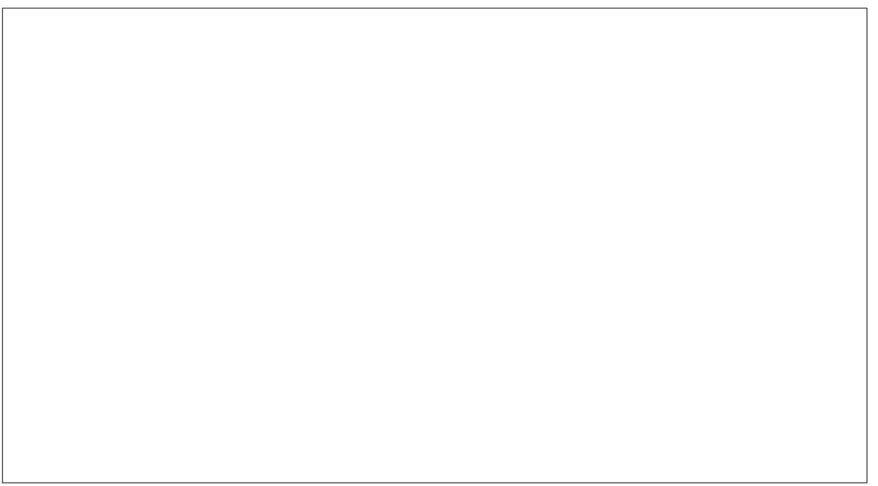
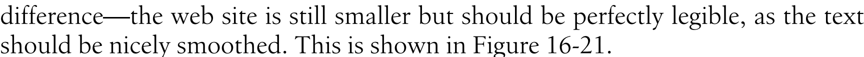


Figure 16-20. Using imagecopyresized() on a picture is fast, but produces low-quality results

Now, to give you an idea why `imagecopyresampled( )` is better, change the `imagecopyresized( )` call to `imagecopyresampled( )`. The parameter list is identical, so just change the function name. This time, you should see a marked



difference—the web site is still smaller but should be perfectly legible, as the text should be nicely smoothed. This is shown in Figure 16-21.

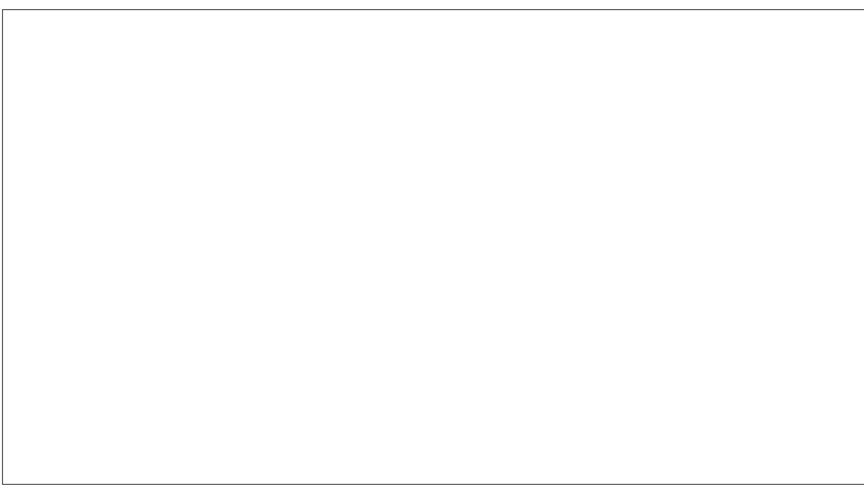


Figure 16-21. Using imagecopyresampled() gives a superior end result

The final special effect we're going to look at is `imagerotate( )`, which rotates an image. This is much easier to do than resizing and resampling, as it only has three parameters: the image to rotate, the number of degrees counter-clockwise you wish to rotate it, and the color to use wherever space is uncovered. The rotation is performed from the center of the source image, and the destination image will automatically be sized to fit the whole of the rotated image.

The last parameter only really makes sense once you have seen it in action, so try out this script:

```
$image = imagecreatefrompng("button.png");
$hotpink = imagecolorallocate($image, 255, 110, 221);
$rotated_image = imagerotate($image, 50, $hotpink);

header("content-type: image/png");
imagepng($rotated_image);
imagedestroy($image);
imagedestroy($rotated_image);
```

You'll need to put your own file in where I have used *button.png*, but otherwise you should see something like Figure 16-22 when you load the picture in your web browser.

The image has been rotated by 50 degrees, anti-aliased to avoid jagged lines, and resized by the minimum amount so that the outputted picture has just enough space to hold the rotated image. Finally, note that the gaps in the image, effectively the "background," have been colored the hot pink we defined. White is usually preferable, but it would not have been quite so obvious in the screenshot.

Figure 16-22. The button rotated 50 degrees counter-clockwise

Points and Lines

Drawing points is accomplished with the function `imagesetpixel()`, which takes four parameters: the image to draw on, the X and Y coordinates, and the color to use. Thus, you can use it like this:

```
$width = 255;
$height = 255;
$image = imagecreatetruecolor($width, $height);

for ($i = 0; $i <= $width; ++$i) {
        for ($j = 0; $j <= $height; ++$j) {
                $col = imagecolorallocate($image, 255, $i, $j);
                imagesetpixel($image, $i, $j, $col);
        }
}

header("Content-type: image/png");
imagepng($image);
imagedestroy($image);
```

In that example, there are two loops to handle setting the green and blue parameters with `imagecolorallocate()`, with red always being set to 255. This color is then used to set the relevant pixel to the newly allocated color, which should give you a smooth gradient like the one in Figure 16-23.

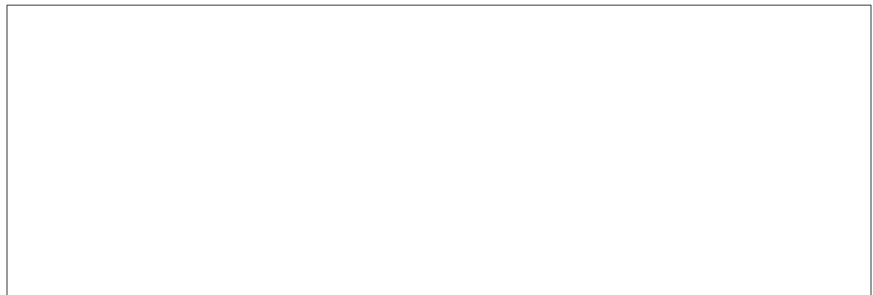


Figure 16-23. Smooth gradiants using per-pixel coloring

Drawing lines is only a little more difficult than individual pixels, and is handled by the `imageline( )` function. This time, the parameters are the image to draw on, the X and Y coordinates of the start of the line, the X and Y coordinates of the end of the line, and the color to use for drawing. We can extend our pixel script to draw a grid over the gradient by looping from 0 to `$width` and `$height`, incrementing by 15 each time, and drawing a line at the appropriate place. `$width` and `$height` were both set to 241 in the previous script because that is 255 - 15 + 1, which means it is the largest grid we can draw using the stock 0–255 color range. The +1 is necessary because drawing a line on the 255th row of the picture would be invisible—it would be outside!

Add these lines before the `header( )` call:

```
for ($i = 0; $i <= $width; $i += 15) {
        imageline($image, $i, 0, $i, 255, $black);
}

for ($i = 0; $i <= $height; $i += 15) {
        imageline($image, 0, $i, 255, $i, $black);
}
```

The first loop draws the vertical lines, so the X coordinate increments by 15 with each loop, whereas the Y coordinates are always 0 and 255, or from the very top to the very bottom. The second loop does the same for the horizontal lines, so this time it is the Y coordinates that change.

To get the script to work, you will also need to add this line after the call to `imagecreatetruecolor( )`:

```
$black = imagecolorallocate($image, 0, 0, 0);
```

The output from that script should generate the picture shown in Figure 16-24.

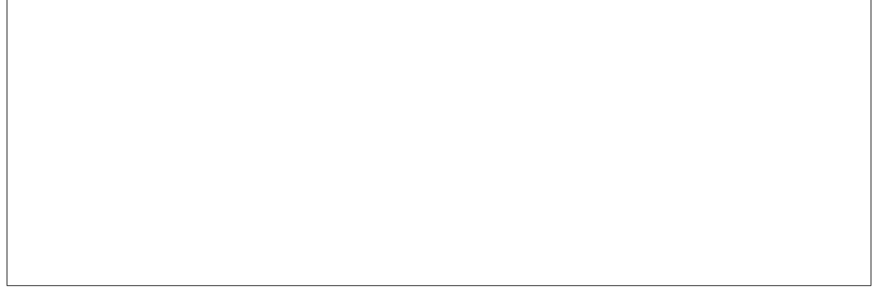
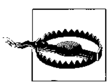


Figure 16-24. Grid lines created with imageline() and loops

The `imagesetthickness( )` function allows you to specify the width in pixels of all lines drawn. All lines drawn using `imageline( )` are affected, but it also affects rectangles, arcs, etc. To use the function, pass in the image to alter as parameter one, and the width in pixels as parameter two, then simply draw lines. The new thickness remains in place until you change it again or destroy the image.

Special Effects Using imagefilter( )



The filters described here were written for the PHP-bundled build of GD, and may not be available in other releases.

The best way to explain this function is to describe how it works, then show a code example. Although the function accepts different numbers of parameters that do very different things, the function returns `true` if the filter was applied successfully and `false` otherwise.

First up is `IMG_FILTER_BRIGHTNESS`, which takes a number between -255 and 255 that represents how much you want to brighten or darken the image. Setting it to 0 leaves the picture unchanged, 255 sets it to full white (brightest), and -255 sets it to full black (darkest). Most pictures tend to look almost invisible beyond +200 or -200.

This code example will lighten our space picture just a little:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_BRIGHTNESS, 50);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

Next up is `IMG_FILTER_COLORIZE`, which takes three parameters between -255 and 255 that respectively represent the red, green, and blue values you want to add or subtract from the image. Setting the blue value to -255 will take all the blue out of all the pixels in the image, whereas setting the red to 128 will add red to them. Setting all three of them to 128 will have the effect of adding white to the picture, brightening it in the same way as `IMG_FILTER_BRIGHTNESS`.

This code example will make our image look more magenta:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_COLORIZE, 100, 0, 100);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

Moving on, the `IMG_FILTER_CONTRAST` filter allows you to change the contrast of the image, and takes just one parameter for a contrast value between -255 and 255. Lower values increase the contrast of the picture, essentially reducing the number of colors so that they are more separate and obvious to the eye. Using positive values brings the colors closer together by mixing them with gray until, at 255, you have a full-gray picture.

This code example shows how even a small positive number makes quite a difference to the resulting image:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_CONTRAST, 20);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

The IMG_FILTER_EDGEDETECT and IMG_FILTER_EMBOSS filters make all the edges in your picture stand out as if they were embossed, and sets everything else to gray. No parameters are needed for either of them, so using them is quite easy.

This next script uses edge detection to grab the edges, then embosses them to make the effect more obvious:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_EDGEDETECT);
imagefilter($image, IMG_FILTER_EMBOSS);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

If you want to blur an image, you have a choice of two filters: IMG_FILTER_GAUSSIAN_BLUR and IMG_FILTER_SELECTIVE_BLUR. The latter is a generic blur function, and the former is a classic "out-of-focus lens" technique that often actually enhances images. Neither function requires parameters.

Although they're easy to use, there's no harm showing an example—here are both of them in action. Just comment out the one you don't want to see:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_GAUSSIAN_BLUR);
imagefilter($image, IMG_FILTER_SELECTIVE_BLUR);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

There's a similar filter, IMG_FILTER_SMOOTH, which gives you a little more control over the output. It takes one parameter, but it takes a little explanation! Unlike the other parameters so far, this isn't a value pertaining to how much you'd like to smooth the image. Instead, it's a *weighting* for an image manipulation matrix, and small changes can affect the output massively.

There isn't enough room here to go into a full discussion of what these manipulation matrices are, but suffice to say you can represent many different transformations—from Gaussian blur to edge detection—using a 3×3 numerical matrix, that defines how the colors of the eight pixels surrounding any given pixel (with the pixel itself being the ninth) should have their RGB values changed. With IMG_FILTER_SMOOTH, the parameter you pass is used as the change value for the pixel itself, which means you get to define how much the pixel's own color is used to form its final color.

You're not likely to want values outside of the range -8 to 8, as even one number makes quite a big difference. At about 10, the picture is almost normal, because the original pixel values are given more weight than the combined sum of its neighbors. But you can get some cool effects between -6 to -8.

This code example smooths the picture just a little:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_SMOOTH, 6);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

There are two helpful filters that alter the colors in a simple way, which are `IMG_FILTER_GRAYSCALE` and `IMG_FILTER_NEGATE`. Both take no parameters: the first sets the picture to grayscale, and the second sets it to use negative colors.

This code example changes the picture to grayscale, then flips it to negative colors:

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_GRAYSCALE);
imagefilter($image, IMG_FILTER_NEGATE);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```

Interlacing an Image

Interlacing an image allows users to see parts of it as it loads, and takes different forms depending on the image type. For example, interlaced JPEGs (called "progressive"), GIFs, and PNG files show low-quality versions of the file as they load. In comparison, non-interlaced JPEGs appear line by line. To enable interlacing on your picture, simply call this function with the second parameter set to 1, or set to 0 if you want to disable it.

Interlacing is likely to affect your file size: JPEGs often get smaller when interlaced because progressive JPEGs use a more complicated mathematical formula to compress the picture, whereas PNG files often get larger. Progressive JPEGs are a mixed blessing, however: Internet Explorer doesn't handle them properly, and rather than showing low-quality versions of the JPEG as it loads, it simply downloads the entire picture and shows it all at once. As a result, non-progressive JPEGs (line by line) appear to load faster on Internet Explorer. Other browsers don't display this problem.

This example shows interlacing in action for PNG files. It's not likely to be very noticeable if you run this on a local web server and/or use small files, because it will be decompressed too fast.

```
$image = imagecreatefrompng("space.png");
imagefilter($image, IMG_FILTER_MEAN_REMOVAL);
imageinterlace($image, 1);
header("content-type: image/png");
imagepng($image);
imagedestroy($image);
```


Getting an Image's MIME Type

So far we have been handcrafting the `header()` function call in each of the image scripts, but many people find MIME types hard to remember and/or clumsy to use. If you fit into this category, you should be using the `image_type_to_mime_type()` function, as it takes a constant as its only parameter and returns the MIME type string. For example, passing in `IMAGETYPE_GIF` will return `image/gif`, passing in `IMAGETYPE_JPEG` will return `image/jpeg`, and passing in `IMAGETYPE_PNG` will return `image/png`.

If you think these constants sound as hard to remember as the MIME types, you're probably right. However, a while back we looked at the getimagesize( ) function, and I mentioned that the third element in the array returned by that function is the type of file it is. These two functions both use the same constant, which means you can use getimagesize( ) and pass the third element into image_type_to_mime_type( ) to have it get the appropriate MIME type for your image—no memorization of constants required.

```
$info = getimagesize("button.png");
print image_type_to_mime_type($info[2]);
```