Introduction To PHP Image Functions

by Jeff Knight of New York PHP

More presentations and
PHundamentals Articles are available.

LAST NEXT

RTFM & YMMV

The PHP Image functions read, manipulate, and output image streams by accessing functions provided by the external GD library (http://www.boutell.com/gd/), which in turn, requires a variety of other libraries to support a number of image formats & font systems.

Since PHP 4.3 the GD library is bundled and installed by default.

In previous versions, to enable GD-support configure PHP `--with-gd[=DIR]`, where DIR is the GD base install directory. To use the recommended bundled version of the GD library (which was first bundled in PHP 4.3.0), use the configure option `--with-gd`. In Windows, you'll include the GD2 DLL PHP_gd2.dll as an extension in PHP.ini.

You can find a good, detailed guide to compiling and enabling GD in PHP by Marco Tabini in the OnLamp.com PHP DevCenter.

array **gd_info** (void)
Returns an associative array describing the version and capabilities of the installed GD library.

```
<?PHP print_r(gd_info()); ?>

Array
(
    [GD Version] => bundled (2.0.34 compatible)
    [FreeType Support] => 1
    [FreeType Linkage] => with freetype
    [T1Lib Support] =>
    [GIF Read Support] => 1
    [GIF Create Support] => 1
    [JPG Support] => 1
    [PNG Support] => 1
    [WBMP Support] => 1
    [XPM Support] =>
    [XBM Support] => 1
    [JIS-mapped Japanese Font Support] =>
)
```

int **imageTypes** (void)
Returns a bit-field corresponding to the image formats supported by the version of GD linked into PHP. The following bits are returned:
IMG_GIF | IMG_JPG | IMG_PNG | IMG_WBMP.

```
<?PHP
if (imagetypes() & IMG_GIF) echo "GIF Support is enabled<br />";
if (imagetypes() & IMG_JPG) echo "JPEG Support is enabled<br />";
if (imagetypes() & IMG_PNG) echo "PNG Support is enabled<br />";
if (imagetypes() & IMG_WBMP) echo "WBMP Support is enabled<br />";
?>
```

```
GIF Support is enabled
JPEG Support is enabled
PNG Support is enabled
WBMP Support is enabled
```

LAST NEXT

Without GD

The PHP Image functions also include several functions that do not require the GD libraries:

array **getImageSize** (string filename [, array imageInfo])
Will determine the size of any GIF, JPG, PNG, SWF, SWC, PSD, TIFF, BMP, IFF, JP2, JPX, JB2, JPC, XBM, or WBMP image file, the dimensions, the file type (correspond to the IMAGETYPE constants), a height/width text string to be used inside a normal HTML IMG tag, the number of bits per channel, and the mime type.

The optional *imageInfo* parameter allows you to extract some extended information from the image file. Currently, this will return the different JPG APP markers as an associative array. Some programs use these APP markers to embed text information in images, such as EXIF data in the APP1 marker and IPTC information in the APP13 marker.

```
<?
PHP print_r ( getImageSize ('http://www.nyphp.org/img/new_york_php.gif' ) ); ?
>

Array
(
    [0] => 165
    [1] => 145
    [2] => 1
    [3] => width="165" height="145"
    [bits] => 7
    [channels] => 3
    [mime] => image/gif
)
```

string **image_type_to_mime_type** (int imageType)
Returns the Mime-Type for an IMAGETYPE constant.

EXIF stands for Exchangeable Image File Format, and is a standard for storing interchange information in image files such as those generated by digital cameras. A set of EXIF functions are available if PHP has been compiled with `--enable-exif`, they do not require the GD library. For more information see ***Chapter 27: EXIF***.

IPTC JPEG File Information is a standard by the International Press Telecommunications Council, in wide use by news agencies and professional photographers. PHP can read and write this data. For more information see ***Chapter 28: IPTC***.

You can also access the ImageMagick libraries using the **`system()`** function, but that's another presentation.

LAST Table of Contents Function Index NEXT

LAST NEXT

What About GIF?

From the GD Manual & GD Library FAQ

Update: I do expect to reintroduce GIF support after July 7th, 2004, however due to a family emergency this may not be immediate. Thanks for your patience.

[author's note: I'll update this site as soon as support makes its way into the bundled version, but since I'm not involved in any of those efforts, I can't give a realistic estimate of when that might be.]

GD 2 creates PNG, JPEG and WBMP images, not GIF images. This is a good thing. Please do not ask us to send you the old GIF version of gd. Unisys holds a patent on the LZW compression algorithm, which is used in fully compressed GIF images. The best solution is to move to legally unencumbered, well-compressed, modern image formats such as PNG and JPEG as soon as possible.

Many have asked whether GD will support creating GIF files again, since we have passed June 20th, 2003, when the well-known Unisys LZW patent expired in the US. Although this patent has expired in the United States, this patent does not expire for another year in the rest of the world. GIF creation will not reappear in GD until the patent expires **world-wide** on July 7th, **2004**.

Now...

The legal issue with GIF files is the LZW compression algorithm, not the file structure itself. Toshio Kuratomi wrote libungif, a modified version Eric S. Raymond's libgif GIF encoder that saves GIFs uncompressed, thus completely avoiding any licensing issues. The obvious problem is that the uncompressed GIF images will be larger than those encoded using the LZW algorithm.

http://prtr-13.ucsc.edu/~badger/software/libungif.shtml

LAST Table of Contents Function Index NEXT

Color Theory

Some Definitions

Color Model: A method for producing a new color by mixing a small set of standard primary colors.

Channels: another term for the primary colors used in a color model, e.g. Red, Green and Blue in RGB. Interchangeable with the term component.

RGB: The file formats manipulated by the GD library all use the RGB color model, which is an abstract representation of the way television and computer monitors work. In the physical world, adjacent phosphor dots of red, green and blue are stimulated to varying degrees to produce a wide range of color. In the mathematical model, three channels of red, green and blue grids of pixels are given values from zero to 100% that correspond to the levels of red, green and blue light.

Bit-depth: the number of steps between zero and 100% in a color model determined by the number of bits used to store the information. The most common and familiar system uses 8 bits to store the value of each channel, thus giving 256 possible values (from 0-255) for each pixel (just for #!@$s and giggles, these values are often still referred to by their hexadecimal values of 0-FF). In an 8-bit RGB color model, it takes 3 channels of 8 bits each to represent any color in the model, hence the familiar six digit hexadecimal color codes from HTML.

LAST NEXT

An Interactive Example

Additive Color System: a color system based on emitted light (as opposed to reflected). Black is achieved when all channels are set to zero and no light is emitted. When all the channels are on all the way, they combine to produce white (note this exactly the opposite of the more familiar subtractive system of inks, pigments and Play-Doh®). A full spectrum of colors are produced by varying the intensities of the various channels.

red: 0

green: 0

blue: 0

or

hex:

redraw

LAST Table of Contents Function Index NEXT

LAST NEXT

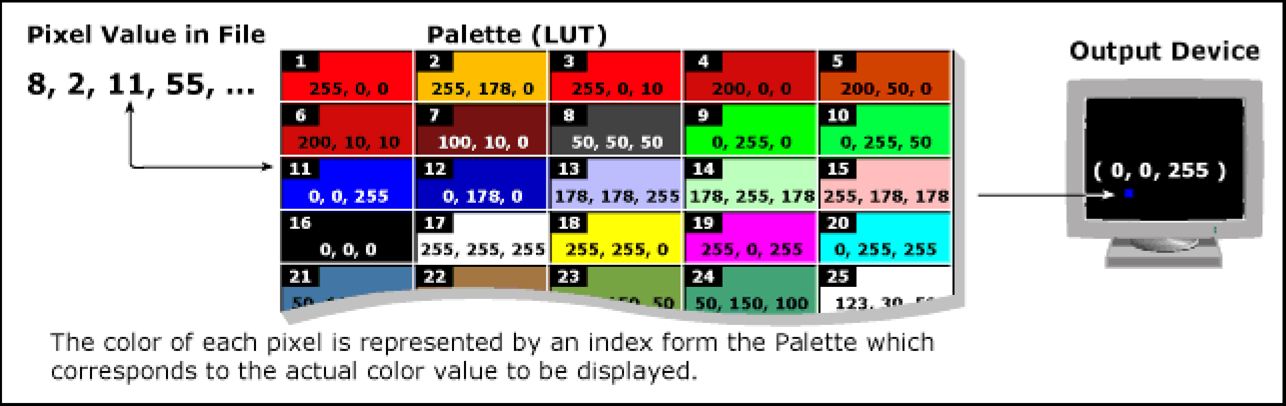
True Color Vs. Indexed Color

True Color

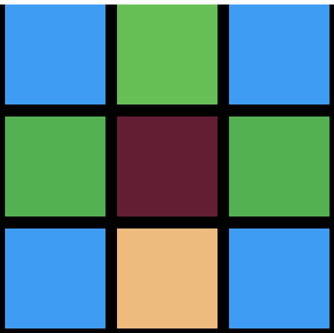
Color models with small bit-depths have gaps of missing colors that can't be represented within their coarse data structure. Models that can represent a large number of colors are known as "true color" or "continuous tone" because they can display images without obvious gaps. The most common true color model is 8-bit RGB, which can be used to represent 2^8 x 2^8 x 2^8, or 16,777,216, colors. When storing color information in a file, each pixel is assigned a byte of color information for each channel, so a 150 x 150 pixel true color image would require 67,500 bytes or about 66k in color data alone. Thus, uncompressed true color image file sizes tend to be quite large.

Specifying Color With Palettes

In order to reduce file sizes, an image with an index color palette uses a 1-dimensional array of color values called a palette, also called a *color map*, *index map*, *color table*, or *look-up table (LUT)*. The color data in the file is stored as a series of small index values which is known as indirect, or pseudo-color storage. Instead of assigning a 3 byte value to each pixel, the index of the color table is used instead. Thus, an 8 bit paletted color file will be 1/3 the size of a true color file (although it can only display a limited 256 colors). Files can be made even smaller by chosing to use fewer colors than 256. Two colors would require only 1 bit per pixel, four 2/px, and eight 3/px, as seen in the example below.

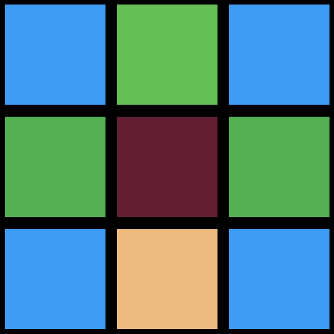


True Color Example:



```
0 0 1 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0 0 1 1 0
0 1 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 0 1 1 1 1 1 0
1 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1
0 0 0 1 0 1 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1 0
0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 1 0 1 1 0 0 0 1 0 1 0 1
0 0 0 1 0 0 1 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0
1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 1 1
1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0
```

Index Color Example:



```
000 => 0 0 1 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0
001 => 0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1
010 => 0 1 0 1 0 1 0 0 1 0 1 1 0 0 0 1 0 1 0 1 0 0 0 1
011 => 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1 0 0 0 1 1 0 0 1 1
100 => 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1
--------------------------------------------------------
0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0
```

Which to Use?

The choice has a dramatic impact on file size and image quality. Click on the image at right to cycle through a 2x magnified sample in true color and indexed to 256 and 32 colors. As a general rule of thumb, you'll want to use true color for photorealistic images and index color for simple graphics like charts & graphs.

Your choice of image file format also has an impact, since JPEG images are always true color, but PNGs can be any one of fifteen color formats including 8 & 16 bit RGB (true color) as well as 1, 2, 4 and 8 bit palettes (indexed).

Image Supplied by FreeFoto.com

LAST Table of Contents Function Index NEXT

The Basics

Let's Get to Some PHP Already!

Image creation in PHP is handled in 5 basic steps:

1. Create a blank image of a specified size in memory.
2. Add content to the blank image.
3. Send the headers.
4. Output the image in the chosen file format.
5. Delete the image from memory.

1. Create the Image:

resource **imageCreate** (int x_size, int y_size)
resource **imageCreateTrueColor** (int x_size, int y_size)
Returns an image identifier representing a blank image of size *x_size* by *y_size*.

Image creation is done with one of two functions `imageCreate()` and `imageCreateTrueColor()`. The latter creates a true color image and the former creates an index color image. Both functions return an integer image identifier and require 2 parameters representing the width and height of the image in pixels. The PHP online manual recommends the use of `imageCreateTrueColor()`, but if you understand its limitations, `imageCreate()` can often be a more appropriate choice.

2. Add content:

Once you have a blank image in memory, there are many copy, transform, draw, and text/font functions to create your masterpiece. These will be covered later in the presentation.

3. Send the Headers:

int **header** (string string [, bool replace [, int http_response_code]])
Used to send raw HTTP headers. See the HTTP/1.1 specification for more information.

Since you're going to be sending image data to the browser instead of HTML, you need to use the header function to send the appropriate content type: image/png, image/jpeg, etc.

4. Output the Image

int **imagePNG** (resource image [, string filename])
Outputs a GD image stream (*image*) in PNG format to standard output (usually the browser) or, if a filename is given by the filename it outputs the image to the file.

---

int **imageJPEG** (resource image [, string filename [, int quality]])
Creates the JPEG file in filename from the *image* image. The *filename* argument is optional, and if left off, the raw image stream will be output directly. *quality* is optional, and ranges from 0 (worst quality, smaller file) to 100 (best quality, biggest file). The default is the default IJG quality value (about 75).

---

int **imageWBMP** (resource image [, string filename [, int foreground]])
int **imageGIF** (resource image [, string filename])
int **imageGD** (resource image [, string filename])
int **imageGD2** (resource image [, string filename [, int chunk_size [, int type]]])

In PHP you manipulate images within memory in GD's native format. The functions listed above are for outputting your image in whatever format you need to use. Like true color vs. indexed, what format you choose will depend upon the particular situation. In general JPEGs are good for true color, photographic images and PNGs are good for indexed color graphics. WBMP, the Wireless Bitmap format, is used for two-color images for WML pages. GIF is similar to PNG, but inferior in every way and has patent trouble, so is best avoided. The GD & GD2 formats are, according to the GD FAQ page "*not* intended for general purpose use and should never be used to distribute images. It is not a compressed format. Its purpose is solely to allow very fast loading of images your program needs often in order to build other images for output."

5. Delete the Image

int **imageDestroy** (resource image)
Frees any memory associated with image *image*. *image* is the image identifier returned by the **imageCreate()** function.

Whether you output the image to a browser, save it to a file, or just give up it will remain in the server's memory until it is destroyed. If you continue to build images without destroying them, the server will crash. Never create an image without destroying it by the end of your script.

LAST NEXT

A Simple Example

simplepng.php

As a first example, consider the PHP file used in the previous True Color/Index Color examples and below. The file is called with a single parameter: the six digit hexadecimal representation of an RGB color, and returns a 50 x 50 pixel PNG set to that color.

<img src="simplepng.php?color=05DFBB">

int **imageColorAllocate** (resource image, int red, int green, int blue)
Returns a color identifier representing the color composed of the given RGB components. The *im* argument is the return from the **imageCreate()** function. *red*, *green* and *blue* are the values of the red, green and blue component of the requested color respectively. These parameters are integers between 0 and 255 or hexadecimals between 0x00 and 0xFF. **imageColorAllocate()** must be called to create each color that is to be used in the image represented by *image*.

In the event that all 256 colors have already been allocated, **imageColorAllocate()** will return -1 to indicate failure.

```
<?PHP


error_reporting(E_ALL ^ E_NOTICE);

/*
simplepng.php
Generates a 50 x 50 pixel png of the color passed in parameter 'color'
*/

// convert guaranteed valid hex value to array of integers
$imColor = hex2int(validHexColor($_REQUEST['color']));

// Step 1. Create a new blank image
$im = imageCreate(50,50);

// Step 2. Set background to 'color'
$background = imageColorAllocate($im, $imColor['r'], $imColor['g'], $imColor['b']);

// Step 3. Send the headers (at last possible time)
header('Content-type: image/png');

// Step 4. Output the image as a PNG
imagePNG($im);

// Step 5. Delete the image from memory
imageDestroy($im);

/**
```

```
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array( 'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                      'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                      'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
                    );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

?>
```

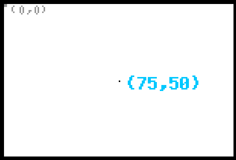
It is a good practice to keep steps 3, 4, and 5 together. If you don't wait until the last possible moment to send the headers, any errors thrown by the code will be interpreted as binary gobbeldygook and you won't see them. It also helps to maintain the important habbit of destorying images by doing it immediately after you output or save them.

LAST Table of Contents Function Index NEXT

The Draw Functions

In order to make a more interesting image you'll use the drawing functions to make lines and shapes. For all functions, the top left is 0,0.

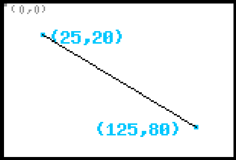
int **imageSetPixel** (resource image, int x, int y, int color)
Draws a pixel at *x*, *y* in image *image* of color *color*.



imageSetPixel ($im, $x , $y , $black) ;

redraw

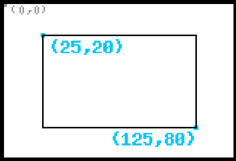
int **imageLine** (resource image, int x1, int y1, int x2, int y2, int color)
Draws a line from *x1*, *y1* to *x2*, *y2* in image *im* of color *color*.



imageLine ($im, $x1 , $y1 , $x2 , $y2 , $black);

redraw

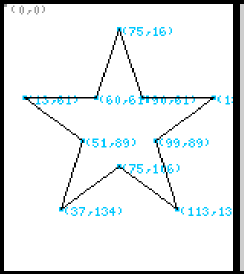
int **imageRectangle** (resource image, int x1, int y1, int x2, int y2, int col)
int **imageFilledRectangle** (resource image, int x1, int y1, int x2, int y2, int color)
Creates a rectangle of color *col* in image *image* starting at upper left coordinate *x1*, *y1* and ending at bottom right coordinate *x2*, *y2*.



imageRectangle ($im $x1 $y1 , $x2 , $y2 , $black) ;

redraw

int **imagePolygon** (resource image, array points, int num_points, int color)
int **imageFilledPolygon** (resource image, array points, int num_points, int color)
Creates a polygon in image *image*. *points* is a PHP array containing the polygon's vertices, ie. points[0] = x0, points[1] = y0, points[2] = x1, points[3] = y1, etc. *num_points* is the total number of points (vertices).

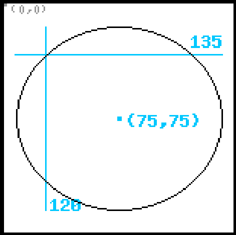


$points = array (13, 61, 60, 61, 75, 16, 90, 61, 137, 61, 99, 89, 113, 134, 75, 106, 37, 134, 51, 89) ;

imagePolygon ($im, $points, (count($points)/2) , $black) ;

redraw

int **imageEllipse** (resource image, int cx, int cy, int w, int h, int color)
int **imageFilledEllipse** (resource image, int cx, int cy, int w, int h, int color)
Draws an ellipse centered at *cx*, *cy* in the image represented by *image*. *W* and *h* specifies the ellipse's width and height respectively. The color of the ellipse is specified by *color*.

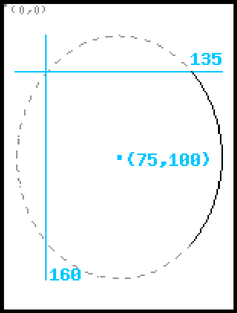


imageEllipse ($im, cx , cy , w , h , $black) ;

redraw

int **imageArc** (resource image, int cx, int cy, int w, int h, int s, int e, int color)
int **imageFilledArc** (resource image, int cx, int cy, int w, int h, int s, int e, int color, int style)
Draws a partial ellipse centered at *cx*, *cy* in the image represented by image. *W* and *h* specifies the ellipse's width and height respectively while the start and end points are specified in degrees indicated by the *s* and *e* arguments. 0° is located at the three-o'clock position, and the arc is drawn counter-clockwise. *style* is a

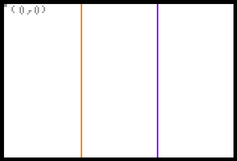
bitwise OR of the following possibilities: **IMG_ARC_CHORD** just connects the starting and ending angles with a straight line, while **IMG_ARC_PIE** produces a rounded edge. **IMG_ARC_NOFILL** indicates that the arc or chord should be outlined, not filled. **IMG_ARC_EDGED**, used together with **IMG_ARC_NOFILL**, indicates that the beginning and ending angles should be connected to the center - this is a good way to outline (rather than fill) a 'pie slice'.



imageArc ($im, $cx , $cy , $w , $h , $s , $e , $black , IMG_ARC_PIE IMG_ARC_CHORD IMG_ARC_EDGED) ;

redraw

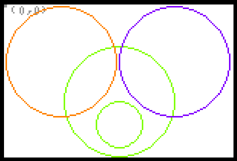
int **imageFill** (resource image, int x, int y, int color)
Performs a flood fill starting at coordinate *x*, *y* with color *color* in the image *image*.



imageFill ($im, $x , $y , $black) ;

redraw

int **imageFillToBorder** (resource image, int x, int y, int border, int color)
Performs a flood fill of color *color* to the border color defined by *border* starting at coordinate *x*, *y* with color *color*.

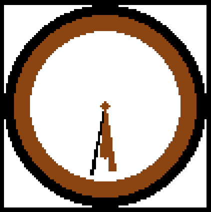


imageFillToBorder ($im, $x , $y , $orange , $black) ;

redraw

Clock Example

Drawing a Clock



An example of the type of image that you might want to generate dynamically is a clock. If the image at right were not generated dynamically, this page would require 43,200 different versions of the image to show any and all times.

The steps to create the image are fairly simple. First, we set the image's background color to white with **imageColorAllocate()** and save that color's resource id in a variable for later use. The first time we call **imageColorAllocate()**, the background color of the image will be set to the defined color (only with **imageCreate()**, not **imageCreateTrueColor()**). Subsequent calls to **imageColorAllocate()** will only define a color within the color index, but not add it anywhere within the image itself. Next we'll need to define the brown and the black colors to be used in the image. With the colors defined, using **imageFilledEllipse()** we'll draw three progressively smaller ellipses: black, brown and then white that will define the face.

We could use **imageFilledRectangle()** in conjunction with some trigonometry to draw the hands, but it is easier to let the math take care of itself with **imageFilledArc()**. For that to work all we need to know is the angle, which is found by converting the time (0-11 hours, 0-59 minutes & 0-59 seconds) to degrees and shifting 90 degrees because for some godawful reason, GD uses 3 o'clock as its 0.

```php
<?PHP
/*
clock1.php
Simple example that generates an analog clock image from the current time
*/

// Step 1. Create a new blank image
$im = imageCreate(101, 101);

// Step 2. Add Content
// begin by defining the colors used in the image
// with imageCreate, the first allocated color becomes the background
$white = imageColorAllocate ($im, 0xFF, 0xFF, 0xFF);
$other = ImageColorAllocate ($im, 0x8B, 0x45, 0x13);
$black = imageColorAllocate ($im, 0x00, 0x00, 0x00);
```

```
// then use those colors to draw a series of overlapping circles
imageFilledEllipse ($im, 50, 50, 100, 100, $black);
imageFilledEllipse ($im, 50, 50,  90,  90, $other);
imageFilledEllipse ($im, 50, 50,  75,  75, $white);

// for the hands, calculate the degrees for arc from current time
$hd = imgDegreesFromTime('hour') ;
$md = imgDegreesFromTime('minute') ;
$sd = imgDegreesFromTime('second') ;

// draw the hands using degrees calculated above with small offsets
imageFilledArc ($im, 50, 50, 52, 52, $hd-6, $hd+6, $other, IMG_ARC_PIE);
imageFilledArc ($im, 50, 50, 65, 65, $md-3, $md+2, $other, IMG_ARC_PIE);
imageFilledArc ($im, 50, 50, 70, 70, $sd-2, $sd+1, $black, IMG_ARC_PIE);

// add a final small dot on top
imageFilledEllipse ($im, 50, 50,   5,   5, $other);

// Steps 3-5. Send headers, image data, & destroy image
// refresh won't work for most browsers & not within HTML
header ('Refresh: 1; URL='.$_SERVER['PHP_SELF']);
header('Content-type: image/png');
imagePNG ($im);
imageDestroy ($im);

/**
 * @param $kind string    'hour', 'minute' or 'second'
 * @return int            calculated degrees w/ 0 at 3 o'clock
 * @desc converts the hour, minute or second to 360¼ value
 *       shifted to 3 o'clock for use in GD image functions
 */
function imgDegreesFromTime($kind) {
    switch ($kind) {
        case 'hour' :
            return ((date('g') * 30) + 270) % 360 ; // 30 = 360 / 12 hours
        case 'minute' :
            return ((date('i') * 6)  + 270) % 360 ; //  6 = 360 / 60 minutes
        case 'second' :
            return ((date('s') * 6)  + 270) % 360 ; //  6 = 360 / 60 seconds
    }
}
?>
```

LAST

Simple Transparency

NEXT

Improving the Clock

The clock would look a lot better if we got rid of the white box around it, leaving just the circle. This is accomplished by setting one of our indexed colors the special color "transparent". Any pixels in the image colored with this index will appear transparent in the final image, allowing the background to show through.

int **imageColorTransparent** (resource image [, int color])
Sets the transparent color in the image *image* to *color*. *image* is the image identifier returned by **imageCreate()** and color is a color identifier returned by **imageColorAllocate()**. The identifier of the new (or current, if none is specified) transparent color is returned.

```
<?PHP
/*
clock2.php
Generates an analog clock png from the current time
on a transparent background
*/

// Step 1. Create a new blank image
$im = imageCreate(101, 101);

// Step 2. Add Content
// begin by defining the colors used in the image
// this time, the first color defined will become transparent
$background = imageColorAllocate ($im, 0, 0, 0);
$background = imageColorTransparent($im,$background);

$nyphpPurp = imageColorAllocate ($im, 0x70, 0x6D, 0x85);
$nyphpBlue = imageColorAllocate ($im, 0x1A, 0x1A, 0x73);
$nyphpGrey = imageColorAllocate ($im, 0xDD, 0xDD, 0xDD);

// then use those colors to draw a series of overlapping circles
imageFilledEllipse ($im, 50, 50, 100, 100, $nyphpBlue);
imageFilledEllipse ($im, 50, 50,  90,  90, $nyphpPurp);
imageFilledEllipse ($im, 50, 50,  75,  75, $nyphpGrey);

// for the hands, calculate the degrees for arc from current time
$hd = imgDegreesFromTime('hour') ;
$md = imgDegreesFromTime('minute') ;
$sd = imgDegreesFromTime('second') ;

// draw the hands using degrees calculated above with small offsets
imageFilledArc ($im, 50, 50, 52, 52, $hd-6, $hd+6, $nyphpPurp, IMG_ARC_PIE);
imageFilledArc ($im, 50, 50, 65, 65, $md-3, $md+2, $nyphpPurp, IMG_ARC_PIE);
imageFilledArc ($im, 50, 50, 70, 70, $sd-2, $sd+1, $nyphpBlue, IMG_ARC_PIE);

// add a final small dot on top
```

```
imageFilledEllipse ($im, 50, 50,   5,   5, $nyphpPurp);

// Steps 3-5. Send headers, image data, & destroy image
header ('Refresh: 1; URL='.$_SERVER['PHP_SELF']);
header('Content-type: image/png');
imagePNG ($im);
imageDestroy ($im);

/**
 * @param $kind string    'hour', 'minute' or 'second'
 * @return int            calculated degrees w/ 0 at 3 o'clock
 * @desc converts the hour, minute or second to 360¼ value
 *       shifted to 3 o'clock for use in GD image functions
 */
function imgDegreesFromTime($kind) {
    switch ($kind) {
        case 'hour' :
            return ((date('g') * 30) + 270) % 360 ; // 30 = 360 / 12 hours
        case 'minute' :
            return ((date('i') * 6)  + 270) % 360 ; //  6 = 360 / 60 minutes
        case 'second' :
            return ((date('s') * 6)  + 270) % 360 ; //  6 = 360 / 60 seconds
    }
}
?>
```

LAST NEXT

Using Existing Images

As you can see, creating images from scratch can be very tedious. It is often best to modify an existing image if you want anything but the simplest graphics. In addition to **imageCreate()** there are many **imageCreateFrom**... functions. The only thing differentiating most of them is the particular graphics file format they read.

int **imageCreateFromPNG** (string filename)
int **imageCreateFromJPEG** (string filename)
int **imageCreateFromWBMP** (string filename)
int **imageCreateFromGIF** (string filename)
int **imageCreateFromGD** (string filename)
int **imageCreateFromGD2** (string filename)
int **imageCreateFromXBM** (string filename)
int **imageCreateFromXPM** (string filename)

Returns an image identifier representing the image obtained from the given filename or URL.

---

int **imageCreateFromString** (string image)

Returns an image identifier representing the image obtained from the given string.

---

int **imageCreateFromGD2Part** (string filename, int srcX, int srcY, int width, int height)

Create a new image from a given part of GD2 file or URL.

```
<?PHP
/*
tree.php
Cycles through the 3 abstract pngs with each refresh
*/

session_start(); // use sessions to remember state

switch ($_SESSION['png']) {
    case 'tree1.png': // if 1, cycle to 2
        $_SESSION['png'] = 'tree2.png' ;
        break;
    case 'tree2.png': // if 2, cycle to 3
        $_SESSION['png'] = 'tree3.png' ;
        break;
    default:          // otherwise cycle to 1
        $_SESSION['png'] = 'tree1.png' ;
        break;
}

// create, send headers, output & destroy
$im = imageCreateFromPNG($_SESSION['png']) ;
header('Content-type: image/png');
```

```
imagePNG($im);
imageDestroy($im);
?>
```

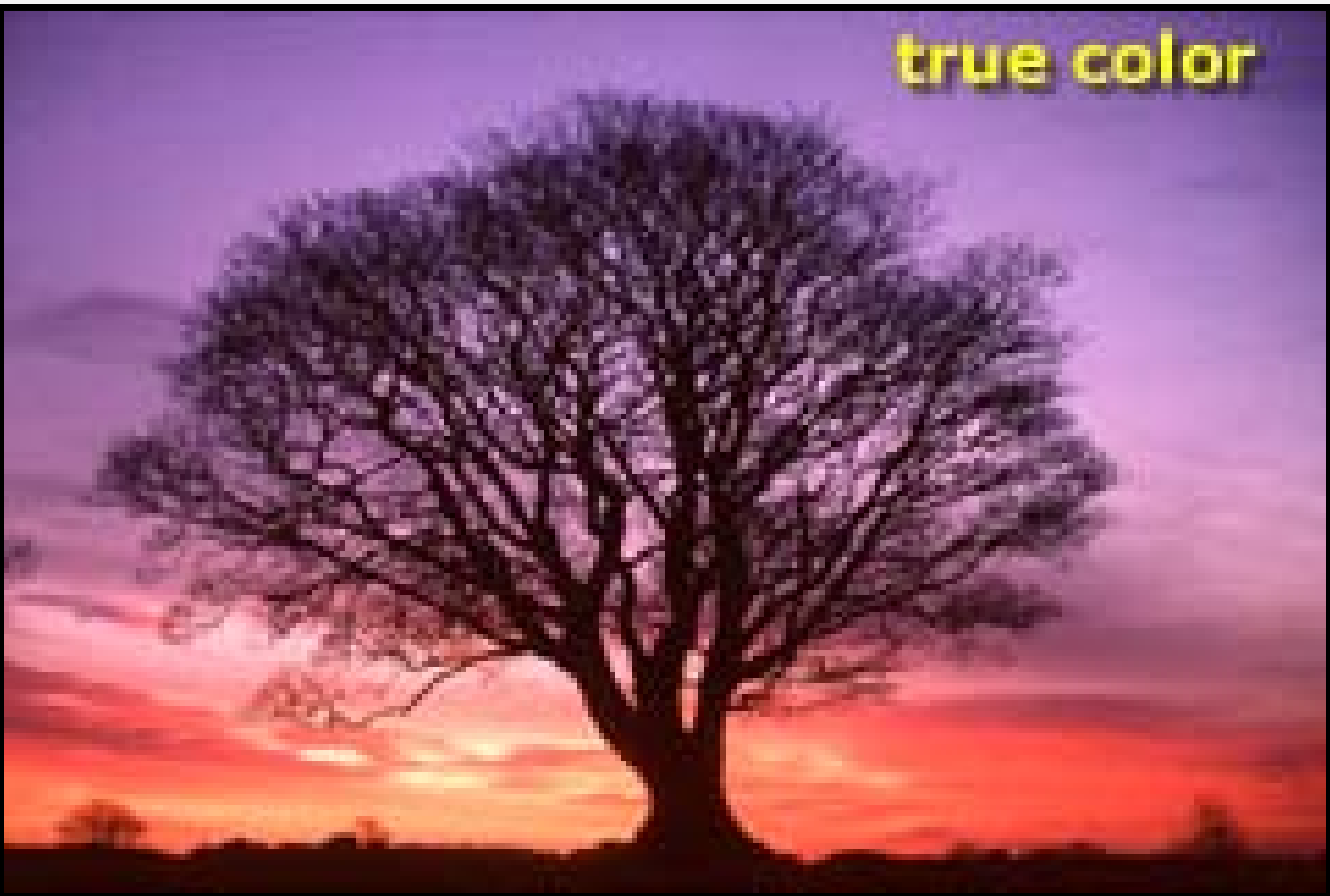


Image Supplied by FreeFoto.com

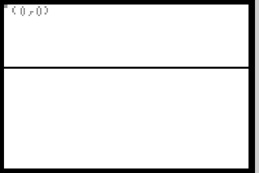
LAST Table of Contents Function Index NEXT

LAST NEXT

Styles, Tiles & Brushes

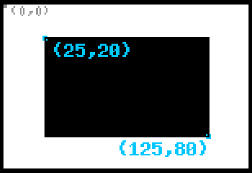
You aren't limited to drawing shapes with a single pixel line, or filling areas with one flat color; the GD library provides several functions that can alter the default behavior of the drawing and filling functions.

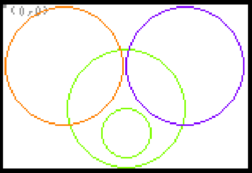
void **imageSetThickness** (resource image, int thickness)
Sets the thickness of the lines drawn when drawing rectangles, polygons, ellipses etc. etc. to *thickness* pixels.



imageThickness ($im, $t) ;

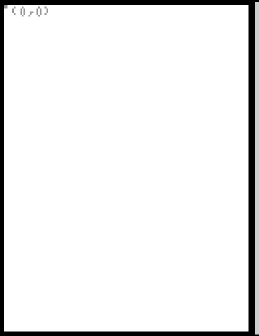
redraw





Unlike the following functions that are only applied in conjunction with the use of special color constants, **imageSetThickness()** is a global property and effects every drawing function after it is called.

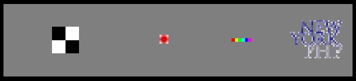
int **imageSetStyle** (resource image, array style)
Sets the style to be used by all line drawing functions when drawing with the special color IMG_COLOR_STYLED. The *style* parameter is an array of pixels.



```
$style = array ( $wh , $wh , $wh , $wh , $wh , $wh , $wh , $wh , $wh , $wh );

imageSetStyle($im, $style) ;

imageLine($im,$x1,$y1,$x2,$y2,IMG_COLOR_STYLED)
```

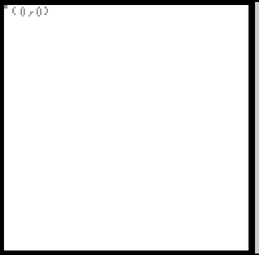
redraw

The next two functions load existing images and use them to create styles. These are the original images provided as options in the examples:



checkerboard red dot rainbow NYPHP

int **imageSetTile** (resource image, resource tile)
Sets the tile image to be used by all region filling functions when filling with the special color IMG_COLOR_TILED. A tile is an image used to fill an area with a repeated pattern. Any GD image can be used as a tile.



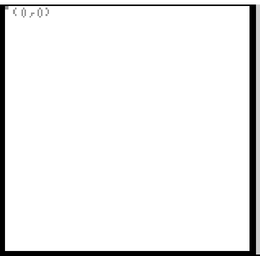
```
imageSetTile ( $im, checkerboard ) ;

imageLine($im,$x1,$y1,$x2,$y2,IMG_COLOR_TILED)
```

redraw

int **imageSetBrush** (resource image, resource brush)
Sets the brush image to be used by all line drawing functions when drawing with the special colors IMG_COLOR_BRUSHED or IMG_COLOR_STYLEDBRUSHED.

```
imageSetBrush ( $im, checkerboard ) ;

imageLine($im,$x1,$y1,$x2,$y2,IMG_COLOR_STYLED)
```

redraw



LAST Table of Contents Function Index NEXT

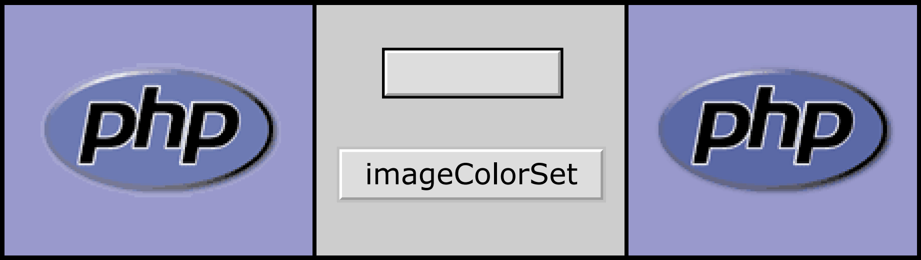
LAST NEXT

The Color Functions

In addition to drawing objects on an existing image, the PHP image functions can also perform transformations on its colors. The main advantage of using index color images with the PHP image functions is that you can globally manipulate colors and color ranges in a predictable manner by altering the index table. This isn't always the case with true color images.

bool **imageColorSet** (resource image, int index, int red, int green, int blue)
Sets the specified index in the palette to the specified color. This is useful for creating flood-fill-like effects in paletted images without the overhead of performing the actual flood-fill.

This function basically reassigns the color value of an index in the table. Any pixel in the image set to that index will now display the new color.



There are several functions provided to determine the index of a color within an image by position or color:

int **imageColorAt** (resource image, int x, int y)
Returns the index of the color of the pixel at the specified location in the image specified by *image*.

int **imageColorClosest** (resource image, int red, int green, int blue)
Returns the index of the color in the palette of the image which is "closest" to the specified RGB value. The "distance" between the desired color and each color in the palette is calculated as if the RGB values represented points in three-dimensional space.

int **imageColorClosestHWB** (resource image, int x, int y, int color)
Returns the index of the color which has the hue, white and blackness nearest to the given color.

int **imageColorExact** (resource image, int red, int green, int blue)
Returns the index of the specified color in the palette of the image. If the color does not exist in the image's palette, -1 is returned.

int **imageColorResolve** (resource image, int red, int green, int blue)
This function is guaranteed to return a color index for a requested color, either the exact color or the closest possible alternative.

Note that the colors in an image you might think are 'exact' and 'closest' aren't always the ones that these functions are going to pick. Using these functions with your images may require some trial and erorr.

```
<?PHP
```

```
error_reporting(E_ALL ^ E_NOTICE);

/*
colorSetPHP.php
uses imageColorSet with an input color to alter the PHP logo
*/

$phpHex = '5B69A6';
$phpColor = hex2int($phpHex);

$newColor = hex2int(validHexColor($_REQUEST['color'],$phpHex)) ;

// should use php_logo_guid(), but that returns a .gif
$im = imageCreateFromPNG('php.png');

$phpCIndex = imageColorExact($im,$phpColor['r'],$phpColor['g'],$phpColor['b']);

imageColorSet($im,$phpCIndex,$newColor['r'],$newColor['g'],$newColor['b']);

header('Content-type: image/png');
imagePNG($im);
imageDestroy($im);

/**
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array( 'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                      'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                      'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
                    );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

?>
```

A few more useful color functions:

int **imageColorsTotal** (resource image)
Returns the number of colors in the specified image's palette.

int **imageColorDeallocate** (resource image, int color)
This function de-allocates a color.

array **imageColorsForIndex** (resource image, int index)
This returns an associative array with red, green, blue and alpha keys that contain the appropriate values for the specified color index.

void **imageTrueColorToPalette** (resource image, bool dither, int ncolors)
Converts a truecolor image to a palette image. If *dither* is TRUE then dithering will be used which will result in a more speckled image but with better color approximation. *ncolors* sets the maximum number of colors that should be retained in the palette. It is usually best to simply produce a truecolor output image instead.

int **imageGammaCorrect** (resource image, float inputgamma, float outputgamma)
Applies gamma correction to a GD image stream (*image*) given an input gamma, *inputgamma* and an output gamma, *outputgamma*.

Gamma correction is the ability to correct for differences in how computer systems interpret color values. Macintosh-generated images tend to look too dark on PCs, and PC-generated images tend to look too light on Macs.

LAST Table of Contents Function Index NEXT

The Interactive Example Source

red: 0

green: 0

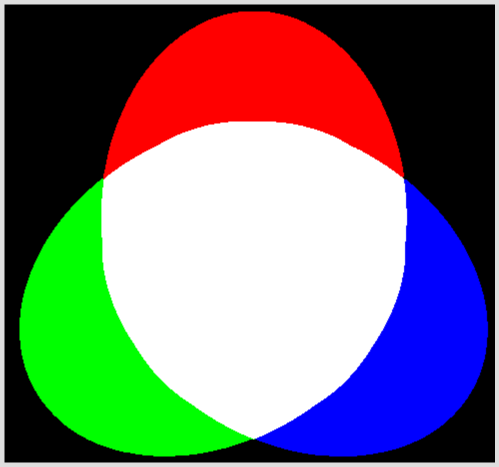
blue: 0

or

hex:

redraw

rgb.php

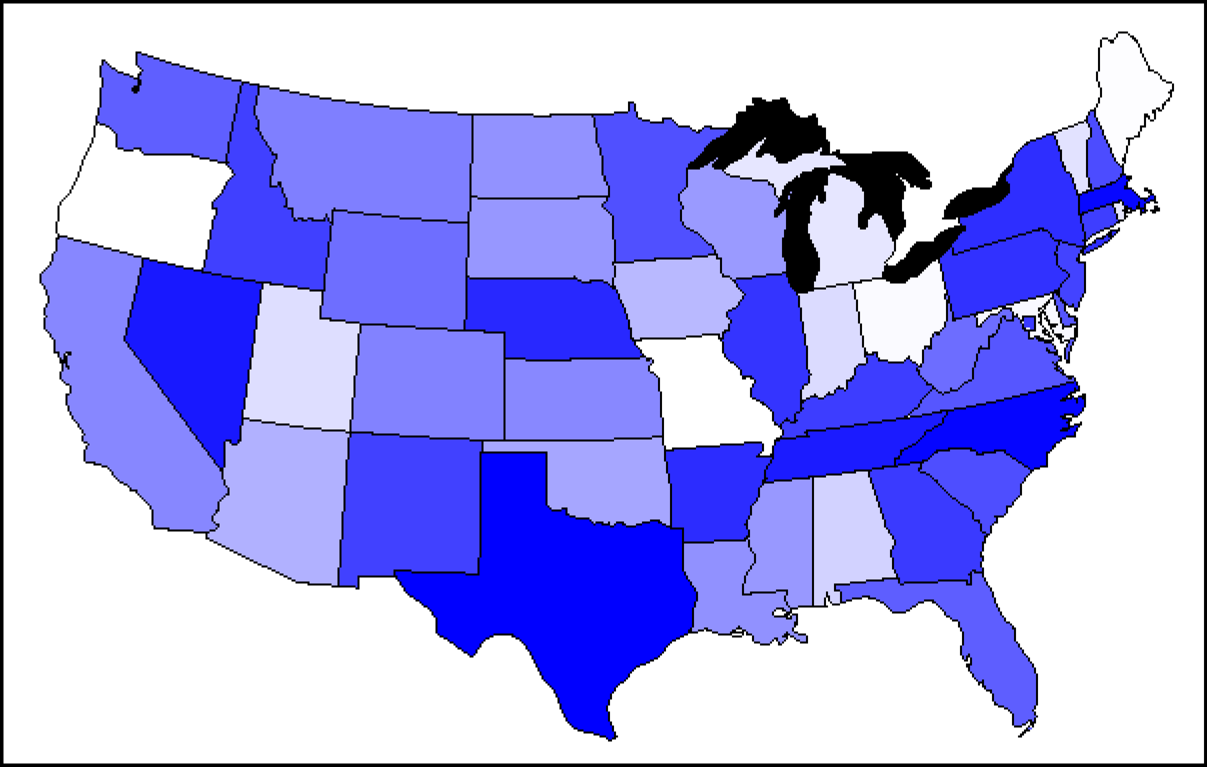


rgb.png

```
<?PHP


error_reporting(E_ALL ^ E_NOTICE);

/*
rgb.php
An interactive example of the additive nature of the RGB color model
*/

if (isset($_REQUEST['hex'])) {
    // if a hex value is passed, use it
    $imColor = hex2int(validHexColor($_REQUEST['hex'])) ;
} else {
    // otherwise use red, green, & blue values
    $imColor['r'] = validIntColor($_REQUEST['red']);
    $imColor['g'] = validIntColor($_REQUEST['green']);
    $imColor['b'] = validIntColor($_REQUEST['blue']);
}

$im = imageCreateFromPNG('rgb.png') ;

// determine the indicies of white and the three primaries
$rIndex = imageColorExact ($im, 0xFF, 0x00, 0x00); // get index of red
$gIndex = imageColorExact ($im, 0x00, 0xFF, 0x00); // get index of green
$bIndex = imageColorExact ($im, 0x00, 0x00, 0xFF); // get index of blue
$wIndex = imageColorExact ($im, 0xFF, 0xFF, 0xFF); // get index of white

// redefine the colors to the input value
// first set pure red, green, and blue portions to their respective components of $imColor
imageColorSet($im, $rIndex, $imColor['r'], 0, 0);
imageColorSet($im, $gIndex, 0, $imColor['g'], 0);
imageColorSet($im, $bIndex, 0, 0, $imColor['b']);
// then set the combined portion to all components of $imColor
imageColorSet($im, $wIndex, $imColor['r'], $imColor['g'], $imColor['b']);

header('Content-type: image/png');
imagePNG($im);
imageDestroy($im);

/**
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array( 'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                      'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                      'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
                    );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

/**
 * @param $input int    integer to be validated
 * @param $default int    default value returned if $input isn't valid
 * @return int            the validated integer
 * @desc returns $input if it is a valid integer color,
 *       otherwise returns $default (which defaults to 0)
 */
function validIntColor($input, $default = 0) {
    // A valid integer color is between 0 and 255
    return (is_numeric($input)) ? (int) max(min($input, 255),0) : (int) $default ;
}

?>
```

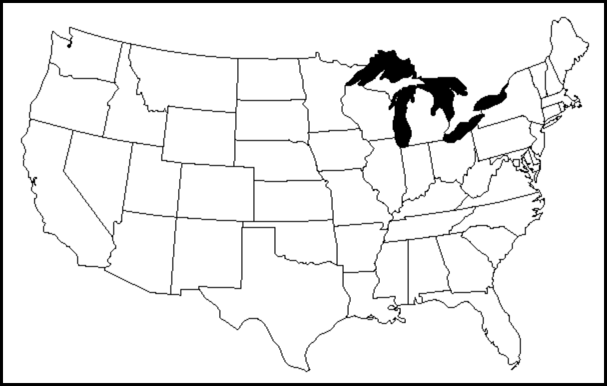
Map Example

In the example below, the original image is an indexed file with each state colored by a different value from the palette. By converting the relative "sales data" of each state to a color value, we can use **imageColorSet()** to display this information graphically. The darker the blue, the more sales in that state.



```
<?PHP
/*
salesbystate.php
Displays a map of the US color-coded to correspond with sales data
*/

// connect to a database & load sales data into an array $states
// with the state code as the key & current sales data as value
require('salesdata.php');

$im = imageCreatefrompng('usa.png') ;
// set predetermined index for each state in image usa.png
$stateIndex = array ('AL' => 16, 'AR' => 17, 'AZ' => 53, 'CA' => 57,
                     'CO' => 52, 'CT' => 37, 'DC' => 40, 'DE' => 38,
                     'FL' => 11, 'GA' => 10, 'IA' => 15, 'ID' =>  1,
                     'IL' => 13, 'IN' => 27, 'KS' =>  3, 'KY' => 26,
                     'LA' => 22, 'MA' => 35, 'MD' => 33, 'ME' => 28,
                     'MI' => 12, 'MN' =>  2, 'MO' =>  8, 'MS' => 21,
                     'MT' => 56, 'NC' => 18, 'ND' =>  7, 'NE' =>  4,
                     'NH' => 32, 'NJ' => 34, 'NM' => 55, 'NV' => 54,
                     'NY' => 19, 'OH' => 23, 'OK' =>  9, 'OR' => 51,
                                   'PA' => 20, 'RI' => 39, 'SC' => 29, 'SD' =>  5,
                                   'TN' => 24, 'TX' => 58, 'UT' => 63, 'VA' => 25,
                                   'VT' => 31, 'WA' =>  6, 'WI' => 14, 'WV' => 30,
                                   'WY' => 50 );

// convert sales data values to a color range
$normalized = gradientFromRange($states);

// In order to fade from blue to white, maximize the blue component
// and then increase the other two components by the same value
foreach ($normalized as $state => $color) {
    imageColorset($im,$stateIndex[$state],$color,$color,255) ;
}

header('Content-type: image/png');
imagePng($im);
imageDestroy($im);

/**
 * @return array
 * @param $usa array states
 * @desc Normalizes an aray of a range of float values to integers
 *        representing gradient color values from 0 to 255
 */
function gradientFromRange($usa) {
    // calculate what we can outside of the for loop
    $lowest = min($usa);
    $ratio = 255 / ( max($usa) - $lowest ) ;
    foreach ($usa as $state => $sales) {
        // subtract the lowest sales from the current sale to zero the figure
        // then multiply by the $ratio to scale highest value to 255.
        // Subtract total from 255 since color is additive,
        // making lowest sales = highest color value (255)
        // & higest sales = lowest color value (0)
        $gradient[$state] = 255 - round(($sales - $lowest) * $ratio)  ;
    }
    return $gradient ;
}

?>
```

Note that in the original image to the right, even though each state is colored with a different index, all the indexes refer to the value of white.

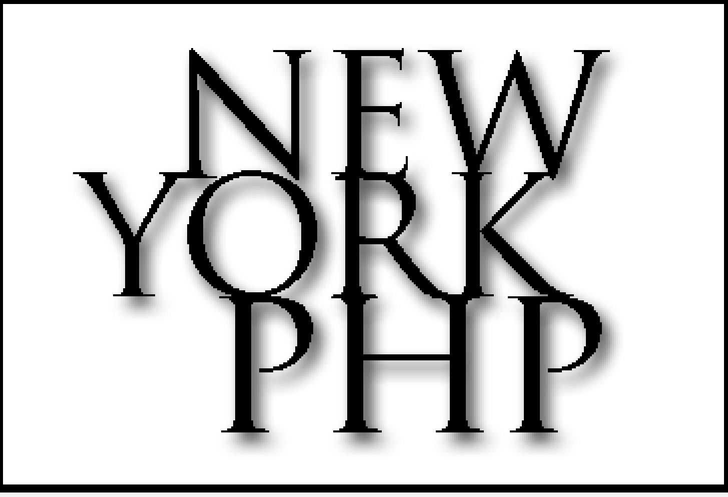


Drop Shadow Example

A practical use of the image functions is to generate a series of permutations from a single set of images. You could, for instance, design all the basic elements of a website in grayscale and use PHP to colorize them to match different color schemes for various subsections or based upon user preferences. The code below demonstrates an example of how to alter the foreground, background and shadow colors of such an image.

Foreground:	000000
Background:	FFFFFF
Shadow:	

redraw



```
<?PHP


error_reporting(E_ALL ^ E_NOTICE);

/*
gradient.php
*/

// convert guaranteed valid hex values to arrays of integers
$foreground = hex2int(validHexColor($_REQUEST['foreground'],'000000'));
$background = hex2int(validHexColor($_REQUEST['background'],'FFFFFF'));
if (isset($_REQUEST['shadow'])) {
    // use a middle grey if color is not valid
    $shadow = hex2int(validHexColor($_REQUEST['shadow'],'666666'));
} else {
    // use a 60% tint of the foreground color if shadow not specified
    $shadow = tintOf($foreground,0.6) ;
}

$im = imageCreateFromPNG('gradient.png');

// Get the indicies of Black & White (the 1st & last values in the table)
$black = imageColorResolve($im,0x00,0x00,0x00) ;
$white = imageColorResolve($im,0xFF,0xFF,0xFF) ;

// from the number of colors, calculate distance between each color step
$colorSteps = imageColorsTotal($im) - 1  ;
```

```
$step['r'] = ($shadow['r'] - $background['r']) / $colorSteps ;
$step['g'] = ($shadow['g'] - $background['g']) / $colorSteps ;
$step['b'] = ($shadow['b'] - $background['b']) / $colorSteps ;

// for each index in the palette starting with white,
// set color to new calculated value
for ($n=$white;$n<$black;++$n)
    imageColorSet($im,$n,
                    round(($n * $step['r']) + $background['r']),
                    round(($n * $step['g']) + $background['g']),
                    round(($n * $step['b']) + $background['b'])
                    ) ;
// then set black to the new foreground color
imageColorSet($im,$black,$foreground['r'],$foreground['g'],$foreground['b']);

header("Content-type: image/png");
imagePNG($im);
imageDestroy($im);


/**
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array(
                    'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                    'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                    'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
                );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

/**
 * @param $color array  3 elements 'r', 'g', & 'b' = int color values
 * @param $tint float   fraction of color $color to return
 * @return array        new color tinted to $tint of $color
 * @desc returns a new color that is a tint of the original
 *       e.g. (255,127,0) w/ a tint of 0.5 will return (127,63,0)
 */
function tintOf($color,$tint) {
    return array ( 'r' => round(((255-$color['r']) * (1-$tint)) + $color['r']),
                   'g' => round(((255-$color['g']) * (1-$tint)) + $color['g']),
                   'b' => round(((255-$color['b']) * (1-$tint)) + $color['b']) );
}

?>
```

Copying, Resizing And Rotating Functions

Transformations on an image like resizing and rotating are achieved indirectly by applying the transformation while copying a source image to a destination.

int **imageCopy** (resource dst_im, resource src_im, int dst_x, int dst_y, int src_x, int src_y, int src_w, int src_h)
Copies a part of *src_im* onto *dst_im* starting at the x, y coordinates *src_x*, *src_y* with a width of *src_w* and a height of *src_h*. The portion defined will be copied onto the x, y coordinates, *dst_x* and *dst_y*.

imageCopy (white .jpg , white .jpg , $dx , $dy , $sx , $sy , $sw , $sh) ;

redraw

+ =>

int **imageCopyMerge** (resource dst_im, resource src_im, int dst_x, int dst_y, int src_x, int src_y, int src_w, int src_h, int pct)
Copies a part of *src_im* onto *dst_im* starting at the x, y coordinates *src_x*, *src_y* with a width of *src_w* and a height of *src_h*. The portion defined will be copied onto the x, y coordinates, *dst_x* and *dst_y*. The two images will be merged according to *pct* which can range from 0 to 100. When *pct* = 0, no action is taken, when 100 this function behaves identically to **imageCopy()**.

imageCopyMerge (white .jpg , white .jpg , $dx , $dy , $sx , $sy , $sw , $sh , $pct) ;

redraw

+ =>

int **imageCopyMergeGray** (resource dst_im, resource src_im, int dst_x, int dst_y, int src_x, int src_y, int src_w, int src_h, int pct)
Copies a part of *src_im* onto *dst_im* starting at the x, y coordinates *src_x*, *src_y* with a width of *src_w* and a height of *src_h*. The portion defined will be copied onto the x, y coordinates, *dst_x* and *dst_y*. The two images will be merged according to *pct* which can range from 0 to 100. When *pct* = 0, no action is taken, when 100 this function behaves identically to **imageCopy()**. This function is identical to **imageCopyMerge()** except that when merging it preserves the hue of the source by converting the destination pixels to gray scale before the copy operation.

imageCopyMergeGray (white .jpg , white .jpg , $dx , $dy , $sx , $sy , $sw , $sh , $pct) ;

redraw

+ =>

int **imageCopyResized** (resource dst_im, resource src_im, int dstX, int dstY, int srcX, int srcY, int dstW, int dstH, int srcW, int srcH)
Copies a rectangular portion of one image to another image. *dst_im* is the destination image, *src_im* is the source image identifier. If the source and destination coordinates and width and heights differ, appropriate stretching or shrinking of the image fragment will be performed. The coordinates refer to the upper left corner. This function can be used to copy regions within the same image (if *dst_im* is the same as *src_im*) but if the regions overlap the results will be unpredictable.

imageCopyResized (white .jpg , white .jpg , $dx , $dy , $sx , $sy , $dw , $dh , $sw , $sh) ;

redraw

+ =>

int **imageCopyResampled** (resource dst_im, resource src_im, int dstX, int dstY, int srcX, int srcY, int dstW, int dstH, int srcW, int srcH)
Copies a rectangular portion of one image to another image, smoothly interpolating pixel values so that, in particular, reducing the size of an image still retains a great deal of clarity. *dst_im* is the destination image, *src_im* is the source image identifier. If the source and destination coordinates and width and heights differ, appropriate stretching or shrinking of the image fragment will be performed. The coordinates refer to the upper left corner. This function can be used to copy regions within the same image (if *dst_im* is the same as *src_im*) but if the regions overlap the results will be unpredictable.

imageCopyResampled (white .jpg , white .jpg , $dx , $dy , $sx , $sy , $dW , $dH , $sH , $sH) ;

redraw

+ =>

bool **imageColorMatch** (resource image1, resource image2)
Makes the colors of the palette version of an image more closely match the true color version. *image1* must be Truecolor, *image2* must be Palette, and both *image1* and *image2* must be the same size.

imageColorMatch (white .jpg , white .png) ;

redraw

+ =>

int **imagePaletteCopy** (resource destination, resource source)
Copies the palette from the *source* image to the *destination* image.

imagePaletteCopy (white .png , white .png) ;

redraw

+ =>

resource **imageRotate** (resource src_im, float angle, int bgd_color)
Rotates the *src_im* image using a given *angle* in degree. *bgd_color* specifies the color of the uncovered zone after the rotation.

imageRotate (white .jpg , $angle , $black) ;

redraw

=>

Thumbnail Example

image:

Upload to Gallery

```
<?php
/*
 * thumbnailer.php
*/

/**
 * @return string
 * @param $o_file string    Filename of image to make thumbnail of
 * @param $t_file string    Filename to use for thumbnail
 * @desc Takes an image and makes a jpeg thumbnail defaulted to 100px high
 */
function makeThumbnail($o_file, $t_file, $t_ht = 100) {
    $image_info = getImageSize($o_file) ; // see EXIF for faster way

    switch ($image_info['mime']) {
        case 'image/gif':
            if (imagetypes() & IMG_GIF)  { // not the same as IMAGETYPE
                $o_im = imageCreateFromGIF($o_file) ;
            } else {
                $ermsg = 'GIF images are not supported<br />';
            }
            break;
        case 'image/jpeg':
            if (imagetypes() & IMG_JPG)  {
                $o_im = imageCreateFromJPEG($o_file) ;
            } else {
                $ermsg = 'JPEG images are not supported<br />';
            }
            break;
        case 'image/png':
            if (imagetypes() & IMG_PNG)  {
                $o_im = imageCreateFromPNG($o_file) ;
            } else {
                $ermsg = 'PNG images are not supported<br />';
            }
            break;
        case 'image/wbmp':
            if (imagetypes() & IMG_WBMP)  {
                $o_im = imageCreateFromWBMP($o_file) ;
            } else {
                $ermsg = 'WBMP images are not supported<br />';
            }
            break;
        default:
            $ermsg = $image_info['mime'].' images are not supported<br />';
            break;
    }

    if (!isset($ermsg)) {
        $o_wd = imagesx($o_im) ;
        $o_ht = imagesy($o_im) ;
        // thumbnail width = target * original width / original height
        $t_wd = round($o_wd * $t_ht / $o_ht) ;

        $t_im = imageCreateTrueColor($t_wd,100);

        imageCopyResampled($t_im, $o_im, 0, 0, 0, 0, $t_wd, $t_ht, $o_wd, $o_ht);

        imageJPEG($t_im,$t_file);

        imageDestroy($o_im);
        imageDestroy($t_im);
    }
    return isset($ermsg)?$ermsg:NULL;
}


?>
```

LAST NEXT

Alpha Channels

There are times when an all-or-nothing approach to transparency isn't sufficient, like when you need the background to show through part or all of your image. In the 70's at New York Tech Catmull & Smith invented an approach to this problem called an alpha channel. Basically, a fourth channel is added to the image containing the values for the transparency of each pixel. Typically, this channel is of the same bit-depth as the others, but GD's alpha channel only supports values from 0 to 127. Alpha Channels are only supported in true color images.

int **imageColorAllocateAlpha** (resource image, int red, int green, int blue, int alpha)
Behaves identically to **imageColorAllocate**with the addition of the transparency parameter alpha which may have a value between 0 and 127. 0 indicates completely opaque while 127 indicates completely transparent.

The color information functions have also been extended to support alpha channels.

int **imageColorClosestAlpha** (resource image, int red, int green, int blue, int alpha)
Returns the index of the color in the palette of the image which is "closest" to the specified RGB value and *alpha* level.

int **imageColorExactAlpha** (resource image, int red, int green, int blue, int alpha)
Returns the index of the specified color+alpha in the palette of the image. If the color does not exist in the image's palette, -1 is returned.

int **imageColorResolveAlpha** (resource image, int red, int green, int blue, int alpha)
This function is guaranteed to return a color index for a requested color, either the exact color or the closest possible alternative.

There are two different ways GD draws within truecolor images. In **blending mode**, the alpha channel component of the color supplied to all drawing functions determines how much of the underlying color should be allowed to shine through. As a result, GD automatically blends the existing color at that point with the drawing color, and stores the result in the image. The resulting pixel is opaque. In **non-blending mode**, the drawing color is copied literally with its alpha channel information, replacing the destination pixel.

int **imageAlphaBlending** (resource image, bool blendmode)
Sets the blending mode for an image, allowing for two different modes of drawing on truecolor images. If blendmode is TRUE, then blending mode is enabled, otherwise disabled. Notice that AlphaBlending is ON by default. So, only use this function if you don't want to use AlphaBlending.

While this function is useful for manipulating layers of images within GD, it is not supported within any of the output file formats except for true color (not indexed) PNGs, and you must specify you want the PNG saved in a manner that supports it (i.e. PNG-24 vs. the standard PNG-8). Note also that not all browsers support PNG-24.

int **imageSaveAlpha** (resource image, bool saveflag)
Sets the flag to attempt to save full alpha channel information (as opposed to single-color transparency) when saving PNG images. You have to unset alpha blending (**imageAlphaBlending($im,FALSE)**), to use it.

LAST Table of Contents Function Index NEXT

LAST NEXT

Utility Functions

int **imagesx** (resource image)
Returns the width of the image identified by *image*.

int **imagesy** (resource image)
Returns the height of the image identified by *image*.

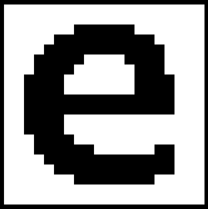
bool **imageIsTrueColor** (resource image)
Returns TRUE if the image *image* is a truecolor image.

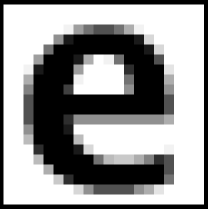
int **imageInterlace** (resource image [, int interlace])
Turns the interlace bit on or off. If interlace is 1 the image will be interlaced, and if interlace is 0 the interlace bit is turned off. If the image is used as a JPEG image, the image is created as a progressive JPEG.
This function returns whether the interlace bit is set for the image.

Interlaced images have their rows stored in some order that is more or less uniformly distributed throughout the image, rather than sequentially. Web browsers make use of this by displaying a crude representation of the image which gets finer over time as the image finishes loading. This is sometimes convenient because it allows the user to see a general impression of the whole image without having to wait for all of it to load.

bool **imageAntialias** (int im, bool on)
Should antialias functions be used or not. This function is currently not documented; only the argument list is available.

This term aliasing is originally from signal processing where it refers to an undersampled function yielding unwanted results. In imaging an example is a diagonal line drawn on a low-resolution raster display, yielding an undesirable "staircase" look. Antialiasing smooths out this discretization of an image by padding pixels with intermediate colors.


aliased


antialiased

LAST Table of Contents Function Index NEXT

LAST NEXT

Fonts: Bitmap

Five bitmap fonts are provided with gd: gdFontTiny, gdFontSmall, gdFontMediumBold, gdFontLarge, and gdFontGiant. These fonts are identified for use in the image functions, in order, by the integers 1-5.

int **imageString** (resource image, int font, int x, int y, string s, int color)
Draws the string *s* in the image identified by *image* at coordinates *x*, *y* (top left is 0, 0) with the color *color*.

int **imageStringUp** (resource image, int font, int x, int y, string s, int color)
Draws the string *s* vertically in the image identified by *image* at coordinates *x*, *y* (top left is 0, 0) with the color *color*.

int **imageChar** (resource image, int font, int x, int y, string c, int color)
Draws the first character of *c* in the image identified by *id* with its upper-left at *x*, *y* (top left is 0, 0) with the color *color*.

int **imageCharUp** (resource image, int font, int x, int y, string c, int color)
Draws the first character of *c* vertically in the image identified by *image* at coordinates *x*, *y* (top left is 0, 0) with the color *color*.

imageString ($im, gdFontMediumBold (3) , $x , $y , , $white);

redraw

In order to calculate the position of text within an image, there are two useful functions to help determine how much space a character will require.

int **imageFontHeight** (int font)
Returns the pixel height of a character in the specified font.

int **imageFontWidth** (int font)
Returns the pixel width of a character in the specified font.

In the `imageString()` example above (simplestring.php), the default $x value is set to center the input string by calculating half of the total of the font width multiplied by the number of characters in the string, then subtraced from the image width:

```
<?PHP


error_reporting(E_ALL ^ E_NOTICE);

/*
simplestring.php
Generates image of a colored box with text
*/
$wd = 400 ;
$ht = 40 ;
```

```
// convert guaranteed valid hex value to array of integers
$bkgColor = hex2int(validHexColor($_REQUEST['bColor']));
$txtColor = hex2int(validHexColor($_REQUEST['tColor'],'ffffff'));
if ($txtColor === $bkgColor) $bkgColor = hex2int('FFFFFF') ;

$font = (is_numeric($_REQUEST['font']))  ? max(min($_REQUEST['font'], 5),1):3;
$string = safeString($_REQUEST['string']) ;

// if a number is passed for x, make sure it is > 0 and < image width - string width
$x = (is_numeric($_REQUEST['x']))
    ? max(min($_REQUEST['x'],$wd - ceil( imageFontWidth($font) * strlen($string))),0)
    : ceil(($wd - (imageFontWidth($font) * strlen($string)))/2);
// if a number is passed for 7, make sure it is > 0 and < image height - string height
$y = (is_numeric($_REQUEST['y']))
    ? max(min($_REQUEST['y'],$ht - imageFontHeight($font)),0)
    : ceil(($ht - imageFontHeight($font)) / 2) ;

$im = imageCreate($wd,$ht);

$bkgC = imageColorAllocate($im, $bkgColor['r'], $bkgColor['g'], $bkgColor['b']);
$txtC = imageColorAllocate($im, $txtColor['r'], $txtColor['g'], $txtColor['b']);

imageString($im,$font,$x,$y,$string,$txtC) ;

// send headers, output & destroy
header('Content-type: image/png');
imagePNG($im);
imageDestroy($im);

/**
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array( 'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                      'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                      'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
                    );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

function safeString($input) {
    return stripslashes($input) ;
}
?>
```

In addition to the fonts distributed with gd, you can also use any bitmap fonts installed on the system. The font file format is binary and architecture dependent. Since this introduction is intended to be run on several environments, there will be no demonstration of this function.

int **imageLoadFont** (string file)
Loads a user-defined bitmap font and returns an identifier for the font (that is always greater than 5, so it will not conflict with the built-in fonts).

LAST Table of Contents Function Index NEXT

Fonts: TrueType

Unlike bitmap fonts that are basically collections of pixel-based images of each character drawn one specific size, vector (or outline) fonts describe each character mathematically as points and lines, and a single font file is used to generate text at any size.

Vector fonts come in a potentially confusing array of types, but the PHP image functions (if you have the appropriate libraries) use the two most common: TrueType and PostScript (Type1). TrueType fonts require the FreeType libraries, and PostScript fonts require the currently orphaned T1Lib library. You can find an excellent guide to getting these libraries compiled and working with PHP & GD (including the version bundled in 4.3) in a *NIX environment here.

In addition to the required libraries, you'll also need the actual font files. PostScript fonts are most often require licensing, you can find some free fonts online and included with X. There are many more free TrueType fonts available, and while they are of generally lower quality, for most graphic purposes are fine.

TrueType

In order to convert a string of characters using a TrueType font into a pixelated image (a process called rendering), the gd library requires either (or both) the FreeType or FreeType 2 open source font rendering libraries. FreeType uses a patented optimization algorithm that requires a royalty fee to be paid or must be disabled (resulting in lower quality images). The more recent FreeType 2 doesn't suffer from this limitation.

The easiest way to tell if one or both of the libraries are installed on your machine is to check for the freetype.h file:

```
# locate freetype.h
/usr/include/freetype1/freetype/freetype.h
/usr/include/freetype2/freetype/freetype.h
```

Rendering with FreeType

array **imageTTFText** (resource image, int size, int angle, int x, int y, int color, string fontfile,

string text)
Draws the string text in the image identified by *image*, starting at coordinates *x*, *y* (top left is 0, 0), at an angle of *angle* in color *color*, using the TrueType font file identified by *fontfile*. Depending on which version of the GD library that PHP is using, when fontfile does not begin with a leading '/', '.ttf' will be appended to the filename and the library will attempt to search for that filename along a library-defined font path.
The coordinates given by *x*, *y* will define the basepoint of the first character (roughly the lower-left corner of the character).
angle is in degrees, with 0 degrees being left-to-right reading text (3 o'clock direction), and higher values representing a counter-clockwise rotation. (i.e., a value of 90 would result in bottom-to-top reading text).
fontfile is the path to the TrueType font you wish to use.
text is the text string which may include UTF-8 character sequences (of the form: &#123;) to access characters in a font beyond the first 255.
color is the color index. Using the negative of a color index has the effect of turning off antialiasing.
Returns an array with 8 elements representing four points making the bounding box of the text. The order of the points is lower left, lower right, upper right, upper left. The points are relative to the text regardless of the angle, so "upper left" means in the top left-hand corner when you see the text horizontallty.

array **imageTTFBBox** (int size, int angle, string fontfile, string text)
Uses a subset of the parameters for **imageTTFText()**, and does nothing but returns the same bounding box array that would be returned by **imageTTFText()**.

Rendering with FreeType 2

array **imageFTText** (resource image, int size, int angle, int x, int y, int col, string font_file, string text, array extrainfo)
Write text to the image using fonts using FreeType 2.

array **imageFTBBox** (int size, int angle, string font_file, string text, array extrainfo)
Give the bounding box of a text using fonts via freetype2

Note: the online manual indicates that the *extrainfo* paramter is optional, but this is not the case. If you don't have any *extrainfo* to pass to this (as yet undocumented) function, simply use array() instead to pass a blank parameter.

LAST NEXT

Fonts: PostScript

PostScript

The PostScript font functions made available by the T1Lib library provide a much higher degree of control than the TrueType functions. Note the ability to control inter-character spacing (*tightness*), multiple levels of antialiasing, as well as slant (italicize) and extend (embold) fonts. Note also that, like `imageDestroy()`, you have to clean up after yourself with `imagePSFreeFont()`.

int **imagePSLoadFont** (string filename)
If everything went right, a valid font index will be returned and can be used for further purposes. Otherwise, the function returns FALSE and prints a message describing what went wrong.

array **imagePSText** (resource image, string text, int font, int size, int foreground, int background, int x, int y [, int space [, int tightness [, float angle [, int antialias_steps]]]])
foreground is the color in which the text will be painted. *background* is the color to which the text will try to fade in with antialiasing. No pixels with the color background are actually painted, so the background image does not need to be of solid color.
The coordinates given by *x*, *y* will define the origin (or reference point) of the first character (roughly the lower-left corner of the character). This is different from the **imageString()**, where *x*, *y* define the upper-right corner of the first character.
space allows you to change the default value of a space in a font. This amount is added to the normal value and can also be negative.
tightness allows you to control the amount of white space between characters. This amount is added to the normal character width and can also be negative.
angle is in degrees.
size is expressed in pixels.
antialias_steps allows you to control the number of colours used for antialiasing text. Allowed values are 4 and 16. The higher value is recommended for text sizes lower than 20, where the effect in text quality is quite visible. With bigger sizes, use 4 as it's less computationally intensive.
Parameters *space* and *tightness* are expressed in character space units, where 1 unit is 1/1000th of an em-square.
This function returns an array containing the following elements: (0 => lower left x-coordinate, 1 => lower left y-coordinate, 2 => upper right x-coordinate, 3 => upper right y-coordinate).

array **imagePSBBox** (string text, int Font, int size [, int space [, int tightness [, float angle]]])
Uses a subset of the parameters for **imagePSText()**, and does nothing but returns the same bounding box array that would be returned by **imagePSText()**.

TIP: The paramter *text* must be a string. In the example below, PHP will quit with an error if $text is undefined:

```
<?
php imagePSText ($im, $text, $font, $textsize, $black, $white, 10, 10); ?>
```

To easily avoid this situation, simply enclose the $text variable in quotes:

```
<?
php imagePSText ($im, "$text", $font, $textsize, $black, $white, 10, 10); ?
>
```

void **imagePSFreeFont** (int fontindex)
Frees memory used by a PostScript Type 1 font.

bool **imagePSExtendFont** (int font_index, float extend)
Extend or condense a font (*font_index*), if the value of the *extend* parameter is less than one you will be condensing the font.

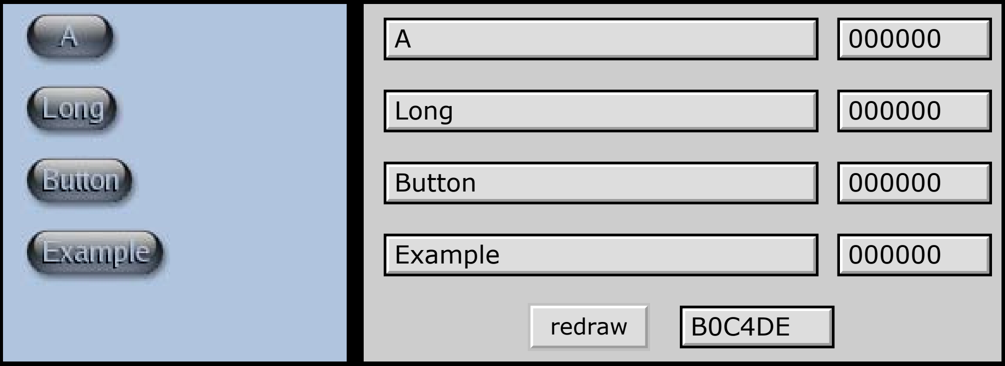
bool **imagePSSlantFont** (int font_index, float slant)
Slant a font given by the *font_index* parameter with a slant of the value of the *slant* parameter.

int **imagePSEncodeFont** (int font_index, string encodingfile)
Loads a character encoding vector from from a file (*.enc) and changes the fonts encoding vector to it. As a PostScript fonts default vector lacks most of the character positions above 127, you'll definitely want to change this if you use an other language than english.

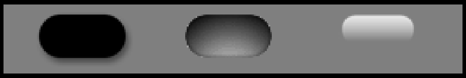
int **imagePSCopyFont** (int fontindex)
Use this function if you need make further modifications to the font, for example extending/condensing, slanting it or changing it's character encoding vector, but need to keep the original as well. Note that the font you want to copy must be one obtained using **imagePSLoadFont()**, not a font that is itself a copied one. You can make modifications to it before copying.
If you use this function, you must free the fonts obtained this way yourself and in reverse order. Otherwise your script will hang.
If everything went right, a valid font index will be returned and can be used for further purposes. Otherwise the function returns FALSE and prints a message describing what went wrong.

LAST Table of Contents Function Index NEXT

Button Example



A	000000
Long	000000
Button	000000
Example	000000
redraw	B0C4DE



btn_shadow.png btn_body.png btn_highlight.png

```
<?PHP

error_reporting(E_ALL ^ E_NOTICE);

/*
btn.php
Outputs a glossy button PNG based on text & color inputs & button state
*/

// set some global parameters where they are easy to change
$font_sz = 12;
$font = 'luxisr.ttf' ; // local ttfs were trouble, so use an X11 font
$text_top = 8 ; // Distance in pixels from top the text should start
$min_cap = 32 ; // The minimum width of text area

// validate the user input
$bg = hex2int(validHexColor($_REQUEST['bg'],'ffffff'));
$fg = hex2int(validHexColor($_REQUEST['fg'],'000000'));
$string = safeString($_REQUEST['string']) ;
// if button is in "down" state, move 2 pixels down & right
$offset = ($_REQUEST['state'] === 'down') ?  2 :  0 ;

// open button body image & calculate height & initial width
$btn_body = imageCreateFromPNG('btn_body.png');
$btn_ht = imagesy($btn_body);
$start_wd = imagesx($btn_body);
$cap_wd = $start_wd/2;

// Recolor button body based upon user input
// from the number of colors, calculate distance between each color step
$colorSteps = imageColorsTotal($btn_body) - 2  ; // don't count transparency
$step['r'] = ($fg['r'] - $bg['r']) / $colorSteps ;
$step['g'] = ($fg['g'] - $bg['g']) / $colorSteps ;
$step['b'] = ($fg['b'] - $bg['b']) / $colorSteps ;

// for each index in the palette starting with white,
// set color to new calculated value
for ($n=$colorSteps;$n>=0;--$n)
    imageColorSet($btn_body,$n,
                    round(($n * $step['r']) + $bg['r']),
                    round(($n * $step['g']) + $bg['g']),
                    round(($n * $step['b']) + $bg['b'])
                    ) ;
```

```
// calculate the width of the text block
$tbb = imageFTBBox ($font_sz, 0, $font, $string, array());
$text_wd = $tbb[4] - $tbb[0]; // image width

// use text width to calculate final button width & x,y
$btn_wd = $start_wd + max(0,$text_wd - $min_cap) ;
$x = (($btn_wd - $text_wd) / 2) ;
$y = $font_sz + $text_top ;

// Create final button image & allocate foreground & background
$btn_out = imageCreateTrueColor($btn_wd,34);
$bgColor = imageColorAllocate($btn_out,$bg['r'],$bg['g'],$bg['b']);
$fgColor = imageColorAllocate($btn_out,$fg['r'],$fg['g'],$fg['b']);
imageFill($btn_out,0,0,$bgColor); // this isn't automatic w/ imageCreateTrueColor

// if the button is not in a down state, first draw the shadow beneath all else
if ($offset == 0) {
    $btn_shadow = imageCreateFromPNG('btn_shadow.png');
    // draw middle of image stretching to accomodate longer text
    imageCopyResampled ($btn_out, $btn_shadow, $cap_wd, 0, $cap_wd, 0, $btn_wd-
($cap_wd*2), $btn_ht, 5, $btn_ht) ;
    // copy left & right sides "caps"
    imageCopy($btn_out, $btn_shadow, 0, 0, 0, 0, $cap_wd, $btn_ht) ;
    imageCopy($btn_out, $btn_shadow, $btn_wd - $cap_wd, 0, $cap_wd, 0, $cap_wd, $btn_ht) ;
    // done with shadow, so destroy it
    imageDestroy($btn_shadow);
}

// Then add the main body of the recolored button
// draw middle of image stretching to accomodate longer text
imageCopyResampled ($btn_out, $btn_body, ($cap_wd + $offset), $offset, $cap_wd, 0, $btn_wd-
($cap_wd*2), $btn_ht, 5, $btn_ht) ;
// copy left & right sides "caps"
imageCopy($btn_out, $btn_body, $offset, $offset, 0, 0, $cap_wd, $btn_ht) ;
imageCopy($btn_out, $btn_body, ($btn_wd - $cap_wd + $offset), $offset, $cap_wd, 0, $cap_wd, $btn_ht) ;
// done with shadow, so destroy it
imageDestroy($btn_body);

// Add the text offset by a pixel colored in the foreground to act as highlight
$tbb = imageFTText ($btn_out, $font_sz, 0, ($x+$offset-1), ($y+$offset+1), $fgColor, $font, "$string", array());

// Next add the highlight layer, whitening anything beneath it
$btn_highlight = imageCreateFromPNG('btn_highlight.png');
// draw middle of image stretching to accomodate longer text
imageCopyResampled ($btn_out, $btn_highlight, ($cap_wd + $offset), $offset, $cap_wd, 0, $btn_wd-
($cap_wd*2), $btn_ht, 5, $btn_ht) ;
// copy left & right sides "caps"
imageCopy($btn_out, $btn_highlight, $offset, $offset, 0, 0, $cap_wd, $btn_ht) ;
imageCopy($btn_out, $btn_highlight, ($btn_wd - $cap_wd + $offset), $offset, $cap_wd, 0, $cap_wd, $btn_ht) ;
// done with highlight, so destroy it
imageDestroy($btn_highlight);

// Finally add the text again, not offset, in the background color
$tbb = imageFTText ($btn_out, $font_sz, 0, ($x+$offset), ($y+$offset), $bgColor, $font, "$string", array());

// send headers, output & destroy
header("Content-type: image/png");
imagePNG($btn_out);
imageDestroy($btn_out);

/**
 * @param    $hex string        6-digit hexadecimal color
 * @return    array            3 elements 'r', 'g', & 'b' = int color values
 * @desc Converts a 6 digit hexadecimal number into an array of
 *       3 integer values ('r'  => red value, 'g'  => green, 'b'  => blue)
 */
function hex2int($hex) {
        return array(
                    'r' => hexdec(substr($hex, 0, 2)), // 1st pair of digits
                    'g' => hexdec(substr($hex, 2, 2)), // 2nd pair
                    'b' => hexdec(substr($hex, 4, 2))  // 3rd pair
```

```
                 );
}

/**
 * @param $input string     6-digit hexadecimal string to be validated
 * @param $default string   default color to be returned if $input isn't valid
 * @return string           the validated 6-digit hexadecimal color
 * @desc returns $input if it is a valid hexadecimal color,
 *       otherwise returns $default (which defaults to black)
 */
function validHexColor($input = '000000', $default = '000000') {
    // A valid Hexadecimal color is exactly 6 characters long
    // and eigher a digit or letter from a to f
    return (eregi('^[0-9a-f]{6}$', $input)) ? $input : $default ;
}

function safeString($input) {
    return (isset($input))?stripslashes($input):'Button';
}


?>
```

LAST Table of Contents Function Index NEXT

LAST NEXT

EXIF

Many image formats, especially JPEG, provide the ability to store textual (& binary thumbnail) metadata within the file itself. Digital cameras, scanners and other imaging equipment will often generate technical data and store it in a image more or less standard format know as the Exchangeable Image File Format or EXIF. Details can be found at http://www.exif.org/

if PHP has been compiled with `--enable-exif`, several functions become available to read EXIF data. Since EXIF data is related to the generation of image (see the example below), there is no need to write EXIF data, and none are provided.

int **exif_imageType** (string filename)
Reads the first bytes of an image and checks its signature. When a correct signature is found a constant will be returned otherwise the return value is FALSE. The return value is the same value that **getImageSize()** returns in index 2 but this function is much faster.

int **exif_read_data** (string filename [, string sections [, bool arrays [, bool thumbnail]]]
Reads the EXIF headers from a JPEG or TIFF image file. It returns an associative array where the indexes are the header names and the values are the values associated with those headers. If no data can be returned the result is FALSE.
filename is the name of the file to read. This cannot be an url.
sections is a comma separated list of sections that need to be present in file to produce a result array

FILE	FileName, FileSize, FileDateTime, SectionsFound
COMPUTED	html, Width, Height, IsColor and some more if available.
ANY_TAG	Any information that has a Tag e.g. IFD0, EXIF, ...
IFD0	All tagged data of IFD0. In normal image files this contains image size and so forth.
THUMBNAIIL	A file is supposed to contain a thumbnail if it has a second IFD. All tagged information about the embedded thumbnail is stored in this section.
COMMENT	Comment headers of JPEG images.
EXIF	The EXIF section is a sub section of IFD0. It contains more detailed information about an image. Most of these entries are digital camera related.

arrays specifies whether or not each section becomes an array. The sections FILE, COMPUTED and THUMBNAIL allways become arrays as they may contain values whose names are conflict with other sections.
thumbnail whether or not to read the thumbnail itself and not only its tagged data.

int **read_exif_data** -- Alias of **exif_read_data()**

Description

```
<?PHP
print_r(exif_read_data('AtTheLodge.jpg',ANY_TAG)) ;
?>

Array
(
    [FileName] => AtTheLodge.jpg
    [FileDateTime] => 1227223275
    [FileSize] => 69909
    [FileType] => 2
    [MimeType] => image/jpeg
    [SectionsFound] => ANY_TAG, IFD0, THUMBNAIL, EXIF
    [COMPUTED] => Array
        (
            [html] => width="480" height="360"
            [Height] => 360
            [Width] => 480
            [IsColor] => 1
            [ByteOrderMotorola] => 1
            [ApertureFNumber] => f/2.6
            [Copyright] => (Copyright Notice)
            [Thumbnail.FileType] => 2
            [Thumbnail.MimeType] => image/jpeg
        )

    [ImageDescription] => Only one beer? (Capation)
    [Make] => PENTAX Corporation
    [Model] => PENTAX Optio S
    [Orientation] => 1
    [XResolution] => 72/1
    [YResolution] => 72/1
    [ResolutionUnit] => 2
    [Software] => Adobe Photoshop 7.0
    [DateTime] => 2003:10:29 17:16:34
    [Artist] => Jeff Knight (Author)
    [YCbCrPositioning] => 1
    [Copyright] => (Copyright Notice)
    [UndefinedTag:0xC4A5] => PrintIM0250
    [Exif_IFD_Pointer] => 380
    [THUMBNAIL] => Array
        (
            [Compression] => 6
            [XResolution] => 72/1
            [YResolution] => 72/1
            [ResolutionUnit] => 2
            [JPEGInterchangeFormat] => 938
            [JPEGInterchangeFormatLength] => 5368
        )
    [ExposureTime] => 1/40
    [FNumber] => 26/10
    [ExposureProgram] => 2
    [ExifVersion] => 0220
    [DateTimeOriginal] => 2003:08:22 22:21:05
    [DateTimeDigitized] => 2003:08:22 22:21:05
    [ComponentsConfiguration] =>
    [CompressedBitsPerPixel] => 2048000/3145728
    [ExposureBiasValue] => 0/3
    [MaxApertureValue] => 28/10
    [MeteringMode] => 5
    [LightSource] => 0
    [Flash] => 73
    [FocalLength] => 580/100
    [FlashPixVersion] => 0100
    [ColorSpace] => 65535
    [ExifImageWidth] => 480
    [ExifImageLength] => 360
    [FileSource] =>
    [CustomRendered] => 0
    [ExposureMode] => 0
    [WhiteBalance] => 0
    [DigitalZoomRatio] => 0/0
    [FocalLengthIn35mmFilm] => 35
    [SceneCaptureType] => 0
    [GainControl] => 0
    [Contrast] => 0
    [Saturation] => 0
    [Sharpness] => 0
    [SubjectDistanceRange] => 0
)
```

int **exif_thumbnail** (string filename [, int &width [, int &height [, int &imagetype]]])
Reads the embedded thumbnail of a TIFF or JPEG image. If the image contains no thumbnail FALSE will be returned.
The parameters *width*, *height* and *imagetype* are available since PHP 4.3.0 and return the size of the thumbnail as well as its type. It is possible that **exif_thumbnail()** cannot create an image but can determine its size. In this case, the return value is FALSE but *width* and *height* are set.
If you want to deliver thumbnails through this function, you should send the mimetype information using the **header()** function.

Starting from version PHP 4.3.0, the function exif_thumbnail() can return thumbnails in TIFF format.

IPTC

Non-technical information such as Author and Keywords can also be attached to image headers. The International Press Telecommunications Council has published as standard in widespread use referred to as IPTC.

array **iptcParse** (string iptcblock)
his function parses a binary IPTC block into its single tags. It returns an array using the tagmarker as an index and the value as the value. It returns FALSE on error or if no IPTC data was found. See getimagesize() for a sample.

```
<?PHP
$size = getImageSize ('AtTheLodge.jpg',&$info);
```

```
$iptc = iptcParse($info['APP13']); // key for IPTC
print_r();
?>

Array
(
    [2#000] => Array
        (
            [0] =>
        )

    [2#120] => Array
        (
            [0] => Only one beer? (Capation)
        )

    [2#122] => Array
        (
            [0] => PUTAMARE (Caption Writer)
        )

    [2#105] => Array
        (
            [0] => At The Lodge (Headline)
        )

    [2#040] => Array
        (
            [0] => Not to be taken seriously (Instructions)
        )

    [2#080] => Array
```

```
        (
            [0] => Jeff Knight (Author)
        )

    [2#085] => Array
        (
            [0] => At the End of the Table (Author'
        )

    [2#110] => Array
        (
            [0] => Jeff Knight (Credit)
        )

    [2#115] => Array
        (
            [0] => Beer (Source)
        )

    [2#005] => Array
        (
            [0] => At The Lodge (Title)
        )

    [2#055] => Array
        (
            [0] => 20030822
        )

    [2#090] => Array
        (
            [0] => New York (City)
        )

    [2#095] => Array
        (
            [0] => NY (State/Province)
        )

    [2#101] => Array
        (
            [0] => USA (USA)
        )

    [2#103] => Array
        (
            [0] => (Transmission Reference)
        )

    [2#015] => Array
        (
            [0] => 2
        )

    [2#020] => Array
        (
            [0] => Category 1 (Supplemental Categor
            [1] => Category 2 (Supplemental Categor
            [2] => Category 3 (Supplemental Categor
        )
    [2#010] => Array
        (
            [0] => 1
        )

    [2#025] => Array
        (
            [0] => Elk (Keyword)
            [1] => Head (Keyword)
            [2] => Lodge (Keyword)
            [3] => Andrew (Keyword)
            [4] => Krook (Keyword)
            [5] => Dan (Keyword)
        )

    [2#116] => Array
        (
            [0] => (Copyright Notice)
        )

)
```

array **iptcEmbed** (string iptcdata, string jpeg_file_name [, int spool])
Embeds binary IPTC data into a JPEG image

LAST Table of Contents Function Index NEXT

LAST

Bibliography / Reference

PHP.net Manual

GD Graphics Library
Compiling and Enabling GD in PHP 4.3 by Marco Tabini
An intro to using the GD image library with PHP by Matt Wade

Encyclopedia of Graphics File Formats
PNG (Portable Network Graphics) Home Site
The GIF Controversy: A Software Developer's Perspective
Web Design Group - Image Use on the Web
RGB World: RGB Color Info
Alpha and the History of Digital Compositing by Alvy Ray Smity

Adobe Solutions Network: Font Formats, File Types and Q&A

LAST Table of Contents Function Index