This article shows how to build an object-oriented graphics layer in PHP. Using object-oriented systems can make building complex graphics much easier than building the graphics using the primitives in the standard PHP library.

I lump graphics editing programs into two distinct categories: painting programs that let you tweak an image pixel by pixel and drawing programs that provide a set of objects like lines, ovals, and rectangles that you can manipulate until you render the drawing to a flat image like a JPEG. Paint programs are great for pixel-perfect control. But for business graphics, a drawing program is the way to go because most graphs are sets of rectangles, lines, and ovals.

PHP's built-in drawing primitives are like a paint program. They're great for rendering to an image, but they aren't so good if you want to think of your image as a set of objects. This article shows you how to build an object-oriented graphics library to sit on top of the PHP graphics library. You'll use the object-oriented extensions provided in PHP V5.

With object-oriented graphics support, your graphics code is much easier to understand and maintain. And you have the potential for rendering a graph into multiple types of media -- Flash movies, SVG, etc. -- from a single bit of graphing source.

The goal

Creating a graphics object library involves three primary goals:

Move from primitives to objects

Instead of using imageline, imagefilledrectangle, and other graphics functions, this library should provide objects like Line, Rectangle, and Oval that can be rendered to an image. It should also support the ability to make larger complex objects or to group objects together.

Allow for z-ordering

Drawing programs let the artist move a graphics object above or below other objects on the drawing surface. The library should support this ability to position one object before or after another using a z value that defines the object's height from the surface of the drawing plane.
Objects with higher z values are drawn later and, thus, appear on top of objects with a lower z value.

**Provide for viewport transformations**

Often, the data's coordinate space is not the same as the image. The graphics primitives in PHP work on the coordinate plane of the image. The graphics library should support the specification of a viewport so you can specify the graphics in a coordinate system that is friendlier to the programmer and automatically scales to fit an image of any size.

Because this is a large set of features, you'll write the code step by step to show how the code evolves to add functionality.

**The basics**

Let's start with a graphics environment object and an interface called `GraphicsObject` implemented by a `Line` class that draws lines. The UML is shown in Figure 1.

**Figure 1. The graphics environment and the graphics object interface**

![Figure 1](image)

The `GraphicsEnvironment` class holds the graphics object and a set of colors. It also contains the width and height. The `saveAsPng` method draws the current image out to the specified file.

The `GraphicsObject` is the interface that any graphics object must implement. To start with, all you need is the `render` method to draw the object. It's implemented by a `Line` class that takes four coordinates: the starting and ending x values, and the starting and ending y values. It also has a color. When `render` is called, the object draws a line from sx,sy to ex,ey of the color specified by name.

The code for the library is shown in Listing 1.

**Listing 1. The basic graphics library**

```php
<?php
class GraphicsEnvironment
{
    public $width;
    public $height;
    public $gdo;
    public $colors = array();

    public function __construct( $width, $height )
    {
        $this->width = $width;
        $this->height = $height;
        $this->gdo = imagecreatetruecolor( $width, $height );
        imagefilledrectangle( $this->gdo, 0, 0,
```
$width, $height,
    $this->getColor( "white" ) );
}

public function width() { return $this->width; }

public function height() { return $this->height; }

public function addColor( $name, $r, $g, $b )
{
    $this->colors[ $name ] = imagecolorallocate(
        $this->gdo,
        $r, $g, $b );
}

public function getGraphicObject()
{
    return $this->gdo;
}

public function getColor( $name )
{
    return $this->colors[ $name ];
}

public function saveAsPng( $filename )
{
    imagepng( $this->gdo, $filename );
}
}

abstract class GraphicsObject
{
    abstract public function render( $ge );
}

class Line extends GraphicsObject
{
    private $color;
    private $sx;
    private $sy;
    private $ex;
    private $ey;

    public function __construct( $color, $sx, $sy, $ex, $ey )
    {
        $this->color = $color;
        $this->sx = $sx;
        $this->sy = $sy;
        $this->ex = $ex;
        $this->ey = $ey;
    }

    public function render( $ge )
    {
        imageline( $ge->getGraphicObject(),
            $this->sx, $this->sy,
            $this->ex, $this->ey,
            $ge->getColor( $this->color ) );
    }
}?

The test code is shown in Listing 2.
Listing 2. The test code for the basic graphics library

Listing 2. The test code for the basic graphics library
<?php
require_once( "glib.php" );

$ge = new GraphicsEnvironment( 400, 400 );

$ge->addColor( "black", 0, 0, 0 );
$ge->addColor( "red", 255, 0, 0 );
$ge->addColor( "green", 0, 255, 0 );
$ge->addColor( "blue", 0, 0, 255 );

$gobjs = array();
$gobjs []= new Line( "black", 10, 5, 100, 200 );
$gobjs []= new Line( "blue", 200, 150, 390, 380 );
$gobjs []= new Line( "red", 60, 40, 10, 300 );
$gobjs []= new Line( "green", 5, 390, 390, 10 );

foreach( $gobjs as $gobj ) { $gobj->render( $ge ); }
$ge->saveAsPng( "test.png" );
?>

This test creates a graphics environment. It then creates a few lines pointing in different directions and having different colors. Next, the render method draws them onto the graphics plane. At the end, the code saves the image as test.png.

The command-line interpreter runs the code throughout the article, as follows:

% php test.php
%

Figure 2 shows how the resulting test.png file looks in Firefox.

Figure 2. A simple graphics object test
It's not the Mona Lisa, but it will do for now.

**Adding dimensions**

With the first requirement -- the ability to have graphic objects -- out of the way, it's time to move on to the second requirement: the ability to position objects above and below each other using a z dimension.

Think of each z level being like a plane the size of the original image. The drawing elements are drawn in order from lowest to highest. For example, let's draw two graphics elements: a red circle and a black box. Begin with the circle at 100 and the box at 200. That puts the circle behind the box, as shown in Figure 3.

**Figure 3. The different z-order planes**

Flip the values around, and show the red circle popping above the black rectangle with only a change to the z values. To do this, you need each `GraphicsObject` to have a `z()` method that returns a numeric z value. Because you'll create different graphics objects (`Line`, `Oval`, and `Rectangle`), you'll also create a base class called `BoxObject` used by all three to maintain the starting and ending coordinates, the z value, and the color of the object (see Figure 4).

**Figure 4. Adding the z dimension to the system**

The new code for the graphics library is shown in Listing 3.
Listing 3. The graphics library that can handle z information

```php
<?php

class GraphicsEnvironment
{
    public $width;
    public $height;
    public $gdo;
    public $colors = array();

    public function __construct( $width, $height )
    {
        $this->width = $width;
        $this->height = $height;
        $this->gdo = imagecreatetruecolor( $width, $height );
        $this->addColor( "white", 255, 255, 255 );
        imagefilledrectangle( $this->gdo, 0, 0,
            $width, $height,
            $this->getColor( "white" ) );
    }

    public function width() { return $this->width; }
    public function height() { return $this->height; }

    public function addColor( $name, $r, $g, $b )
    {
        $this->colors[ $name ]= imagecolorallocate( $this->gdo, $r, $g, $b );
    }

    public function getGraphicObject()
    {
        return $this->gdo;
    }

    public function getColor( $name )
    {
        return $this->colors[ $name ];
    }

    public function saveAsPng( $filename )
    {
        imagepng( $this->gdo, $filename );
    }
}

abstract class GraphicsObject
{
    abstract public function render( $ge );
    abstract public function z();
}

abstract class BoxObject extends GraphicsObject
{
    protected $color;
    protected $sx;
    protected $sy;
    protected $ex;
    protected $ey;
    protected $z;

    public function __construct( $z, $color, $sx, $sy, $ex, $ey )
    {
        $this->z = $z;
        $this->color = $color;
    }
```
The test code also needs to be upgraded, as shown in Listing 4.

**Listing 4. The upgraded test code**

```php
<?php
require_once( "glib.php" );

function zsort( $a, $b ) {
    if ( $a->z() < $b->z() ) return -1;
    if ( $a->z() > $b->z() ) return 1;
    return 0;
}

$ge = new GraphicsEnvironment( 400, 400 );
$ge->addColor( "black", 0, 0, 0 );
$ge->addColor( "red", 255, 0, 0 );
$ge->addColor( "green", 0, 255, 0 );
$ge->addColor( "blue", 0, 0, 255 );
```
$gobjs = array();
$gobjs []= new Oval( 100, "red", 50, 50, 150, 150 );
$gobjs []= new Rectangle( 200, "black", 100, 100, 300, 300 );

usort( $gobjs, "zsort" );

foreach( $gobjs as $gobj ) { $gobj->render( $ge ); }

$ge->saveAsPng( "test.png" );
?>

There are two things to notice here. First is the addition of the creation of the **Oval** and **Rectangle** objects, where the first parameter is the z value. Second is the call to usort, which uses the **zsort** function to sort the graphics objects by the z value.

When you run the program, the test.png file should look like Figure 5.

**Figure 5. The red circle behind the black square**

![Figure 5](image.png)

Now, make the following code change:

$gobjs []= new Oval( 200, "red", 50, 50, 150, 150 );
$gobjs []= new Rectangle( 100, "black", 100, 100, 300, 300 );

Run the code again, and suddenly the oval pops up above the rectangle, as shown in Figure 6.
Figure 6. The red circle is now above the rectangle

The red circle appears above the rectangle, even though it's created first and added to the array first. That is the real value of the z dimension: You can create objects in any order you choose and position them relative to each other by adjusting the z value on each object.

In this code, the z-order sorting is done outside the library. Let's fix that by creating a new container object called a Group that can hold a bunch of GraphicsObjects. The Group object will then handle the sorting.

The code for the Group class is shown in Listing 5.

Listing 5. The Group class

```php
function zsort( $a, $b )
{
    if ( $a->z() < $b->z() ) return -1;
    if ( $a->z() > $b->z() ) return 1;
    return 0;
}

class Group extends GraphicsObject
{
    private $z;
    protected $members = array();
    public function __construct( $z )
    {
        $this->z = $z;
    }
    public function add( $member )
    {
        $this->members []= $member;
    }
    public function render( $ge )
    {
        usort( $this->members, "zsort" );
        foreach( $this->members as $gobj )
        {
            $gobj->render( $ge );
        }
    }
}
```
The `Group` object’s job is to hold an array of objects and then, when it's rendered, do the sort and render the objects one by one.

The updated test code is shown in Listing 6.

**Listing 6. The upgraded test code**

```php
<?php
require_once( "glib.php" );

$ge = new GraphicsEnvironment( 400, 400 );

$ge->addColor( "black", 0, 0, 0 );
$ge->addColor( "red", 255, 0, 0 );
$ge->addColor( "green", 0, 255, 0 );
$ge->addColor( "blue", 0, 0, 255 );

$g1 = new Group( 0 );
$g1->add( new Oval( 200, "red", 50, 50, 150, 150 ) );
$g1->add( new Rectangle( 100, "black", 100, 100, 300, 300 ) );

$g1->render( $ge );
$ge->saveAsPng( "test.png" );
?>
```

Now all the client has to do is create one `Group` object. It will handle ordering and rendering everything else.

**Creating a viewport**

A *viewport* is an artificial coordinate system that is translated into the physical coordinate system of the image. The extents of the viewport can be anything you want them to be. For example, the start and end of both the x and y axes could be -2 and 2, making the center of the viewport's coordinate plane 0, 0. That would be a nice viewport for trigonometric graphics like sins or cosines. Or, the viewport could be asymmetric, with y values ranging from -1 to 1 and x values range from 0 to 10,000, depending on your needs.

The other value of a viewport is that the logic that builds a 400-by-400 image is the same logic that builds a 4,000-by-2,000 image. The code writes to viewport, and the viewport does the mapping to the physical size of the image automatically.

To start your viewport work, you'll fix the viewport to range from 0,0 to 1,1 by having the graphics objects call back to the graphics environment to translate their viewport coordinates to physical coordinates. You'll make life a little easier by putting all the code into the `BoxObject` base class.

Figure 7 shows two things about the new code. First is the addition of the `tx` and `ty` methods, which translate x and y coordinates from the viewport to the physical image. Second is the addition of the `draw` method on the `BoxObject`, which derived classes should use to do the drawing. The
BoxObject handles the viewport translation in the `render` method and calls the `draw` method with the physical coordinates. This way, the `Line`, `Oval`, and `Rectangle` classes can all work with viewport coordinates without having to worry about the translation.

**Figure 7. The additions for the graphic environment viewport translations**

The code for the new library is shown in Listing 7.

**Listing 7. The graphics library with the start of viewport support**

```php
<?php
class GraphicsEnvironment
{
    public $width;
    public $height;
    public $gdo;
    public $colors = array();

    public function __construct( $width, $height )
    {
        $this->width = $width;
        $this->height = $height;
        $this->gdo = imagecreatetruecolor( $width, $height );
        $this->addColor( "white", 255, 255, 255 );
        imagefilledrectangle( $this->gdo, 0, 0, $width, $height, $this->getColor( "white" ) );
    }

    public function width() { return $this->width; }
    public function height() { return $this->height; }

    public function addColor( $name, $r, $g, $b )
    {
        $this->colors[ $name ] = imagecolorallocate( $this->gdo, $r, $g, $b );
    }

    public function getGraphicObject()
    {
        return $this->gdo;
    }

    public function getColor( $name )
    {
        return $this->colors[ $name ];
    }
}
```
public function saveAsPng( $filename )
{
    imagepng( $this->gdo, $filename );
}

public function tx( $x )
{
    return $x * $this->width;
}

public function ty( $y )
{
    return $y * $this->height;
}

abstract class GraphicsObject
{
    abstract public function render( $ge );
    abstract public function z();
}

function zsort( $a, $b )
{
    if ( $a->z() < $b->z() ) return -1;
    if ( $a->z() > $b->z() ) return 1;
    return 0;
}

class Group extends GraphicsObject
{
    private $z;
    protected $members = array();
    public function __construct( $z )
    {
        $this->z = $z;
    }
    public function add( $member )
    {
        $this->members []= $member;
    }
    public function render( $ge )
    {
        usort( $this->members, "zsort" );
        foreach( $this->members as $gobj )
        {
            $gobj->render( $ge );
        }
    }
    public function z() { return $this->z; }
}

abstract class BoxObject extends GraphicsObject
{
    protected $color;
    protected $sx;
    protected $sy;
    protected $ex;
    protected $ey;
    protected $z;

    public function __construct( $z, $color, $sx, $sy, $ex, $ey )
    {
        $this->z = $z;
        $this->color = $color;
    }
}
$this->sx = $sx;
$this->sy = $sy;
$this->ex = $ex;
$this->ey = $ey;
}

public function render( $ge )
{
    $rsx = $ge->tx( $this->sx );
    $rsy = $ge->ty( $this->sy );
    $rex = $ge->tx( $this->ex );
    $rey = $ge->ty( $this->ey );
    $this->draw( $rsx, $rsy, $rex, $rey,
        $ge->getGraphicObject(),
        $ge->getColor( $this->color ) );
}

abstract public function draw( $sx, $sy, 
    $ex, $ey, $gobj, $color );

public function z() { return $this->z; }
}

class Line extends BoxObject
{
    public function draw( $sx, $sy, $ex, $ey,
        $gobj, $color )
    {
        imageline( $gobj, $sx, $sy, $ex, $ey, 
            $color );
    }
}

class Rectangle extends BoxObject
{
    public function draw( $sx, $sy, $ex, $ey,
        $gobj, $color )
    {
        imagefilledrectangle( $gobj, $sx, $sy, 
            $ex, $ey, $color );
    }
}

class Oval extends BoxObject
{
    public function draw( $sx, $sy, $ex, $ey,
        $gobj, $color )
    {
        $w = $ex - $sx;
        $h = $ey - $sy;
        imagefilledellipse( $gobj,
            $sx + ( $w / 2 ), $sy + ( $h / 2 ),
            $w, $h, $color );
    }
}?

The translation code in the GraphicsEnvironment class is highlighted, as is the render code in the GraphicsObject, which calls back to the graphic environment for coordinate translation.

The test code changes only a little (see Listing 8). The objects now need to be specified in the viewport between 0,0 and 1,1.
Listing 8. The test code with the new viewport coordinates

```php
$g1 = new Group( 0 );
$g1->add( new Oval( 200, "red", 0.1, 0.1, 0.5, 0.5 ) );
$g1->add( new Rectangle( 100, "black", 0.4, 0.4, 0.9, 0.9 ) );
```

This is all well and good, but you don't really want a viewport between 0,0 and 1,1. You want any arbitrary viewport -- for example, between -1000,-1000 and 1000,1000. To get that to work, the graphics environment needs to know the start and end coordinates of the viewport.

Figure 8 shows the upgraded `GraphicsEnvironment` class with member variables that store the starting and ending coordinates of the viewport as `vsx`, `vsy` and `vex`, `vey`. The graphics objects don't need to change.

**Figure 8. A graphics environment with a flexible viewport specification**

Listing 9 shows the upgraded `GraphicsEnvironment` code.

Listing 9. The updated `GraphicsEnvironment`

```php
class GraphicsEnvironment
{
    public $vsx;
    public $vsy;
    public $vex;
    public $vey;
    public $width;
    public $height;
    public $gdo;
    public $colors = array();

    public function __construct( $width, $height, $vsx, $vsy, $vex, $vey )
    {
        $this->vsx = $vsx;
        $this->vsy = $vsy;
        $this->vex = $vex;
        $this->vey = $vey;
        $this->width = $width;
        $this->height = $height;
        $this->gdo = imagecreatetruecolor( $width, $height );
        $this->addColor( "white", 255, 255, 255 );
        imagefilledrectangle( $this->gdo, 0, 0, $this->width, $this->height, $this->getColor( "white" ) );
    }

    public function width() { return $this->width; }
}
```
Now the constructor takes four additional parameters, which are the starting and ending points of the viewport. The $tx$ and $ty$ functions use these new viewport coordinates to translate viewport coordinates into physical coordinates.

The code for the test is shown in Listing 10.

**Listing 10. The viewport test code**

```php
<?php
require_once( "glib.php" );

$ge = new GraphicsEnvironment( 400, 400,
   -1000, -1000, 1000, 1000 );

$ge->addColor( "black", 0, 0, 0 );
$ge->addColor( "red", 255, 0, 0 );
$ge->addColor( "green", 0, 255, 0 );
$ge->addColor( "blue", 0, 0, 255 );

$g1 = new Group( 0 );
$g1->add( new Oval( 200, "red", -800, -800, 0, 0 ) );
$g1->add( new Rectangle( 100, "black", -400, -400, 900, 900 ) );

$g1->render( $ge );
$ge->saveAsPng( "test.png" );
?>
The test creates a viewport between -1000,-1000 and 1000,000. The objects are repositioned to fit into the new coordinate system.

The output of the test looks like Figure 9.

**Figure 9. The viewport drawn image translated to a 400-by-400 image**

If you change the image size to 400 by 200 like this:

```php
$ge = new GraphicsEnvironment(400, 200, -1000, -1000, 1000, 1000);
```

you get an image that is scaled vertically, like Figure 10.
Figure 10. The 400-by-200 version of the graphic

This demonstrates how the code automatically adjusts the image to fit into the requested image.

Creating a viewport

Dynamic graphics can add a new level of interactivity to your application. Using object-oriented systems like these can make building complex graphics much easier than building the graphics using the primitives in the standard PHP library. Plus, you have the option of rendering to any size or type of image you like and the longer-term ability to use the same code to render to different types of media, such as SVG, PDF, Flash, and others.
Related topics

- **PHP.net** is the place to learn about the latest with PHP.
- Visit the **GD Graphics Library**.
- The book *Computer Graphics: Principles and Practice in C* (Addison-Wesley Professional, 1995), by James Foley and Andries van Dam, is an excellent resource for graphics work.
- Adobe's **Scalable Vector Graphics** is an interesting place to look for information on graphics.
- To learn more about PHP, visit developerWorks' **PHP project resources**.
- Innovate your next open source development project with **IBM trial software**, available for download or on DVD.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)