

Paint 3-D images with PHP

Skill Level: Intermediate

[Mike Brittain \(mike@mikebrittain.com\)](mailto:mike@mikebrittain.com)

Director of Technology

ID Society

28 Mar 2006

PHP, a language originally intended for Web development, has been used for years to manage dynamic Web sites and database applications. Extensions to the language available through the PHP Extension and Application Repository (PEAR) have allowed developers to take the language in new and interesting directions. PEAR's Image_3D package is an object-oriented interface for creating three-dimensional (3-D) graphics in a variety of formats, including PNG and SVG, two image formats with increasing support by modern Web browsers. Find out how to use the Image_3D package, learn the limitations of using dynamic 3-D images, and investigate solutions and practical applications of 3-D graphics.

Section 1. Before you start

This tutorial is for PHP programmers interested in dynamically generating 3-D graphics. Images can be created from scratch, built up from objects and light sources positioned in space according to X, Y, and Z coordinates. Programmers with experience using 3-D Studio Max will see how to import objects from 3DS files into PHP creations. Complex surfaces defined by parametric equations can easily be mapped using Image_3D. Finally, business applications and data reports can be enhanced by adding pie charts generated on the fly.

About this tutorial

Image_3D is an object-oriented interface for creating 3-D images. Objects and lights are positioned in a 3-D space according to X, Y, and Z coordinates. Images are then rendered into 2-D space and can be stored as PNG, SVG, or output to the shell (for

use on ANSI shells). The package can be used to easily generate a handful of simple 3-D objects, including cubes, cones, spheres, text, and pie graphs. In addition, there is support for importing and modifying objects created in 3-D Studio Max. Developers with a strong understanding of 3-D spaces can take advantage of the custom polygram and surface maps to create some interesting objects.

The first half of this tutorial will demonstrate how to use command-line PHP scripts to generate 3-D image files. After setting up a basic 3-D space with colors and lights, each unique 3-D object type will be investigated, as well as each of the output file formats. The second half will discuss how to turn these basic examples into practical applications. Generating 3-D images is a processor-intensive task, so you will identify a solution for this problem in order to take dynamically generated images to the Web, without crashing your Web servers. The Image_3D package supports only static image formats, so you will build a simple JavaScript solution for animating the 3-D spaces. Finally, you will write a PHP class for displaying data reports in colorful pie charts that can be integrated with business applications.

System requirements

The following software and tools are required to follow along:

PHP V5

The Image_3D package is written using the PHP V5 object and class syntax.

Image_3D

The PEAR package will need to be installed. Typically, the installation of Image_3D should be easy if you have root (or administrator) access to your machine. Because Image_3D is in alpha release, the PEAR installation may complain that the package is not "stable." Use the `-f` option to force the installation:

```
pear install -f Image_3D
```

GD

This graphics library is required to output PNG files, though other file types can be generated from Image_3D in the absence of GD. As of PHP V4.3, a version of the GD library comes bundled with new PHP installations. So there's a good chance you won't need to worry about recompiling PHP. You can use `phpinfo()` to see if your current PHP installation is GD-enabled.

Web server

Examples in the second half of the tutorial can be built for use on a Web site. The [Apache's open source Web server](#) and [IBM HTTP Server](#) are two options, if you don't already have a Web server available.

SVG viewer

SVG files can be viewed in [Mozilla Firefox V1.5](#), which includes native SVG support. Internet Explorer users can add SVG support to the browser by downloading the [Adobe SVG plug-in](#).

Prerequisites

This tutorial assumes at least a base knowledge of objects and classes, as the entire interface for Image_3D is object-oriented. Access to a Linux® shell account or a Windows® command prompt is required for executing command-line PHP examples shown in the first half of the tutorial. Basic experience with JavaScript is also assumed.

Section 2. Laying out your space

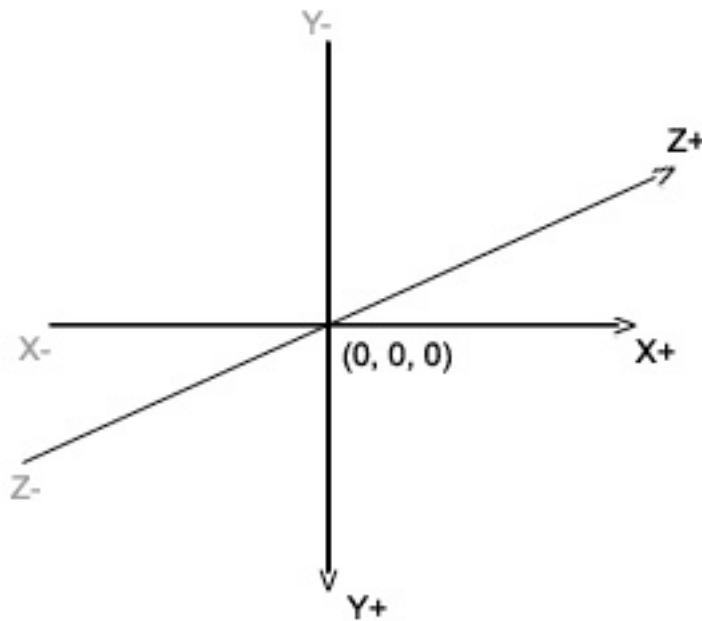
It's probably safe to say that most PHP developers have a fair understanding of 2-D graphics. Most have probably used Adobe Photoshop, Corel Paint Shop Pro, GIMP, or some other program to create at least a few basic graphics. Let's get started by discussing how a 3-D space, or *world*, correlates to the familiar 2-D canvas. You'll also investigate tools such as lights, colors, and transformations that will help you design your 3-D images.

Getting oriented

Typical graphics programs work with X- and Y-axes when displaying a bitmap. Each pixel is positioned along these axes. The origin point ($X=0$, $Y=0$) is located in the upper-left corner. A pixel located at 30, 20, is 30 units (usually pixels) to the right of the origin, and 20 units down.

When working in your 3-D space, the origin will represent the center of the space, rather than a corner. The third axis, Z, is mutually perpendicular to the X- and Y-axes. Consider the positive orientation of the Z-axis to be into the screen of your computer. The negative orientation is away from the screen. Figure 1 illustrates the direction of each axis.

Figure 1. 3-D coordinate system



As you create 3-D objects within the coordinate system, each object or point will be measured in positive or negative units from the origin.

Your 3-D tool set

You will be creating PHP objects for each of the objects (cones, spheres, etc.) you place into your 3-D space. Additionally, you will create PHP objects representing light sources within the space. Color objects will be used to modify each shape or light source and can be used to adjust the alpha-transparency of an object. Matrix objects will be created to transform the size, rotation, or position of each object, or even transform the entire space as a whole.

Section 3. Creating your first world

Your first image will include all the basic elements required for any image created with Image_3D. From this example, you'll have the right space and lighting to quickly examine all of the basic objects available in the package.

Cones

The goal of this first example will be to create a lighted cone-shaped object with a white background. The resulting space will be rendered into a 400x400-pixel image.

Listing 1. Creating a cone in a 3-D space

```
<?php
require_once('Image/3D.php');

// Create the blank three-dimensional space
$world = new Image_3D();
$world->setColor(new Image_3D_Color(255,
255, 255));

// A blue light from the left
$light1 = $world->createLight(-300, 0,
-300);
$light1->setColor(new Image_3D_Color(100,
100, 255));

// A green light from the upper-right
$light2 = $world->createLight(300, -300,
-300);
$light2->setColor(new Image_3D_Color(100,
255, 100));

// Build the cone object
$cone = $world->createObject('cone',
array('detail' => 64));
$cone->setColor(new Image_3D_Color(255, 255,
255));
$cone->transform($world->createMatrix\
('scale', array(70, 220, 70)));
$cone->transform($world->createMatrix('rotation',
array(-45, -120, -10)));
$cone->transform($world->createMatrix\
('move', array(-50, -30, 10)));

// Render and save the 2-D image
$world->createRenderer('perspectively');
$world->createDriver('gd');
$world->render(400, 400, 'object.png');
?>
```

Every image you create begins by creating a new `Image_3D` object, which you will call your `$world`. The background color is set for the space by creating an `Image_3D_Color` object. The three parameters for this object's constructor are the RGB values for the color you want to create.

You create two `Image_3D_Light` objects within the space. Imagine that the first -- a blue light -- is positioned to the left of the origin and away from the plane of the screen. The second is positioned to the upper right of the origin and also away from the screen. This one is green.

Next, a cone is positioned in the space. Note that the color you assign to this object

is white. This is so the object takes on the colors of the two lights you positioned in the space -- the best method for highlighting opposing aspects of the object with different colors.

Three transformations are applied to the cone. First, the scale of the object is increased. The cone object is unique in the respect that it is not assigned a size when it is created. It essentially is created in a 1x1x1 space and needs to be resized. Second, rotate the cone so it captures the lights well. This is somewhat of a trial-and-error process. Then you move the cone, simply to center it on the image.

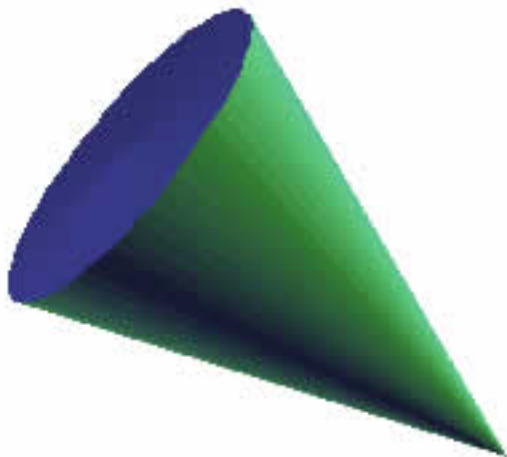
The last step for your script is to set up the 2-D rendering of the 3-D space. You will use the "perspectively" rendering engine in all of your examples (the other option is "isometric"). The `gd` output driver is selected, which will create a PNG image. Set the size of the image to 400x400 pixels, and it will be saved to the filename: `object.png`.

The script should be run from the command line to create the image:

```
php -f build_cone.php
```

The generated image file should appear as shown below.

Figure 2. 3-D cone generated using Image_3D package



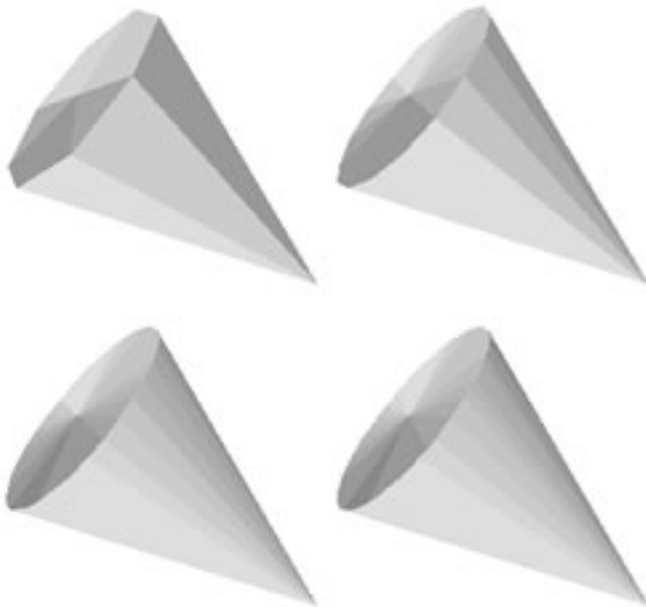
Controlling detail and rendering time

I hope that example was pretty straightforward. You will use the same space,

lighting, and rendering over a handful of additional examples to showcase the other built-in object types that come with Image_3D.

Before moving on, there are two points to make about your first script. When instantiating the cone object, you passed an array as the second argument, `array('detail' => 64)`, which set the number of polygons used to generate the cone. A lower number causes a more angular look, while a higher number generates a much smoother surface (see Figure 3).

Figure 3. Various levels of detail in 3-D cones. Detail levels shown are 8, 16, 32, and 64



With a higher level of detail, Image_3D will take a longer time to generate the image. Cones are fairly simple objects to build, so the effect is not horrible. When you get into spheres and more complex shapes, you'll see that the rendering speed can quickly grow out of control.

There is a built-in statistics method for the Image_3D object that will provide some information about how the image was generated. Put the following line at the end of the previous script.

```
echo $world->stats();
```

Rerunning the script should display the output in Listing 2.

Listing 2. Statistics for generating a cone object

```
Image 3D
Objects: 1
Lights: 2
Polygons: 130
Points: 67
```

Aside from these simple details, consider using the PEAR::Benchmark package (see [Resources](#)) to measure the execution time for your scripts as they generate more detailed 3-D images.

Section 4. Light sources and color

Without color or light, you are unable to see the objects in your 3-D space. Let's take a moment to investigate these two important items which allow us to see your creations.

Turn up the lights

If you were to adjust the color of the cone in the example above to red (255, 0, 0) and remove the two light sources, the resulting image would show a black cone against a white background. The red surface of the cone will not be represented without a light source shining on it.

You can easily remedy this by adding at least one light source -- and it's best if this is a white light that will best bring out the color of the object.

A single light source will only lighten one side of the object, or objects, in your space. Adding another light to the opposite side will help us see more of the objects. If you use two white lights, they won't provide much character to the object, as the contrasting angles will only take on varying shades of the object's color.

As discussed, you will instead create white-colored objects in these examples and apply colored lights to them. This allows for a greater range of shades and hues to bring out the features of each object.

Controlling colors

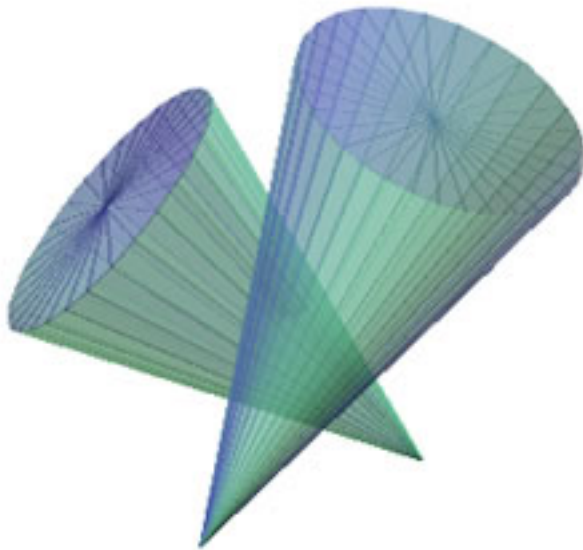
In the example above, you set the color of the cone object using the `Image_3D_Color` class. The first three parameters for the constructor are RGB values for the desired color. There is a fourth parameter, `alpha` transparency,

which was not shown. Adding the value, 150, to your script produces a cone you can partially see through.

```
$cone->setColor(new Image_3D_Color(255, 255, 255, 150));
```

Figure 4 shows the effect of transparency on the cone.

Figure 4. Two overlaid 3-D cone objects, both have alpha-transparency of 150 applied



Buff up the chrome

There is one more effect that can be applied to RGB colors: a chrome finish! Rather than supplying the standard `Image_3D_Color` object as the color for the cone, create a new `Image_3D_Color_Metal` object (don't forget to include `Image/3D/Color/Metal.php`, it's not included by `Image/3D.php` as other classes are). See Listing 3.

Listing 3. Applying a metal finish to 3-D objects

```
require_once('Image/3D/Color/Metal.php');

$cone = $world->createObject('cone', array('detail' => 64));
$metal = new Image_3D_Color_Metal(255,255,255);
$metal->setMetal(1.25);
$cone->setColor($metal);
```

The `setMetal()` method takes a float as the parameter. A little experimentation showed that values between 0.5 and 2.0 had the best effects. Anything over 2.0 seemed to wash out completely. Experiment with this value to how the lighting and color of the cone is affected.

Section 5. Modifying objects and shapes

If you've ever used the transform tool in Adobe Photoshop, you're probably familiar with its ability to move, scale, rotate, flip, and skew elements. The `transform()` method for 3-D objects and spaces is analogous to that tool.

Transformation matrices

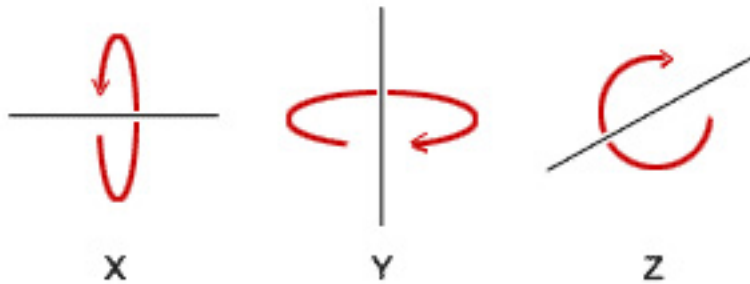
That sure seems like a frightening term. In practice, a *transformation matrix* is really just a control for moving, scaling, or rotating an object. This is accomplished by first creating the `Image_3D_Matrix`, then passing it to a `transform()` method. Not so bad, after all.

Here are three examples (note that each matrix takes an array with three values as the second parameter):

```
$obj->transform($world->createMatrix\  
( 'move', array(-50, -30, 10)));  
$obj->transform($world->createMatrix\  
( 'scale', array(70, 220, 70)));  
$obj->transform($world->createMatrix\  
( 'rotation', array(45, 0, 0)));
```

The values of the array for the 'move' matrix indicate the movement along the X-, Y-, and Z-axes. The values applied to 'scale' indicate the degree of scaling along the same three axes. The values associated with 'rotation' indicate the amount of rotation around each axis, from 0 to 360 degrees. Figure 5 illustrates the direction of rotation (remember that negative values could be applied to reverse these directions).

Figure 5. Orientations of rotation around the X-, Y-, and Z-axes



When and where to apply transformations

Each of these transformations can be applied to an individual object or to the entire 3-D space. You saw how a cone could be rotated, scaled, and moved within the space of the first example image you created. If you had applied the transformations to `$world`, instead, they would have affected everything within the 3-D space; all objects and lights in the space would be affected as one big room.

Transformations are applied to each object in the order specified. This may seem obvious, but if you are applying many transformations to individual objects, and to the space as a whole, you may get unexpected results if you are careless about the order in which you add your transformations.

Section 6. Additional objects

You've covered a lot of details about the lighting, color, and positioning of your cone. Let's get back to seeing what other types of objects can be created in `Image_3D`.

Cubes

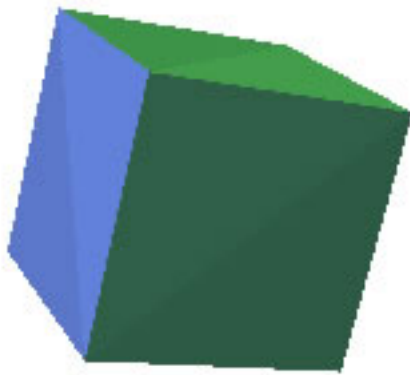
Like cones, cubes are another simple object to create. Only a small number of polygons are needed to build a cube, and all cubes have the same level of detail no matter what their size. When creating a cube, the second parameter for `createObject()` is an array of three values: the width, height, and depth of the cube.

The lines below can be used in place of the `$cone` object in [Listing 1](#) to produce a cube:

```
$cube = $world->createObject('cube',  
array(100, 100, 100));  
$cube->setColor(new Image_3D_Color(255,  
255, 255));  
$cube->transform($world->\  
createMatrix('rotation',  
array(-60,60,40)));
```

The resulting image:

Figure 6. 3-D cube generated using Image_3D package

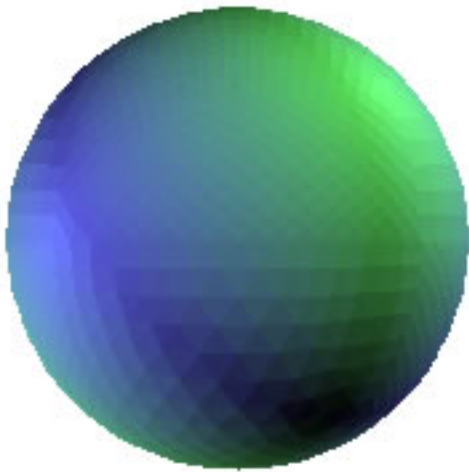


Spheres

Continuing again by editing your original example from [Listing 1](#), you'll replace the cone with a sphere. Here, the second parameter will be a hash containing a value for `r`, which represents the radius of the sphere, and `detail`, which is an integer that defines the relative smoothness in the object. Detail can be as low as 1, which looks more like a crumpled piece of paper than a sphere, and the quality and rendering time increases from there. Using a value of 4, 5, or 6 works best for detail. Anything higher than 6 causes extremely long rendering times (measured in minutes, rather than seconds).

```
$sphere = $world->createObject('sphere',  
array('r' => 85, 'detail' => 5));  
$sphere->setColor(new Image_3D_Color(255, 255, 255));
```

The resulting image:

Figure 7. 3-D sphere generated using Image_3D package

Compare the statistics reported for generating this sphere (see Listing 4) with those seen for the cone in [Listing 2](#).

Listing 4. Statistics for generating sphere object with detail of 5

```
Image 3D
Objects:  1
Lights:   2
Polygons: 4096
Points:   2050
```

In fact, if you increase the detail of this sphere to 6, the numbers of polygons and points quadruple.

Torus

According to [Wikipedia](#), a *torus* is a closed surface defined as the product of two circles: $S^1 \times S^1$. If your geometry is rusty, you can refer to it simply as a donut.

Drop the following code into your 3-D space to see how a torus is generated.

Listing 5. Instantiating a torus

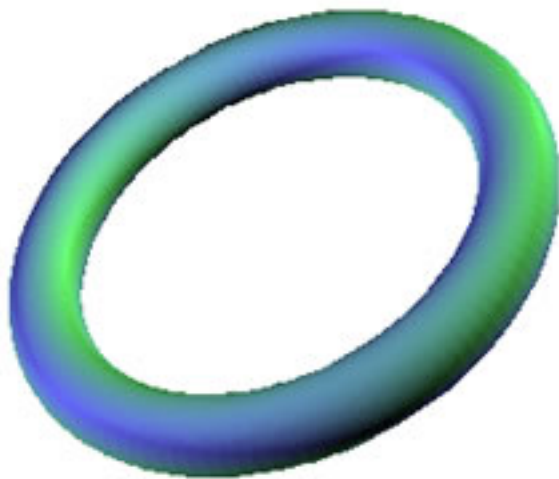
```
$torus = $world->createObject('torus', array('inner_radius' => 90,
                                             'outer_radius' => 120,
```

```
        'detail_1' => 30,  
        'detail_2' => 30));  
$torus->setColor(new Image_3D_Color(255, 255, 255));  
$torus->transform($world->createMatrix('Rotation', array(-45,0,-30)));  
$torus->transform($world->createMatrix('Move', array(0,-20,0)));
```

Once again, you have another set of values passed to the second parameter of `createObject()`. In this case, `inner_radius` and `outer_radius` are aptly named for the inside and outside radius of the donut. `Detail_1` and `detail_2` are somewhat less obvious. If you experiment with adjusting these values, you'll find that `detail_1` controls the amount of banding (lines perpendicular to the ring), and `detail_2` controls the amount of striping (lines that run parallel to the outer edge of the ring).

The resulting image appears:

Figure 8. 3-D torus generated using Image_3D package



Text

Text can also be rendered in three dimensions using the `Image_3D` package. Unfortunately, there are very few options available for styling the text. For example, there is no way to select a font; you're stuck with the single font that is built into the package. Regardless, Listing 6 shows how to create a text object.

Listing 6. Instantiating a text object

```
$text = $world->createObject('text', 'Databases!');
```

```
$text->setColor(new Image_3D_Color(255, 255, 255));  
$text->transform($world->createMatrix('Scale', array(6, 6, 6)));  
$text->transform($world->createMatrix('Rotation',  
    array(-35, 30, -15)));  
$text->transform($world->createMatrix('Move', array(-150, 10, 20)));
```

In place of the array in previous examples passed into the second parameter of `createObject()`, to create a text object, you simply need to supply the text to be displayed.

Incredibly, you've gotten halfway into a PHP tutorial and there hasn't been a single mention of the ubiquitous systems about which so many PHP tutorials are written. A 3-D homage to these yet unmentioned systems is shown below.

Figure 9. 3-D text generated using Image_3D package



3D Studio Max

The tools you've seen so far for creating 3-D images have been fairly basic, and there's not a whole lot you could do building images from scratch. The ability to import pre-made 3D Studio Max files (see Listing 7) adds some exciting potential.

Listing 7. Instantiating a 3D Studio Max file

```
$obj = $world->createObject('3ds', 'Image_3D.3ds');  
$obj->setColor(new Image_3D_Color(255, 255, 255));  
$obj->transform($world->createMatrix('Rotation', array(110, 40, 0)));  
$obj->transform($world->createMatrix('Scale', array(7, 7, 7)));
```

As you can see, creating the object is about as simple as locating the 3D Studio Max file. If you've spent any time digging through the Image_3D installation, you may have located some examples included with the PEAR package. The Image_3D.3ds file imported in the listing above is taken from those examples. The resulting image is shown below.

Figure 10. 3D Studio Max file imported and rendered using Image_3D package



Of course there's a catch: The 3D Studio Max file imported into the example above was less than 200 KB and is fairly basic. Opening larger files can cause your server or PHP to quickly grind to a halt.

Combining multiple objects in one space

So far, you've only looked at single objects in a 3-D space. But there's no reason to stop there. In Listing 8, you'll look at an example of how to combine more than 100 objects in the same space.

As before, this code simply replaces the lines in [Listing 1](#) that define the `$cone`.

Listing 8. Generating a cube assembled from sphere objects

```
for ($x=0; $x < 5; $x++) {
  for ($y=0; $y < 5; $y++) {
    for ($z=0; $z < 5; $z++) {
      $sphere = $world->createObject('sphere',
        array('r' => 25, 'detail' => 4));
      $sphere->setColor(new Image_3D_Color(255, 255, 255));
      $sphere->transform($world->createMatrix('Move',
        array($x * 75, $y * 75, $z * 75)));
      $sphere->transform($world->createMatrix('Rotation',
        array(45, 30, 15)));
    }
  }
}

$world->transform($world->createMatrix('Move', array(-225, -100, 0)));
```

Listing 8 defines three loops that increment `$x`, `$y`, and `$z` from 0-4. Inside the middle loop, a single sphere object is created. Using a `move` transformation, each new sphere is offset from the origin in the X, Y, and Z directions. After being moved, each sphere is rotated about the origin.

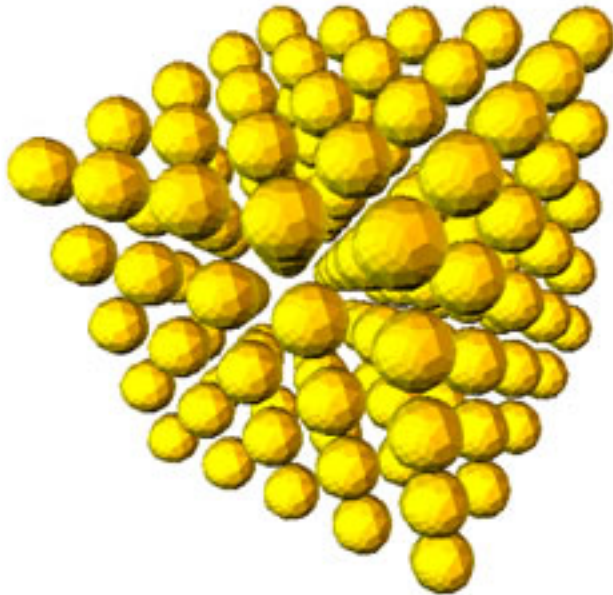
After each of the spheres is created and all three loops are complete, the entire `$world` is shifted to center the spheres within the image boundaries.

Keep in mind that this example is generating 125 spheres. If you had trouble generating one sphere at a detail of 4, consider how long it will take to generate 124

more of them. A detail value of 4 or 5 will look really great, but it makes more sense to dial this back to 1 or 2 until you know everything is working as expected.

You may agree that the resulting image, shown in Figure 11, is much more exciting than simple cubes and cones.

Figure 11. 125 spheres are arranged to create a cube-shaped matrix



Section 7. Custom shapes and surfaces

There are two objects that allow for more custom object generation in Image_3D: polygons and maps.

Building polygons

Assuming you can map out a 3-D shape with multidimensional coordinates, you can put these coordinates together to build the sides of a 3-D object. Using some careful calculation, you could build cubes, cones, or spheres from scratch using a number of points and polygons. But there are easier methods for creating those objects.

Instead, let's build a three-pointed star that has a third dimension: depth. The star will need a front and back face, which will have essentially the same coordinate points defining the outer edges. Six points will be needed -- three for the points of

the star and three used to pinch in the sides. To make things easy, your front face will have points that all lie at Z=0. The back face will be a constant distance, 60, from the front face. So all the outer points will have the same X and Y coordinates, but the Z coordinate will be changed to 60.

Three sides will be mapped out to connect between each of the star's outer points.

The majority of Listing 9 deals with laying out the points that need to be connected to create this object. The `foreach` loops that follow are used to build `Image_3D_Point` objects, which in turn are used to create the final polygon object.

The last line rotates the entire `$world` so you can see the side of the star. If you don't rotate the space, you'll only see the front face of the star in two dimensions.

Listing 9. Building a three-pointed star

```
$polygons = array();

// Front face
$polygons[] = array(
    array(0, -120, 0), array(-18, -12, 0),
    array(-86, 48, 0), array(0, 18, 0),
    array(86, 48, 0), array(18, -12, 0)
);

// Back face
$polygons[] = array(
    array(0, -120, 60), array(-18, -12, 60),
    array(-86, 48, 60), array(0, 18, 60),
    array(86, 48, 60), array(18, -12, 60)
);

// 3 Sides
$polygons[] = array(
    array(0, -120, 0), array(-18, -12, 0),
    array(-86, 48, 0), array(-86, 48, 60),
    array(-18, -12, 60), array(0, -120, 60)
);
$polygons[] = array(
    array(-86, 48, 0), array(0, 18, 0),
    array(86, 48, 0), array(86, 48, 60),
    array(0, 18, 60), array(-86, 48, 60)
);
$polygons[] = array(
    array(86, 48, 0), array(18, -12, 0),
    array(0, -120, 0), array(0, -120, 60),
    array(18, -12, 60), array(86, 48, 60)
);

foreach ($polygons as $poly) {
    $points = array();
    foreach ($poly as $set) {
        $points[] = new Image_3D_Point($set[0], $set[1], $set[2]);
    }
    $p = $world->createObject('polygon', $points);
    $p->setColor(new Image_3D_Color(255, 255, 255));
}

$world->transform($world->createMatrix('Rotation',
    array(0, -25, -15)));
```

This listing creates the image shown in Figure 12.

Figure 12. A custom polygram generated using Image_3D package



Mapping 3-D surfaces

The next two examples create some interesting shapes, but also involve a bit of calculus. If you have never taken calculus or are a bit rusty, this is not exactly the forum to brush up the skills. However, an attempt will be made to give a basic idea of what is going on in the models below.

If you've ever had experience plotting surfaces and curves using the Mathematica software, these next examples will probably look familiar to you.

Helicoid

The first of these two listings will create what appears to be a spiraling ramp. Every point in the surface has X-, Y-, and Z-coordinates derived from the following parametric function:

```
f[s,t] = {s * cos(2 * t), s * sin(2 * t), t}
```

To create this surface, you will create a new "map" object in your space.

```
$map = $world->createObject('map');
```

The map takes an array of `Image_3D_Point` objects, like you saw in the custom polygon above, to generate a 3-D surface. Each of these points could be defined according to your function above:

```
new Image_3D_Point( ($s * cos(2 * pi() * $t)),
    ($s * sin(2 * pi() * $t)),
    ($t) );
```

You will vary the parameters `s` and `t` over the ranges 0-1 and 0-6, respectively. To give the surface a decent size, you will multiply each coordinate by an arbitrary value you assign to `$scale`.

Listing 10 shows the generation of the map object, and, as with other examples, you will set the color, rotation, and position of the final object.

Listing 10. A helicoidal parameterized surface

```
$map = $world->createObject('map');

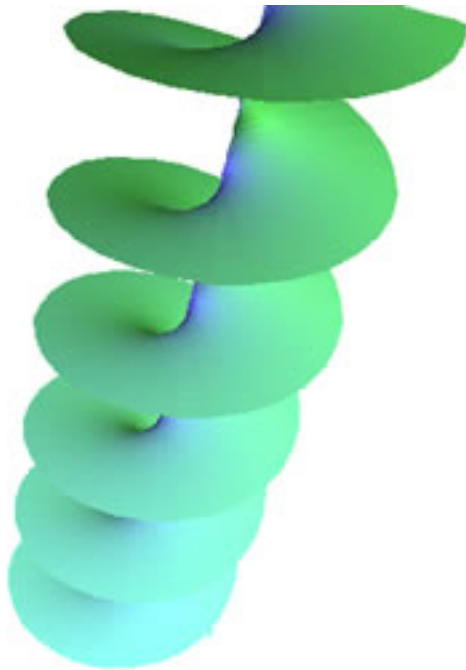
$scale = 120;
$detail = 30;
$levels = 6;
$increment = 1 / $detail;

for ($s = 0; $s <= 1; $s += $increment) {
    $row = array();
    for ($t = 0; $t <= $levels; $t += $increment) {
        $row[] = new Image_3D_Point(
            $scale * ($s * cos(2 * pi() * $t)), // x
            $scale * ($s * sin(2 * pi() * $t)), // y
            $scale * ($t) // z
        );
    }
    $map->addRow($row);
}

$map->setColor(new Image_3D_Color(255, 255, 255));
$map->transform($world->createMatrix('Rotation', array(-50, 0, 15)));
$map->transform($world->createMatrix('Move', array(50, -220, 0)));
```

Figure 13 illustrates the resulting image. The levels of detail, scale, and height of the helicoid can be manipulated by the `$scale`, `$detail`, and `$levels` variables.

Figure 13. Helicoid generated using map object of Image_3D package



The bundt pan

Your second surface integral, an application of Stokes' Theorem, also has an interesting result. You might describe the surface as the product of a sine wave that has a varying height, Z , and radiates in every direction from the origin of the X and Y plane.

In a noncalculus manner of speaking, if a torus can be described as a donut, you may think this looks somewhat like a bundt pan.

This surface will be described according to the following parametric function:

```
f[r,t] = {r * cos(t), r * sin(t), sin(4 * pi * r)}
```

Again, you will plot points for this surface to be added into the map object. Each point is defined in Listing 11.

Listing 11. Defining the points

```
new Image_3D_Point( ($r * cos($t)),
                   ($r * sin($t)),
                   (sin(4 * pi() * $r))
                   );
```

You will wrap each point inside two loops that vary the parameters r and t to produce a smooth surface. Each X -, Y -, and Z -coordinate will be multiplied by a

\$scale value, as shown in Listing 12.

Listing 12. A 3-D bundt pan

```
$map = $world->createObject('map');

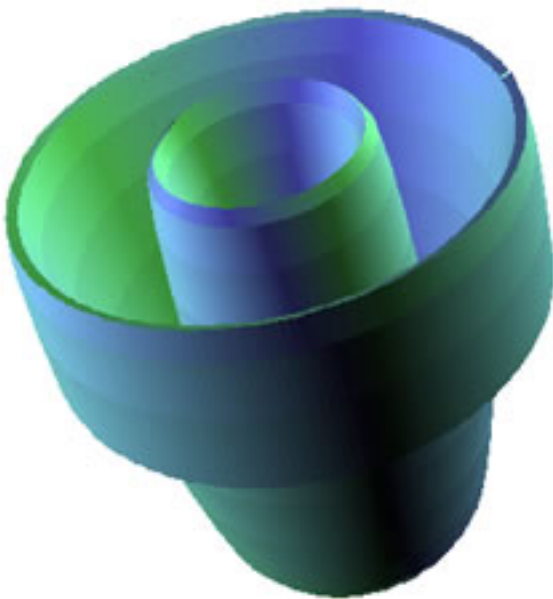
$scale = 130;
$detail = 35;
$increment = 1 / $detail;

for ($r = 0; $r <= 1; $r += $increment) {
    $row = array();
    for ($t = 0; $t <= (2 * pi()); $t += $increment) {
        $row[] = new Image_3D_Point( $scale * ($r * cos($t)),
                                     $scale * ($r * sin($t)),
                                     $scale * (sin(4 * pi() * $r))
        );
    }
    $map->addRow($row);
}

$map->setColor(new Image_3D_Color(255, 255, 255));
$map->transform($world->createMatrix('Rotation', array(-45, 0, -15)));
```

The resulting image:

Figure 14. 3-D surface generated from parametric function built using map object from Image_3D package



Section 8. Additional output drivers

So far, all the images you have created use the GD driver. You have seen that this driver produces PNG images. But there are four additional drivers at your disposal: SVG, SVGRotate, ZBuffer, and ASCII.

SVG

The Scalable Vector Graphics (SVG) file format is an XML definition of vectors that make up 2-D images. The W3C standardized the format in 2001, but its use on the Web has been hampered by the availability of browsers that can display SVG files. Currently, the best options are Firefox V1.5, which has built-in support for SVG, and Internet Explorer with the Adobe SVG plug-in.

Creating SVG images from Image_3D is simple enough. Just change the last two lines of code in [Listing 1](#) to read as follows:

```
$world->createDriver('svg');  
$world->render(400, 400, 'object.svg');
```

The resulting image file should closely resemble the first cone you created.

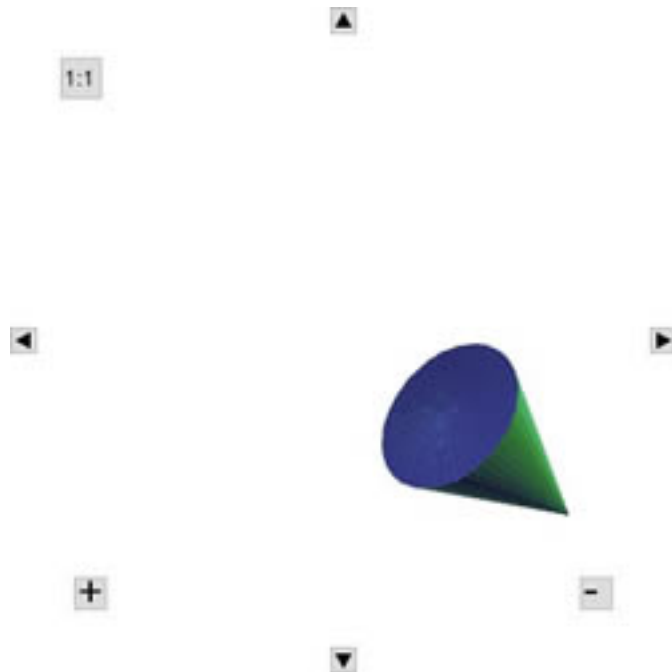
An exciting aspect of SVG files is that the XML tree that defines the image can be manipulated using JavaScript and DOM parsing routines. Imagine manipulating images in the same manner that dHTML is used to manipulate Web pages. It happens that the SVGRotate driver generates an SVG file that includes the script needed for just that purpose.

Changing the output driver is, again, a very simple change in the last two lines of code:

```
$world->createDriver('svgrotate');  
$world->render(400, 400, 'object.svg');
```

The resulting file, when viewed in an SVG-aware browser, includes built-in controls for rotating and scaling the cone in real time. A screenshot of the image is shown below.

Figure 15. SVGRotate driver includes controls and scripting to manipulate the image in real time



Note that support for scripted SVG files is limited. The resulting file worked in Internet Explorer with the Adobe SVG plug-in, but did not display correctly in Firefox.

ZBuffer

Conversion of 3-D spaces to a 2-D image requires a special algorithm that can determine which objects overlay and hide others. In 3-D modeling, management of the visibility of objects in a 2-D space is referred to as *Z-buffering*.

So far, none of your examples have had problems with objects overlaying each other. If you run into this issue, the best option is to change your output driver from GD to ZBuffer. The ZBuffer driver still uses GD to produce a PNG image, but it does better management of the 3-D objects because they are painted into a 2-D bitmap.

Using the code in Listing 13, you will create two objects that take up the same space, intentionally causing a conflict in the Z-buffering of the objects. The sphere is given an alpha-transparency of 150 so you can see the plane that slices through the center of it.

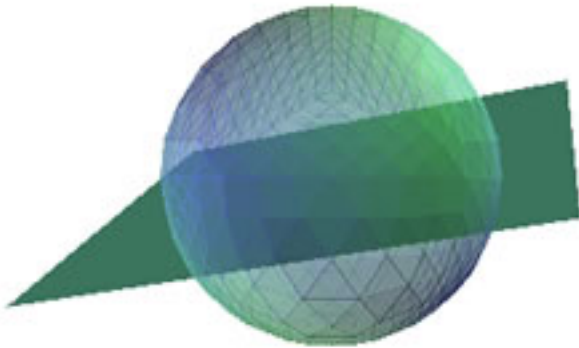
Listing 13. A four-sided plane slicing through a sphere

```
$sphere = $world->createObject('sphere',  
                                array('r' => 85, 'detail' => 4));  
$sphere->setColor(new Image_3D_Color(255, 255, 255, 150));  
$plane = $world->createObject('polygon', array(  
                                new Image_3D_Point(-120, 0, -120),  
                                new Image_3D_Point(-120, 0, 120),
```

```
        new Image_3D_Point( 120, 0, 120),  
        new Image_3D_Point( 120, 0, -120)  
    ) );  
$plane->setColor(new Image_3D_Color(255, 255, 255));  
$plane->transform($world->createMatrix('Rotation', array(15,15,-10)));
```

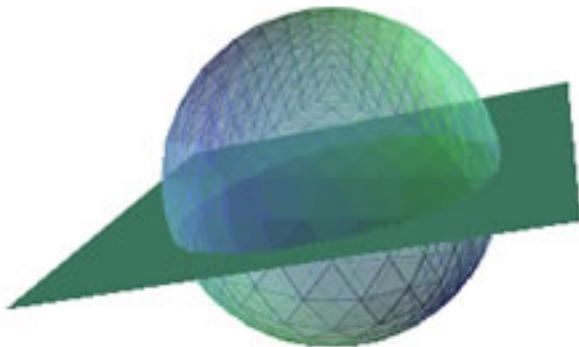
If you render this object using the GD driver, the sphere will appear to sit in front of the plane, as illustrated below.

Figure 16. Two overlapping 3-D objects rendered separately using the GD driver



However, you know from the coordinates of the plane that you created that it should slice directly through the sphere. Changing the driver to ZBuffer will resolve this issue in the resulting PNG file, as shown below.

Figure 17. ZBuffer conflicts between two overlapping 3-D objects resolved by applying the ZBuffer driver



ASCII

The ASCII driver is not included in your examples, but the purpose of this driver is to allow for the creation of images that will be displayed on a color-enabled ANSI terminal. If you're working from a Windows® PC, the output from this driver will

probably not look very useful to you.

Section 9. Turning to practical examples

So far, the examples you have seen demonstrate the capabilities of this package to generate 3-D images in PHP. Who would have guessed that this language, which was invented to manage Web pages, could be used to generate such sophisticated image files? Well that's all fine and good, but unless you're a 3-D wizard or a calculus hound, you may be yawning. Let's take a look at how you can take your simple objects and command-line scripts and generate some more interesting examples.

Animating 3-D images

The GD graphics library has had a checkered past with respect to support of GIF images. Due to licensing concerns, support for GIFs was removed from GD after V1.6, though it has apparently been restored in V2.0.28. In any case, `Image_3D` doesn't export animated GIFs, and PNG files don't support animation.

Rather than creating a single file that will play a looped animation, let's see how you can take multiple files to create a flip-book effect. You'll extend the example with the cube of spheres, generating 30 individual files. When viewed in succession, the resulting animation will show the cube rotating around two axes.

You'll use a single script to generate each of the images: `$cycles`. Just inside the outermost for loop, you will set up the three rotational values that will change for each file, but will be applied to every sphere within the individual files. Your goal is for the animation to loop smoothly, so your rotation should travel through 360 degrees.

For the Y-axis rotation, start with a static rotational value: 30. From there, you add an increment of the 360-degree loop, based on which file you are generating in the sequence. The modulus operator is used to shave off any whole increments of 360 -- a rotation of 375 will be cut back to 15 degrees, staying within the first circle of rotation.

```
$rot_y = (30 + (int) (360 / $cycles * $i)) % 360;
```

The hardest part of generating these images has been covered. The rest of the code should look familiar. You create a `$world` object and give it two light sources. Using

three loops, you create the cube of spheres.

Remember that the first cube of spheres that you constructed had five cubes per side, necessitating the creation of 125 3-D objects. You're also creating 30 of these images -- more than 3,000 spheres! In the code below, you have scaled back to four spheres per side and reduced the detail level of each sphere to three (see Listing 14).

Listing 14. Generating 30 flip-book pages

```
<?php
require_once('Image/3D.php');

$cycles = 30;
for ($i=0; $i < $cycles; $i++) {

    $rot_x = 45;
    $rot_y = (30 + (int) (360 / $cycles * $i)) % 360;
    $rot_z = (15 + (int) (360 / $cycles * $i)) % 360;

    $world = new Image_3D();
    $world->setColor(new Image_3D_Color(255, 255, 255));

    $light1 = $world->createLight(-500, 0, -500);
    $light1->setColor(new Image_3D_Color(255, 255, 0));

    $light2 = $world->createLight(300, -300, -1000);
    $light2->setColor(new Image_3D_Color(255, 162, 0));

    for ($x=0; $x < 4; $x++) {
        for ($y=0; $y < 4; $y++) {
            for ($z=0; $z < 4; $z++) {
                $sphere = $world->createObject('sphere',
                    array('r' => 25, 'detail' => 3));

                $sphere->setColor(new Image_3D_Color(255, 255, 255));

                $sphere->transform($world->createMatrix('Move',
                    array(($x * 75) + 50, $y * 75, $z * 75)));

                $sphere->transform($world->createMatrix('Rotation',
                    array($rot_x, $rot_y, $rot_z)));
            }
        }
    }

    $world->transform($world->createMatrix('Move',
        array(-225, -100, 0)));

    $world->createRenderer('perspectively');
    $world->createDriver('gd');
    $world->render(800, 800, 'animated_png/anim' . ($i+1) . '.png');
}
?>
```

Run the code from the command line and go make a pot of coffee. This will take a while.

The next step is to stitch all these images together. You'll do that by displaying the first image on an HTML page and using JavaScript to swap through the pile of

images. Swapping images on the page is a standard technique, handled by changing the `src` property of an `Image` object. Your JavaScript example below sets up the array of image for the flip book, defines a function for swapping the image, writes the first image to the page, and executes the `animate()` function using the `setInterval()` timer.

Listing 15. HTML and JavaScript used to create a flip book

```
<html>
<head>
<title>Animated PNG</title>
</head>
<body>

<script type="text/javascript">
var imageset = new Object();
imageset.seconds = 0.1;
imageset.imgTag = "animation";
imageset.images = new Array();
for ($i=1; $i <= 30; $i++) {
    imageset.images.push('animation_01/anim' + $i + '.png');
}

function animate(animObj) {
    if (!animObj.index) {
        animObj.index = 0;
    }

    animObj.index++;
    if (animObj.index >= animObj.images.length) {
        animObj.index = 0;
    }

    document.images[ animObj.imgTag ].src =
        animObj.images[ animObj.index ];
}

document.write('');

setInterval("animate(imageset);", imageset.seconds * 1000);
</script>

</body>
</html>
```

The resulting animation (see [Download](#)) brings your static 3-D images to life.

Viewing dynamic images on the Web

One of the difficulties of generating 3-D images is that it can take a lot of time, memory, and processor cycles to create making them inherently bad for use on Web sites. What can you do to help with these problems?

First off, if you decide to run these scripts from a Web browser, you may find that the server connection times out while you are waiting for the `Image_3D` package to

finish generating the image file. You can turn off PHP's maximum execution time by placing the following line near the top of your script:

```
set_time_limit(0);
```

Generating complex 3-D images may also eat up a lot of memory in PHP. Running from the command line shouldn't be a problem, but once you run these scripts over the Web, you may find, again, that your server is closing the connection. Make sure you have allotted enough memory to your PHP scripts. You can typically change the value for `memory_limit` in `php.ini` or a local `.htaccess` file.

In all of the examples, `Image_3D` creates images and saves them to a file. The name of the file can be replaced, however, with one of two PHP output streams:

- `php://stdout`
- `php://output`

This method failed when you tested on a Windows server, though it's possible that the version of GD did not properly support output streams. So if you have trouble with this method, consider saving the image to file, opening the file and piping the contents back to the output buffer. It may not seem like the most efficient solution, but remember that you're generating 3-D images. File I/O is probably not going to be the bottleneck in this script.

Remember that when generating an image file from PHP, always include the appropriate content type for the server's response headers. In the case of PNG images, use `image/png`, as shown below:

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

Caching dynamic files

Generating 3-D images on the fly is fine and all, but if you get more than a few hits to the script in an hour, you may cause a big performance dip on your Web server. Let's walk through an example of how you might prevent this.

Imagine that you have a script that creates 3-D images of spheres. Accordingly, you name the script `spheres.php`. If it only created a sphere of one size and color, it wouldn't be very useful. Suppose you pass two parameters to the script every time you call it: one parameter, `radius`, will be the `r` value for creating the sphere object. The second parameter, `color`, will be a nine-digit RGB value for the color of the sphere. Here is an example of how you might call this script from a Web page:

```

```

Assuming you use a nice detail level of 5 or greater for generating the sphere, you'll probably want to avoid recreating the image every time a visitor comes to your Web page.

The approach you will take is to create a formatted filename for every image you create, which will include the two parameters used to generate the dynamic image -- for example, `sphere_20_255000000.png`.

Every time the `spheres.php` script is called, you will first look to see if you have a matching image file that satisfies the request. If so, open the file and pipe the contents to the output stream. If not, build the `Image_3D` object, save the new image file, then open the file and pipe the contents to the output stream. While the first visitor may have to wait a short period to see the image, future visitors will get the version that has been cached away.

And your system administrator will love you!

Getting your piece of the pie

You're finally at the point where you'll see a really neat example of `Image_3D` that could be used directly in your business and database applications.

`Image_3D` supports one more object type that hasn't been introduced yet: a pie chart. Let's jump right in and see what a slice of pie looks like in three dimensions. Each piece of the pie will be a different color, so rather than using white objects and colored lights, you'll use a white light and colored objects.

Start with your standard environment from [Listing 1](#), but replace the two light sources with a single white light (see [Listing 16](#)).

Listing 16. A blue 45-degree slice of a pie chart

```
$light = $world->createLight(0, 1000, 1000);
$light->setColor(new Image_3D_Color(255, 255, 255));

$pie = $world->createObject('pie', array('start' => 0,
                                       'end' => 45,
                                       'detail' => 20,
                                       'outside' => 150));
$pie->setColor(new Image_3D_Color(0, 0, 255));

$world->transform($world->createMatrix('Scale', array(1, 1, 10)));
$world->transform($world->createMatrix('Rotation', array(-60, 0, 0)));
```

Creating the pie object requires four entries in your `array` parameter passed to the

`createObject()` method. `start` and `end` refer to the beginning and ending degree measurements of the slice. These are measured from 0 degrees. If the pie was a clock face, 0 degrees lies at the 3 o'clock position, and angular measurement is made in a clockwise direction. `Detail` provides a level of smoothness for the object, and `outside` is the relative size of the circular pie.

You use two transformations on this object. The scale transformation is used to give each slice a feeling of depth using a Z-axis size of 10. The rotation applied to the X-axis tilts the pie into the familiar orientation seen in business charts (Figure 18).

Figure 18. A single pie object



As you add slices around the pie, you may find that they overlap each other in an unexpected fashion. This will be simple enough to fix by replacing the GD driver with ZBuffer.

Suppose you are using the Apache Derby database to store customer records for an e-commerce Web site. Each customer record will include the state for each mailing address. A quick SQL query would reveal how many customers live in each state, as shown in Listing 17.

Listing 17. Selecting a count of customer records from distinct states

```
SELECT COUNT( * ) AS customers, state
FROM customers
GROUP BY state
ORDER BY customers DESC
```

Designing a class interface

You can easily generate markup that converts the SQL result set into an HTML table. Let's add a fancy pie chart to accompany the table. For this example, start by designing a class interface for how you want your program to build the 3-D chart. Each slice will be defined as a percentage of the whole. You can specify the color of each slice, in case you want to associate those slices directly with data in the HTML table. Finally, an optional third parameter will allow for calling out specific slices by pulling them away from the main pie (`$slice->explode`); see Listing 18.

Listing 18. Using a pie chart class to build a 3-D image

```
<?php
require_once 'pie_chart_class.php';

$chart = new PieChart(400, 400, array(255,255,255));

$chart->addSlice(15, array(255,0,0), true);
$chart->addSlice(5, array(255,255,0), true);
$chart->addSlice(35, array(0,255,0));
$chart->addSlice(30, array(0,255,255));
$chart->addSlice(15, array(0,0,255));

$chart->render('pie.png');
?>
```

The percentages listed are sample data, only. What you've left out is how you would generate the results set from Derby and translate each count of customers into a percentage of all customers. But that is standard database interaction and somewhat of a tangent to your task at hand.

Constructing the class

Now that you have a sample of the class interface, fill in the gaps by creating the class. There's nothing particularly advanced about this class. The only detail worth noting is that as each slice is added to the pie, it is added in a clockwise direction. For any slices that have the `explode` property set to true, you will need to calculate the directions along the X- and Y-axes in which you will move the slice to pull it out of the pie (see Listing 19).

Listing 19. PHP class for generating 3-D pie charts

```
<?php
require_once 'Image/3D.php';

class PieChart {

    private $slices;
    private $width;
    private $height;
    private $bg;
    private $world;

    public function __construct($width, $height, $bg_list)
    {
        $this->width = $width;
        $this->height = $height;
        $this->bg = $bg_list;
    }

    public function addSlice ($percent, $color_list, $explode=false)
    {
        $this->slices[] = new PieChart_Slice($percent,
                                           $color_list, $explode);
    }
}
```

```

public function render ($filename)
{
    $radius = round((min($this->width,$this->height) * 0.85) / 2);

    $world = new Image_3D();
    $world->setColor(new Image_3D_Color($this->bg[0],
                                       $this->bg[1], $this->bg[2]));

    $light = $world->createLight(0, 1000, 1000);
    $light->setColor(new Image_3D_Color(255, 255, 255));

    $start = 0;
    foreach ($this->slices as $slice) {

        $end = $start + $slice->degrees;

        $options = array('start' => $start,
                        'end' => $end,
                        'detail' => 20,
                        'outside' => $radius);
        $pie = $world->createObject('pie', $options);
        $color = new Image_3D_Color($slice->rgb[0],
                                   $slice->rgb[1],
                                   $slice->rgb[2]);

        if ($slice->explode) {
            $mid = $end - (($end - $start) / 2);
            $dx = cos(deg2rad($mid)) * ($radius * 0.15);
            $dy = sin(deg2rad($mid)) * ($radius * 0.15);
            $pie->transform($world->createMatrix('Move',
                                                array($dx, $dy, 0)));
            $color->addLight($color, 0.4);
        }
        $pie->setColor($color);

        $start = $end;
    }

    $world->transform($world->createMatrix('Scale',
                                          array(1, 1, 10)));
    $world->transform($world->createMatrix('Rotation',
                                          array(-60, 0, 0)));

    $world->createRenderer('perspectively');
    $world->createDriver('zbuffer');
    $world->render($this->width, $this->height, $filename);
}
}

class PieChart_Slice {

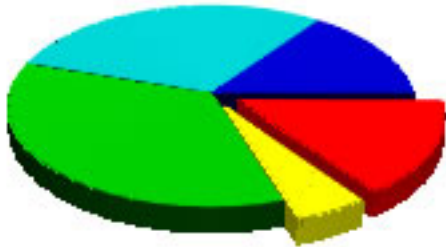
    public $percent;
    public $rgb;
    public $degrees;
    public $explode;

    public function __construct($percent, $color_list, $explode=false)
    {
        $this->percent = $percent;
        $this->rgb = $color_list;
        $this->explode = $explode;
        $this->degrees = 360 * ($percent / 100);
    }
}
?>

```

The resulting chart is displayed below:

Figure 19. A pie chart with called-out slices in red and yellow



Section 10. Summary

Image_3D is an easy PEAR package to set up and use in PHP for dynamically generating images. The class interface should be straightforward for anyone who has dabbled in PHP V5. You have investigated all of the object types, lightings, colors, and transformations included in the package. You also have looked at how to take simple command-line scripts and build more complex applications, along with a strategy for making them usable even across the Web. Finally, you designed a simple class that can manage the creation of pie charts your supervisor will drool over.

Not all of the examples are going to be immediately useful to you. Remember, however, that this package is still in alpha release. If you find some functionality missing or broken, consider submitting a request to the PEAR wishlist.

Downloads

Description	Name	Size	Download method
PHP 3-D code sample	os-php-3d.source.zip	779KB	HTTP

[Information about download methods](#)

Resources

Learn

- Read about and install [PEAR::Image_3D](#).
- Learn how to extend the pie graph class with "[Connecting PHP Applications to Apache Derby](#)."
- Learn more about SVG images and scripting in the "[Add interactivity to your SVG](#)."
- Brush up on the PHP V5 object/class syntax by reading "[Getting started with objects with PHP V5](#)."
- See this wiki to learn more about the [Cartesian coordinate system](#).
- Read "[Generate dynamic bitmap graphics with PHP and gd](#)" to learn how to generate dynamic bitmap images using PHP and the gd library.
- Read "[Create graphics the smart way with PHP](#)" to learn how to build an object-oriented graphics layer in PHP.
- Stay current with developerWorks [technical events and webcasts](#).
- Expand your PHP skills by visiting developerWorks [PHP project resources](#).
- Browse all of the [PHP content](#) on developerWorks.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download [Adobe's SVG Viewer plug-in](#).
- Use [PEAR::Benchmark](#) to measure the execution time of your PHP scripts.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Mike Brittain



Mike Brittain is the director of technology at ID Society, a full-service Internet marketing agency in New York City. He has been developing Web sites and applications for more than 10 years, focusing on open source languages and applications. When not at his computer, he can often be found on skis or a snowboard. He can be reached at mike@mikebrittain.com.