Pro .NET 2.0 Graphics Programming

Eric White

Apress®

Pro .NET 2.0 Graphics Programming

ISBN (pbk): 1-59059-445-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Mark Horner
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Marilyn Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Kinetic Publishing Services, LLC
Proofreader: Linda Seifert
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Interior Designer: Diana Van Winkle, Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 3

Pens and Brushes

When it comes to drawing graphics, *pens* and *brushes* are the basic tools of the trade. GDI+ has two classes that represent these two essential drawing tools. You use the `Pen` class to draw lines, curves, and outlines of shapes. You use the `Brush` class to fill shapes with colors and patterns. If you've read the first two chapters of this book, you've already seen the `Pen` and `Brush` classes used to demonstrate other concepts of GDI+.

These two classes are very flexible, giving you plenty of options that affect the way their operations are performed. By understanding these options, and the exact semantics of their methods, you'll be able to harness the full range and power they offer. This chapter covers the capabilities of the `Pen` and `Brush` classes, with many examples of their use.

First, we'll examine the `Pen` class. In particular, we'll look at pens with different widths, pens that draw dashed lines, pens that cap lines with arrowheads and other shapes, and the different ways you can join lines. We'll also look at the `Pens` class, which provides predefined pens for your use. Then we'll move on to look at the `Brush` class. We'll examine texture brushes, linear gradient brushes, and hatch brushes, as well as the predefined brushes in the `Brushes` class.

Next, you'll see how to use a brush to create a pen. This is a technique that allows you to use brush-type styles to perform pen-type operations, providing a great deal of extra flexibility in the types of effects you can achieve. Finally, I'll include a few words about how you can expect the instantiation of `Pen` and `Brush` objects to affect the performance of your applications.

Drawing with the Pen Object

You use the GDI+ `Pen` class to create custom pens. You can specify the `Color` and `Width` properties of the pen at the time you construct the `Pen` object. In fact, the `Pen` class is furnished with a number of other properties, such as the `StartCap` and `EndCap` properties, which allow you to add shapes to the start or end of lines, and the `DashStyle` property, which allows you to draw dashed and dotted lines. You can adjust the behavior of the `Pen` object any time after its instantiation by changing the values of these properties. We'll explore all of these capabilities in this section.

The Pen Class

The simplest pen is a solid pen with a width of 1 pixel. In this case, you are just required to specify the pen's color. So, to instantiate such a pen, you would use the class constructor like this:

```
Pen p = new Pen(Color.Black);
```

The following example creates such a pen, then uses it to draw a line from the pixel (0, 0) to the pixel (100, 100), and then calls the `Pen` object's `Dispose` method to release the resources used by the `Pen` object:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black);
    g.DrawLine(p, 0, 0, 100, 100);
    p.Dispose();
}
```

Figure 3-1 shows the resulting line.

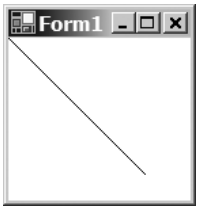

Figure 3-1. *A diagonal line*

■ **Note** The `FillRectangle` method here simply paints the background of the window (the client rectangle) white; as you can see, it uses a brush to do that. I've just included this operation in order to make the rendered examples look a little cleaner. We'll discuss brushes in detail later in this chapter.

In fact, there are four constructors for the pen class: two constructors allow you to specify a `Brush` object instead of a color, and two constructors give you the option of specifying the pen's width. The four constructors are as follows:

```
// Specified color, default width (1px)
Pen p1 = new Pen(myColor);
// Specified color, specified width (px)
Pen p2 = new Pen(myColor, myWidth);
// Specified brush, default width (1px)
Pen p3 = new Pen(myBrush);
// Specified brush, specified width (px)
Pen p4 = new Pen(myBrush, myWidth);
```

In these lines, *myColor* represents a `Color` object (specified in any of the ways discussed in Chapter 2), *myWidth* is a value of `Float` type, and *myBrush* is a `Brush` object. You'll see examples

of all these constructors as we progress through this chapter. We'll consider using the constructors to combine pen and brush capabilities (by specifying a `Brush` object) toward the end of the chapter, after the discussion of brushes.

The GDI+ `Pen` class has a number of properties, offering you a large amount of control over how your `Pen` object draws graphic elements. Next, we'll take a look at some of the most commonly used properties of the `Pen` class.

The Width of the Pen

The first and most obvious thing to get right is the way you control a pen's width. The *width* of a pen in GDI+ means much the same as it does in real life: thin pens draw fine lines, and thick pens draw heavy lines. But there are a couple of additional subtleties that you need to consider.

To begin, let's draw a rectangle with a pen width of 1 pixel:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black);
    g.DrawRectangle(p, 3, 3, 8, 7);
    p.Dispose();
}
```

Figure 3-2 shows the resulting rectangle. The top-left corner of this rectangle is the point with coordinates (3, 3), and it is 8 pixels long and 7 pixels deep.

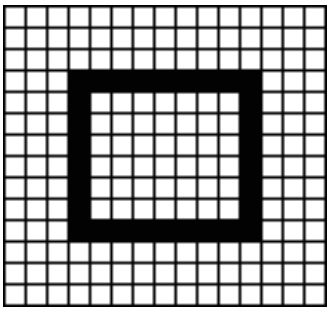

Figure 3-2. *A 1-pixel-wide pen*

■ **Note** In the illustrations here, the grid represents (part of) the client rectangle, and the top-left cell in the grid represents the pixel (0, 0).

As this example shows, the constructor that doesn't take a pen width as an argument creates a pen with a width of 1 pixel. Of course, you can create a pen with a width greater than 1 pixel. To do that, just use another one of the `Pen` class's constructors. Let's re-create the rectangle, this time using a pen width of 3 pixels:

```
Graphics g = e.Graphics;
Pen p = new Pen(Color.Black, 3);
p.Alignment = PenAlignment.Center;
g.DrawRectangle(p, 3, 3, 8, 7);
p.Dispose();
```

Figure 3-3 shows the result of this code. Note two things about this rectangle. First, unsurprisingly, the sides of the rectangle are all 3 pixels wide. Second, although the top-left corner of the rectangle is defined to by the pixel (3, 3), you can see that the *apparent* corner of the rectangle is the pixel (2, 2). That's because the pen's width is greater than 1 pixel—it's 3 pixels—and so in this case, all the pixels adjacent to the pixel (3, 3) also are drawn black.

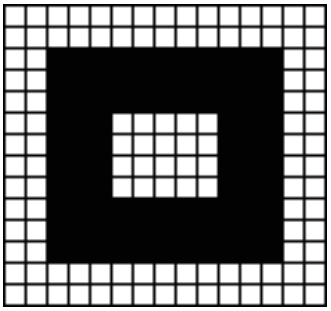

Figure 3-3. *A 3-pixel-wide pen*

This example uses the `Pen` object's `Alignment` property to control exactly how the pen should present its width on the drawing surface. It sets this property to `Center`, which dictates that the shape should be drawn as if the *center* of the pen described the shape. The best way to appreciate this is to take a look at what happens when you change the value of the `Alignment` property to `Inset`:

```
Graphics g = e.Graphics;
Pen p = new Pen(Color.Black, 3);
p.Alignment = PenAlignment.Inset;
g.DrawRectangle(p, 3, 3, 8, 7);
p.Dispose();
```

Figure 3-4 shows the result. This time, the top-left corner of the rectangle, (3, 3), is the outermost, and the shape has been drawn as if the *outside* of the pen described the shape (that is, the pen ran along the *inside* of the shape).

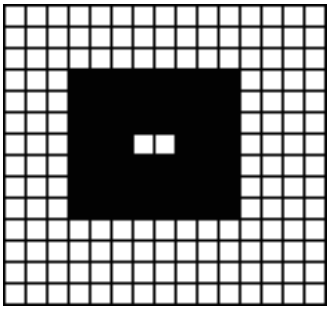

Figure 3-4. *Line drawn with Inset alignment*

The `PenAlignment` enumeration used here is part of the `System.Drawing.Drawing2D` namespace. This enumeration has three other values: `Outset`, `Left`, and `Right`. In my tests, I've found them to work in much the same way as `Center`, although Microsoft's documentation suggests that each has its own distinct effect (with `Outset` doing the opposite of `Inset`, and `Left` and `Right` creating a sort of left and right shadow, respectively).

Finally, note that if you don't set the `Alignment` property, it defaults to `Center`.

Dashed Lines

For some applications, you might want to draw dashed lines. For example, if you were creating a drawing program, you might want to use dashed lines to indicate that an object within the drawing space is selected. You have several different ways to create dashed lines and shapes. The simplest way is to set the `Pen` object's `DashStyle` property to one of the values provided by the `DashStyle` enumeration, like this:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Pen p = new Pen(Color.Black, 1);

    p.DashStyle = DashStyle.Dash;
    g.DrawLine(p, 3, 3, 100, 3);
    p.Dispose();
}
```

Figure 3-5 shows the dashed line produced by this code.

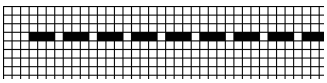

Figure 3-5. *A dashed line*

The DashStyle enumeration is provided by the System.Drawing.Drawing2D namespace, which offers five predefined styles: Solid, Dash, DashDot, DashDotDot, and Dot. Here is an example that uses all of these predefined styles:

```
p.DashStyle = DashStyle.Solid;
g.DrawLine(p, 3, 2, 100, 2);

p.DashStyle = DashStyle.Dash;
g.DrawLine(p, 3, 6, 100, 6);

p.DashStyle = DashStyle.DashDot;
g.DrawLine(p, 3, 10, 100, 10);

p.DashStyle = DashStyle.DashDotDot;
g.DrawLine(p, 3, 14, 100, 14);

p.DashStyle = DashStyle.Dot;
g.DrawLine(p, 3, 18, 100, 18);
p.Dispose();
```

Figure 3-6 shows the line produced by each style.

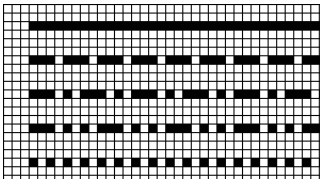

Figure 3-6. *Various styles of dashed and dotted lines*

Custom Dashed Lines

The predefined dash styles can be useful, but if you've tried viewing the result of the preceding code fragment at high resolution, you would probably find it difficult to tell them apart. For this reason, you might prefer to create your own customized dash styles. You can do this by using an array of integers that describe the pixel length of the dashes and the spaces between them. Here's an example:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Pen p = new Pen(Color.Black, 1);

    // The following line creates the custom dash pattern:
    float[] f = {15, 5, 10, 5};
```

```
    p.DashPattern = f;
    g.DrawRectangle(p, 10, 10, 80, 100);
    p.Dispose();
}
```

In this example, the dash pattern is specified by the array `{15, 5, 10, 5}`. It uses a four-element array, so the dash pattern will have a cycle of four: a 15-pixel dash, then a 5-pixel space, then a 10-pixel dash, and then a 5-pixel space. The pattern begins in the bottom-left corner, as shown in Figure 3-7.

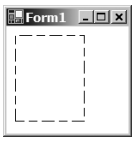

Figure 3-7. *A custom dashed line*

When you assign a value to the `Pen` object's `DashPattern` property, its `DashStyle` property is automatically set to the value `DashStyle.Custom`.

Wider Dashed Lines

If the pen's width is set to a value greater than 1 pixel, the actual length of each dash and space is calculated by multiplying the values specified in the array by the width of the pen. For example, let's try the previous custom dashed line with a pen width of 2 pixels:

```
Pen p = new Pen(Color.Black, 2);

// The following line creates the custom dash pattern:
float[] f = {15, 5, 10, 5};

p.DashPattern = f;
g.DrawRectangle(p, 10, 10, 80, 100);
p.Dispose();
```

Now the pattern array `{15, 5, 10, 5}` produces alternating dashes of length 30 and 20, separated by spaces 10 pixels long. You can see this by comparing the result shown in Figure 3-8 with the one shown in Figure 3-7.

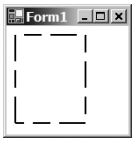

Figure 3-8. *Wider custom dashed lines*

This behavior means that the dashes and spaces in your lines are relative to the width of the pen. If you were creating a drawing program that allowed users to zoom in and out of their

drawings, you would want this to happen. If you didn't have this behavior, as the user zoomed in, your nicely dashed line would not appear dashed in the same way.

Arrowheads and Other Line Caps

You can specify how GDI+ decorates the beginning and end of a line by using *line caps*. In practice, this means assigning values to the `Pen` object's `StartCap` and `EndCap` properties. The following code draws a line with a `StartCap` of `LineCap.Round` and an `EndCap` of `LineCap.ArrowAnchor`:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black, 10);
    p.StartCap = LineCap.Round;
    p.EndCap = LineCap.ArrowAnchor;
    g.DrawLine(p, 30, 30, 80, 30);
    p.Dispose();
}
```

Note that you get the best results when you use a pen whose width is greater than 1 pixel. This example uses a 10-pixel width pen, which creates a very effective rounded end, as shown here:

Table 3-1 shows the available line caps and their appearances.

Table 3-1. *Available Line Caps*

StartCap and EndCap Values	Result
`ArrowAnchor`	
`DiamondAnchor`	
`Flat`	
`Round`	
`RoundAnchor`	
`Square`	
`SquareAnchor`	
`Triangle`	

To create the capped lines, you use the `LineCap` values, as in the following example:

```
p.StartCap = LineCap.DiamondAnchor;
p.EndCap = LineCap.DiamondAnchor;
```

You can see that all of the anchor line caps (`ArrowAnchor`, `DiamondAnchor`, `RoundAnchor`, and `SquareAnchor`) have a very distinctive appearance, in that they add a decorative anchor to the end of the line. The others merely "trim" the line end in some way. All of these values belong to the `LineCap` enumeration, which is part of the `System.Drawing.Drawing2D` namespace.

If you were creating a diagramming program for a particular style of object-oriented analysis, you might want different end caps than those available by default. You can create custom line caps by instantiating the `CustomLineCap` class and setting the `CustomStartCap` and `CustomEndCap` properties of the `Pen` class. To create a custom line cap, you create a `GraphicsPath` object with the desired path of your line cap, then instantiate the `CustomLineCap` class using the `GraphicsPath` object (which is covered in Chapter 6). You can set various options regarding scaling, how lines join within the line cap, and so on. Building custom line caps is not necessary for most custom controls, so we will not delve into the subject too deeply, but it's useful to know that you can make your own line caps if you need them for your application.

Joined Lines

Suppose you were building an application that allowed users to draw lines representing electrical conduit. A conduit run is composed of straight lines and bends. Your GDI+ code would need to draw a number of straight lines that were joined where they met.

Whenever you perform a single operation that involves the rendering of joined lines, you can set the style in which the lines are joined. To do this, you use values from the `System.Drawing.Drawing2D` namespace's `LineJoin` enumeration, which has four values: `Miter` (the default), `Bevel`, `MiterClipped`, and `Round`. `MiterClipped` is not a common choice for writing custom controls, and we've been using `Miter` (by default) in the examples in this chapter. So here, let's just briefly demonstrate the other two.

The following code sets the `Pen` object's `LineJoin` property to the value `LineJoin.Bevel`, and then uses it to draw a rectangle:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black, 10);
    p.LineJoin = LineJoin.Bevel;
    e.Graphics.DrawRectangle(p, 20, 20, 60, 60);
    p.Dispose();
}
```

Unsurprisingly, the resulting rectangle has beveled corners, like this:

Alternatively, you can set the `LineJoin` property to the value `LineJoin.Round`:

```
Pen p = new Pen(Color.Black, 10);
p.LineJoin = LineJoin.Round;
e.Graphics.DrawRectangle(p, 20, 20, 60, 60);
p.Dispose();
```

Then the rectangle has rounded corners, like this:

Once again, you can see that this feature produces better effects when the pen width is relatively high (but not so high that you can't make out the shape being drawn!). These images are produced with a 10-pixel pen width.

Predefined Pens

In Chapter 2, you learned that the `Color` structure makes available 141 predefined colors. For convenience, GDI+ provides a predefined set of 141 colored pens. Each of these pens has a width of 1 pixel, and there is a prebuilt pen for every color defined in the `Color` structure.

To access these pens, use the `Pens` class, like this:

```
private void Form1_Paint (object sender,
                          System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawRectangle(Pens.MistyRose, 20, 20, 40, 40);
}
```

The result of this code fragment is a rather pinkish-looking rectangle whose edges have a width of 1 pixel.

As a general rule, if your graphics allow you to use the pens from the `Pens` class, then you should do so. Because they are already built and ready to use, they offer some performance advantages. We'll return briefly to the subject of performance at the end of this chapter.

Note that when you create `Pen` objects using the `Pens` class, you don't create the pen explicitly, and therefore it's wrong to try to call the `Dispose` method of the resulting `Pen` object. If you attempted to do that, as in the following code fragment, then an exception would result.

```
Pen blackPen = Pens.Black;
g.DrawRectangle(blackPen, 20, 20, 40, 40);
blackPen.Dispose();                  // generates an exception
```

In general, the rule to follow is this: if you created it, then you dispose of it. If you did not create it, then leave it alone. Appendix C discusses the dynamics of classes that implement the `IDisposable` interface.

Filling with the Brush Object

In GDI+, you use brushes to fill shapes with colors, patterns, and images. The GDI+ `Brush` class itself is an `abstract` class, and so you cannot instantiate it directly. Rather, the GDI+ API provides the five classes shown in Table 3-2, which extend the `Brush` class and provide concrete implementations.

Table 3-2. *GDI+ Brushes*

Class	Description
`SolidBrush`	Fills a shape with solid colors
`TextureBrush`	Fills the shape with a raster-based image (BMP, JPG, and so on)
`LinearGradientBrush`	Fills the shape with a color gradient that changes evenly from one color to another, in a specified direction and between two specified (parallel) boundary lines
`PathGradientBrush`	Fills the shape with a gradient that changes evenly as you move from the boundary of the shape defined by the path to the center of the shape
`HatchBrush`	Fills the shape with various patterns

In this section, we'll look at some of the features of these brushes in more detail. It seems sensible to start with the simplest of these: the `SolidBrush`.

The SolidBrush Class

The following code creates a solid brush, and then uses it to fill a rectangle:

```
SolidBrush b = new SolidBrush(Color.Crimson);
g.FillRectangle(b, 20, 20, 40, 40);
b.Dispose();
```

This renders a rectangle whose interior is filled with crimson, as shown in Figure 3-9. A `SolidBrush` has only one constructor, and after you instantiate it, no options can be set.

Figure 3-9. *A filled rectangle*

The TextureBrush Class

A `TextureBrush` object fills a shape with a raster-based image. It uses an image that comes from an image file such as a `.bmp`, `.jpg`, or `.png` file. You can obtain such an image from a file by using the `Bitmap` class, which is a subclass of the `Image` class (we'll discuss the `Image` class in detail in Chapter 5). To do that, use code like this:

```
Bitmap bmp;
bmp = new Bitmap("alphabet.gif");
```

In order to demonstrate a few simple examples that involve the `TextureBrush` class, we need a suitable image—one that has an obvious boundary and looks clearly different if we flip (or reflect) it horizontally or vertically. So, in these examples, we'll use the image `alphabet.gif`:

■ **Note** The image file `alphabet.gif` is included as part of the sample code for this book and available for download from the Downloads section of `www.apress.com`. For the example to work, you must place the image file in the same directory as the executable. If you're compiling with debugging information turned on, then Visual Studio .NET places the executable in the /bin/debug subdirectory of your project's directory, so you must place the image there.

For a first example here, let's just draw a simple rectangle, filled using this pattern:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("alphabet.gif");
    TextureBrush tb = new TextureBrush(bmp);

    g.FillRectangle(tb, 20, 20, 200, 70);
    bmp.Dispose();
    tb.Dispose();
}
```

Here, we first specify the name and location of our raster-based image as an argument to the `Bitmap` class constructor and specify the resulting `Bitmap` object in the constructor for the `TextureBrush` object. The result is a `TextureBrush` object that fills shapes using a tile effect based on the pattern in the image file.

Once you have your `TextureBrush` object, you can use it to create some filled shapes. In this example, we draw a filled rectangle, as shown in Figure 3-10.

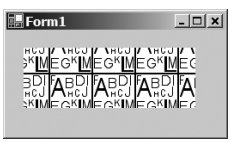

Figure 3-10. *Rectangle filled with a TextureBrush*

Notice that the top-left pixel of the rectangle painted here is (20, 20). You might be a little surprised to see only part of the `alphabet.gif` image nestling into the top-left corner of the rectangle. This is because the `TextureBrush` object fills the rectangle by using the image as a tile, arranging copies of the tile (image) on the drawing surface. Of course, it needs a starting point for the tiling pattern, and unless you tell it otherwise, it starts the tiling pattern from pixel (0, 0). With that in place, the top-left corner of the rectangle falls in the middle of a tile, and so you see only a part of the image in that corner.

■ **Tip** The `TextureBrush` behavior of tiling from the pixel (0, 0) is the default. You can override this default by setting a transformation on the `TextureBrush` object. See Chapter 8 for details about transformations.

Overlapping Shapes

The way that the `TextureBrush` creates the fill allows you to draw a lot of shapes using the same brush, and if these drawing operations overlap, the bitmaps that make up the drawing operations will be aligned. To see how this works, let's add more overlapping shapes to the previous example. Modify the `Paint` event to draw another overlapping rectangle, as follows:

```
Graphics g = e.Graphics;
Bitmap bmp = new Bitmap("alphabet.gif");
TextureBrush tb = new TextureBrush(bmp);

g.FillRectangle(tb, 20, 20, 200, 70);
g.FillRectangle(tb, 45, 45, 70, 150);
bmp.Dispose();
tb.Dispose();
```

Figure 3-11 shows how the overlapping shapes appear.

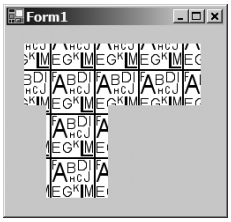

Figure 3-11. *Multiple fills using a TextureBrush*

Selection from an Image As a Tile

You might have a large bitmap and want to use only a portion of it for your brush. The `TextureBrush` constructor is overloaded to allow you to select a portion of the image, which is then used as the tile when the `TextureBrush` fills. To demonstrate this, let's adjust the previous example to use the overloaded constructor, as follows:

```
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("alphabet.gif");
    TextureBrush tb = new TextureBrush(bmp, new Rectangle(0, 0, 25, 25));

    g.FillRectangle(tb, 20, 20, 200, 70);
    g.FillRectangle(tb, 45, 45, 70, 150);
    bmp.Dispose();
    tb.Dispose();
```

As shown in Figure 3-12, this example uses just a 25-pixel square taken from the top-left corner of `alphabet.gif` to tile the two rectangles.

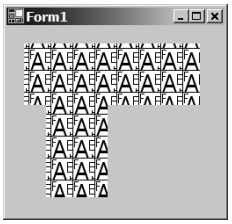

Figure 3-12. *Using a selection from an image*

Different Tiling Effects

You've seen the basic tiling effect that the `TextureBrush` achieves. You can use the `TextureBrush` object's `WrapMode` property to change the way it uses the tiles. Once again, GDI+ provides an enumeration, `WrapMode`, whose values can be used in conjunction with the `TextureBrush.WrapMode` property to produce different tiling effects, as you'll see in the following examples.

The `WrapMode` enumeration is part of the `System.Drawing.Drawing2D` namespace, and it contains five values: `Tile`, `Clamp`, `TileFlipX`, `TileFlipY`, and `TileFlipXY`. We'll briefly compare them all here by using them, one at a time, to fill the entire client rectangle.

First, let's see the effect of setting the `WrapMode` property to `WrapMode.Tile`. This is the default, and it tiles the bitmap over the entire drawing surface.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("alphabet.gif");
    TextureBrush tb = new TextureBrush(bmp);

    tb.WrapMode = WrapMode.Tile;
    g.FillRectangle(tb, this.ClientRectangle);
    bmp.Dispose();
    tb.Dispose();
}
```

As you can see in Figure 3-13, there isn't much new to note here, except that we're filling the *client rectangle*, instead of filling a smaller rectangle with specified corner pixels.

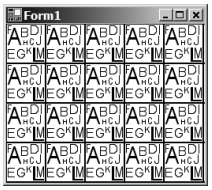

Figure 3-13. *Using WrapMode.Tile*

If you set the `WrapMode` property to `WrapMode.Clamp`, the texture brush doesn't tile the shape at all. Instead, it clamps the bitmap to the origin of the drawing surface.

```
tb.WrapMode = WrapMode.Clamp;
g.FillRectangle(tb, this.ClientRectangle);
```

Figure 3-14 shows the results.

Figure 3-14. *Using WrapMode.Clamp*

If a clamped brush is used to fill a rectangle that is larger than the brush, any portion of the rectangle outside the image will not be drawn.

If you set the `WrapMode` property to `WrapMode.TileFlipX`, the texture brush tiles the bitmap over the drawing surface, flipping every other horizontal bitmap horizontally.

```
tb.WrapMode = WrapMode.TileFlipX;
g.FillRectangle(tb, this.ClientRectangle);
```

You can see this effect in Figure 3-15.

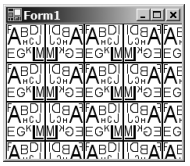

Figure 3-15. *Using WrapMode.TileFlipX*

A similar effect is gained when you set the `WrapMode` property to `WrapMode.TileFlipY`.

```
tb.WrapMode = WrapMode.TileFlipY;
g.FillRectangle(tb, this.ClientRectangle);
```

As you can see in Figure 3-16, this time, the texture brush flips every other *vertical* bitmap *vertically*.

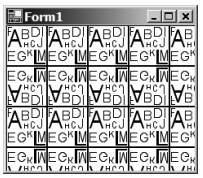

Figure 3-16. *Using WrapMode.TileFlipY*

Finally, the effect of setting the `WrapMode` property to `WrapMode.TileFlipXY` gives an amalgam of the previous two effects, flipping the bitmap in both the vertical and horizontal directions.

```
tb.WrapMode = WrapMode.TileFlipXY;
g.FillRectangle(tb, this.ClientRectangle);
```

Figure 3-17 shows the result.

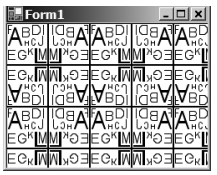

Figure 3-17. *Using WrapMode.TileFlipXY*

Image Attribute Specifications with TextureBrush

Before we move on to study the other `Brush` classes, it's worth mentioning that GDI+ allows you to affect how an image is drawn by using the `ImageAttributes` class to specify certain image attributes. We'll look at these techniques in more detail in Chapter 5, but I mention it now because the `TextureBrush` can take advantage of these techniques.

GDI+ allows you to pass an `ImageAttributes` object to the `TextureBrush` constructor at the time you create the `TextureBrush` object. The `ImageAttributes` object can contain attribute values that affect the way the brush draws the image. To demonstrate this, here's a quick example that uses an `ImageAttributes` object to manipulate the version of `alphabet.gif` that is painted on the screen:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Bitmap bmp = new Bitmap("alphabet.gif");
    Graphics g = e.Graphics;
    // Use a color matrix to change the color properties of the image
    float[][] matrixItems = {
                        new float[] {0.2f, 0, 0, 0, 0},
                        new float[] {0, 0.8f, 0, 0, 0},
                        new float[] {0, 0, 1, 0, 0},
                        new float[] {0, 0, 0, 1, 0},
                        new float[] {0, 0, 0, 0, 1}};
    ColorMatrix colorMatrix = new ColorMatrix(matrixItems);
```

```
    // Create an ImageAttributes object and set its color matrix
    ImageAttributes imageAtt = new ImageAttributes();
    imageAtt.SetColorMatrix(
                        colorMatrix,
                        ColorMatrixFlag.Default,
                        ColorAdjustType.Bitmap);

    // Create a TextureBrush object using the alphabet.gif image,
    // adjusted by the attributes in the ImageAttributes object
    TextureBrush tb = new TextureBrush(
                bmp,
                new Rectangle(0, 0, bmp.Width, bmp.Height),
                imageAtt);
    tb.WrapMode = WrapMode.Tile;
    g.FillRectangle(tb, this.ClientRectangle);
    bmp.Dispose();
    tb.Dispose();
}
```

The effect of all this is that the `TextureBrush` fills the client rectangle with a tile based on the image `alphabet.gif`, but whose background color is blue, not white. Unfortunately, the color doesn't come out too well in grayscale, but Figure 3-18 gives you an idea of what to expect. This example makes use of objects and enumerations that belong to the `System.Drawing.Imaging` namespace. You'll learn more about color matrices and the `ImageAttributes` object in Chapter 5.

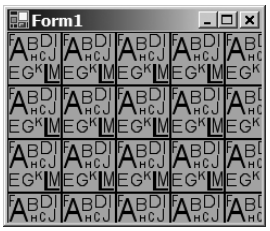

Figure 3-18. *A TextureBrush that uses ImageAttributes*

TILES AND ANTI-ALIASING

Texture brushes also behave in accordance with the coordinate system and anti-aliasing rules discussed in Chapter 2. You may recall that the coordinate system model of GDI+ is such that coordinates refer to gridlines that run horizontally and vertically through the center of the pixels. What would it mean to draw an image at a location halfway through a pixel? If anti-aliasing is turned on, and you create a texture brush and then draw it on integral coordinates, the edges of the brush's bitmap will be anti-aliased. However, it is only the *edges* that are anti-aliased. The internal portion of the bitmap is not affected by the anti-aliasing option.

That said, drawing images with anti-aliasing is a dubious activity at best, and you will almost never want to do it. GDI+ will let you attempt it, but the results will probably not be what you want. It is far better to explicitly avoid doing anti-aliasing while drawing an image.

The LinearGradientBrush Class

A `LinearGradientBrush` object fills shapes with a linear *color gradient*. In its simplest terms, a color gradient consists of a gradual color change between two specified colors as you traverse a straight-line path that tends toward a specified angle. We'll demonstrate this with a couple of examples in a moment.

In fact, the `LinearGradientBrush` class encapsulates two-color gradients and multicolor gradients, as you'll see in this section. For a simple, *two-color linear gradient* brush, you can describe the gradient line either with a pair of points or with a rectangle and an angle. The `LinearGradientBrush` constructor is overloaded to take in these different possibilities, and you'll see them here.

A Gradient Described by Two Points

First, let's instantiate a `LinearGradientBrush` object by providing two points (a starting point and an ending point) and two colors (a starting color and an ending color):

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;

    LinearGradientBrush lgb = new LinearGradientBrush(
                                        new Point(0, 0),
                                        new Point(50, 10),
                                        Color.White,
                                        Color.Black);
    g.FillRectangle(lgb, this.ClientRectangle);
    lgb.Dispose();
}
```

Figure 3-19 shows what you see when you run this application. Let's consider how this works. First, note that the two points specified describe the gradient line. Here, it starts at the pixel (0, 0) and ends at the point (50, 10). Of course, you can't *see* the gradient line in the figure, but you can imagine it. Second, note that at the start of the gradient line (the top-left corner of the client rectangle), the brush has painted the starting color, `White`. At the end of the line, the brush has painted the ending color, `Black`. In between, the color changes gradually from white to black as you look along this imaginary line.

Figure 3-19. *Using two points to describe a gradient with LinearGradientBrush*

Finally, note that this pattern is repeated in two directions—not horizontally and vertically, but in directions parallel and perpendicular to the direction of the gradient line. The effect is one of stripes that run perpendicular to the gradient line.

A Gradient Described by a Rectangle

Another way to define the gradient is to specify a rectangle and an angle in degrees. Here's an example:

```
LinearGradientBrush lgb = new LinearGradientBrush(
                                    new Rectangle(0, 0, 50, 50),
                                    Color.White,
                                    Color.Black,
                                    75f);
g.FillRectangle(lgb, this.ClientRectangle);
lgb.Dispose();
```

Figure 3-20 shows the gradient produced by this example. In this case, the first color is used in the upper-left corner of the rectangle, and the second color is used in the lower-right corner of the rectangle. In between, the color changes as per the gradient specified by the fourth argument, which represents the angle that the gradient line tends to the horizontal.

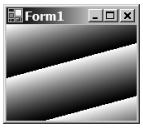

Figure 3-20. *Using a rectangle to describe a gradient with LinearGradientBrush*

Note that the gradient line is specified by the angle in the fourth argument, *not* by the positions of the two opposite corners of the rectangle! Note also that the angle specified represents the angle that the gradient line makes with the horizontal, in degrees. If you were to specify an angle of zero, the gradient line would be horizontal, and the stripes would appear to be vertical, as shown in Figure 3-21.

Figure 3-21. *A LinearGradientBrush with an angle of zero*

Multicolor Gradients and Other LinearGradientBrush Options

At the beginning of the section, I mentioned that a `LinearGradientBrush` object can create both two-color and multicolor gradients. A *multicolor gradient* is like a sequence of top-to-tail two-color transitions, which pass through certain specified colors at certain points along the gradient line. You describe this sequence by specifying colors in the sequence and indicating how far along the gradient line you want each color to occur. For example, you might define

a multicolor gradient to create a "chrome" look, like the effect you see when you look at a shiny bathroom faucet.

If you want to try a multicolor gradient fill, take a look at the `LinearGradientBrush` class's `SetInterpolationColors` method. It expects an array of the colors in the gradient, an array of numbers representing the positions, and an integer count of the total number of set colors in the sequence.

You can set other options when you're using gradient brushes. For example, you can control how one color transitions to another color. You can also enable and disable gamma correction. This is a complicated subject, but the essence of it is that you define a gradient between two colors, and with gamma correction, the gradient has the appearance of equal brightness at all points along the gradient.

These effects are quite advanced, and I won't go into detail about them here. With the increase in processing power, and the increase in users' expectations for a slick, fancy look, you can use gradients to good effect when building custom controls. But keep in mind that these effects can be overused. In general, you should stick to the norms of the Windows user interface. The Windows XP look contains some good examples of gradients that are subtly and tastefully implemented.

The PathGradientBrush Class

The `PathGradientBrush` object also provides a brush that fills a shape with a color gradient. In this case, you specify a path and two colors. The resulting color gradient begins at all points on the specified path and ends at a point that is deemed to be the center of the shape. This type of gradient reminds me of coloring maps in grade school, where we would color the outlines of the countries with a darker color than the center of the countries. You'll see an example in a moment.

In order to use the `PathGradientBrush` class effectively, you use it in conjunction with the `GraphicsPath` class. So, let's take a quick look at the `GraphicsPath` class, which will be covered in detail in Chapter 6, before we try a `PathGradientBrush` example.

You use the `GraphicsPath` class to describe a shape composed of multiple line segments. We say that a `GraphicsPath` object is *closed* if the path's ending point is the same as its starting point; otherwise, we say that the `GraphicsPath` object is *open*. You can convert an open `GraphicsPath` object to a closed one by connecting the last point to the first point. You can do this explicitly by calling the object's `CloseFigure` method. When you put together a `GraphicsPath` object, you first construct it, and then add segments to it. You can add different types of segments to the path. The following example creates a `GraphicsPath` object, adds three line segments to create an open path, then closes the path, and then draws the path with a pen.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsPath gp = new GraphicsPath();

    gp.AddLine(10, 10, 110, 15);
    gp.AddLine(110, 15, 100, 96);
    gp.AddLine(100, 96, 15, 110);
    gp.CloseFigure();
```

```
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawPath(Pens.Black, gp);
    gp.Dispose();
}
```

The result looks like Figure 3-22.

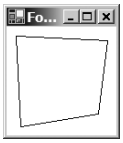

Figure 3-22. *Line drawn with a GraphicsPath*

■ **Note** In the `GraphicsPath` example, the second `AddLine` method call—`gp.AddLine(110, 15, 100, 96);`— is not technically necessary. If you add two segments that are not joined, the `GraphicsPath` class will automatically add a segment to join them. In the example, for clarity, I showed how to add all segments explicitly, except for the last one, which is added by the call to `CloseFigure`.

This is not the whole story as far as the `GraphicsPath` class is concerned, but it's sufficient information to enable you to take a meaningful look at the `PathGradientBrush` class, as demonstrated in the following example:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsPath gp = new GraphicsPath();

    gp.AddLine(10, 10, 110, 15);
    gp.AddLine(110, 15, 100, 96);
    gp.AddLine(100, 96, 15, 110);
    gp.CloseFigure();

    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.SmoothingMode = SmoothingMode.AntiAlias;

    PathGradientBrush pgb = new PathGradientBrush(gp);
    pgb.CenterColor = Color.White;
    pgb.SurroundColors = new Color[]
    {
        Color.Blue
    };
    g.FillPath(pgb, gp);
```

```
    g.DrawPath(Pens.Black, gp);
    pgb.Dispose();
    gp.Dispose();
}
```

In this example, we pass the `GraphicsPath` object to the constructor of the `PathGradientBrush` class. Then we need to set the `PathGradientBrush.CenterColor` property. Here, we've set it to white (using the `Color` class). The next statement is interesting:

```
pgb.SurroundColors = new Color[]
{
    Color.Blue
};
```

We pass an array of colors for the `SurroundColors` property. In this case, we created an array with only one color in it, so when you run the application, the `PathGradientBrush` paints blue at the edge of the path, transitioning to white at the center. This code paints the shape as shown in Figure 3-23.

Figure 3-23. *A PathGradientBrush*

To see the full effect of the color array in the previous code fragment, let's modify our example to set the `SurroundColors` property to an array that contains several colors, like this:

```
pgb.SurroundColors = new Color[]
{
    Color.Blue,
    Color.Red,
    Color.Green,
    Color.Yellow
};
```

You'll need to try this out for yourself to appreciate the effect, because it doesn't show up too well in grayscale. You'll see a figure shaped similar to the one shown in Figure 3-23, which is white in its center but whose colors vary around the edge of the graphic. The four corners (from top left, clockwise) are blue, red, green, and yellow.

■ **Caution** You cannot set the `SurroundColors` property to an array that contains more colors than there are segments in the `GraphicsPath` object. If you do so, you will get an exception.

The HatchBrush Class

The `HatchBrush` class provides a brush that can fill a shape with a variety of patterns. Hatch brushes are particularly useful when printing to a printer that is capable of printing only in black and white. For example, if you're drawing a bar chart or a map and filling adjacent regions with different colors, but the printer can print only in grayscale, adjacent regions might be difficult to differentiate. Instead, you can use hatch brushes, filling one region with a crosshatch, another region with diagonal lines, and so on, so that they are easily distinguishable.

The patterns for hatch brushes are always two-color patterns, based on a 16-pixel-by-16-pixel repeat. This is a limitation within the .NET Framework of the hatch brush functionality. You set the hatch style via the `HatchStyle` enumeration, which belongs to the `System.Drawing.Drawing2D` namespace.

The following code sets the `HatchStyle` to the `Cross` pattern, and then fills the client area of the window:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;

    HatchBrush hb = new HatchBrush(
                    HatchStyle.Cross,
                    Color.White,
                    Color.Black);
    g.FillRectangle(hb, this.ClientRectangle);
    hb.Dispose();
}
```

When run, the example looks like Figure 3-24.

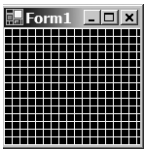

Figure 3-24. *A HatchBrush with HatchStyle.Cross*

You can set three options when creating a hatch brush: the foreground color, the background color, and the hatch style. There are 56 different hatch styles—too many to demonstrate here! For more information, see the `System.Drawing.Drawing2D.HatchStyle` enumeration in the .NET Framework documentation.

Predefined Brushes

GDI+ includes a `Brushes` class, which (like the `Pens` class discussed earlier) provides a set of predefined `SolidBrush` objects. There are 141 predefined brushes—one for each predefined color. In fact, you've already used this class several times in this chapter to fill the background of the client area of your window with white, like this:

```
g.FillRectangle(Brushes.White, this.ClientRectangle);
```

Each time you request a brush from the `Brushes` class, you do not *explicitly* create the brush, so you should not call the `Dispose` method on it. It seems that the intent of the Framework designers is that the Framework does not dispose of solid brushes acquired from the `Brushes` class until the application terminates. These brushes will typically be used often within an application, and the resource cost is minimal, so eliminating the creation/disposal cycle as far as possible provides better performance.

Creating a Pen from a Brush

Now that we've discussed the key features and differences of both pens and brushes, let's take a look at how you can use brush-type effects when you draw lines with a very wide pen. As an example, suppose that you have an image of water, and you want to create the effect that anywhere you draw a line, you let the image of water show through. Clearly, this will not be very interesting if your lines are only 1 pixel wide. Instead, using a wider pen, you can work as though you were filling the mark described by a pen with an effect that you can achieve only with a brush.

In the following example, we'll create a hatch brush, then create a pen from the brush, and then draw a rectangle with the pen. We'll create this effect with a hatch brush, but we could just as easily create the effect with a brush made from an image, or any other kind of brush. We'll give the pen a width of 8 pixels, which is wide enough to ensure that we can see the pattern of the hatch brush where the pen draws:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    HatchBrush hb = new HatchBrush(
             HatchStyle.WideUpwardDiagonal,
             Color.White,
             Color.Black);
    Pen hp = new Pen(hb, 8);
    g.DrawRectangle(hp, 15, 15, 70, 70);
    hb.Dispose();
    hp.Dispose();
}
```

To create the `Pen` object here, we've used one of the overloaded `Pen` constructors you saw near the beginning of this chapter. This constructor expects two arguments: the first argument is a `Brush` object and the second (optional) argument is an integer specifying the pen's width. When run, the example looks like Figure 3-25.

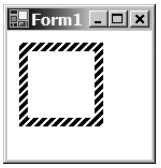

Figure 3-25. *A Pen that uses a Brush*

A Note About Performance

Most professional programmers have had exposure to pens and brushes in the context of graphical drawing. Some toolkits, such as GDI (the predecessor to GDI+), provide methods that combine the shape-*drawing* tasks of a pen and the shape-*filling* tasks of a brush into a single operation. However, as you've seen, in GDI+ these operations are completely separate.

If you want to draw an outline and fill the interior of it, you do it in two distinct operations. This is a very good thing. It results in drawing code that is cleaner and more straightforward. It doesn't really restrict you in any way. It just means that you must make two calls into the library if you want to both draw the outline of a shape and fill the shape.

As a developer, this separation of the drawing and filling tasks might cause you to be concerned about the potential effects on performance, particularly if each shape you draw requires two calls into the Windows API. As it turns out, the cost of marshaling an additional call into the Windows API is negligible, particularly when you compare this to operations such as changing pens and brushes. Experiments with the .NET Framework show a similar performance profile—changing pens is expensive compared with drawing operations with a pen.

If the separation of the drawing and filling tasks enables you to change pens and brushes less often, then the net result is a huge performance gain. Furthermore, if you organize your code so that you draw all lines of a given color before moving on to another color, your code executes much faster. The same is true of filling regions or drawing text.

A LESSON IN PROFILING AND OPTIMIZING CODE

When I needed to get a significant performance gain in some drawing code of a grid control, my first approach was to implement some sophisticated and complicated algorithms and data structures. These eliminated some work at drawing time, and I hoped they would give me the desired performance improvement. These optimizations helped, but not nearly as much as I wanted.

Frustrated, I spent some time profiling the various operations in the underlying toolkit. It was then that I realized the issue: the process of changing from one pen or brush to another is very slow. So, I made a very simple modification to the code: I arranged that the code first drew all of the cells in the grid that had a white background with black text (which was the most common case), and *then* drew all of the exceptional cases with different fonts, colors, and the like. I immediately got a performance increase of nearly an order of magnitude!

Aside from this specific lesson about changing pens and brushes in GDI+, there's a broader lesson that I took away from this experience: when profiling and optimizing code that makes extensive use of *any* underlying toolkit (not just graphical toolkits), be suspicious of the performance profiles of the various methods in the toolkit.

Summary

This chapter covered the essential details behind the fundamental GDI+ concepts of pens and brushes.

In GDI+, you generally use `Pen` objects for drawing shapes. You've seen that you can specify what color the pen should draw in, and you have the option of indicating what width the pen should be. You've also seen that the `Pens` class provides a set of 141 predefined colored pens, all with a 1-pixel width.

For filling operations, you use `Brush` objects. The `Brush` class itself is an `abstract` class, which manifests itself in the form of five special brush types: `SolidBrush`, `TextureBrush`, `LinearGradientBrush`, `PathGradientBrush`, and `HatchBrush`. We looked at how the `TextureBrush` uses an image file to create a textured effect, and how the `LinearGradientBrush` and `PathGradientBrush` use the color gradient technique to achieve their effects.

In passing, you saw that the `Pen` class is not the only class that allows you to draw shapes. In our study of the `PathGradientBrush`, we made use of the `GraphicsPath` class. You'll meet the `GraphicsPath` class again in Chapter 6.

You also learned that you can perform single operations that combine the visual capabilities of a brush with the drawing capabilities of a pen. This gives you a technique that is just like drawing with a brush.

The basic set of functionality provided by `Pen` and `Brush` objects is enough in itself to enable you to draw powerful, effective graphics. But it gets even better when you combine these capabilities with other GDI+ functionality. In the next chapter, you'll see how to use text and fonts to enhance your graphics even further.