DropShadowEffect

DropShadowEffect adds a slightly offset shadow behind an element. You have several properties to play with, as listed in Table 14-3.

Table 14-3. *DropShadowEffect Properties*

Name	Description
Color	Sets the color of the drop shadow (the default is Black).
ShadowDepth	Determines how far the shadow is from the content, in pixels (the default is 5). Use a ShadowDepth of 0 to create an outer-glow effect, which adds a halo of color around your content.
BlurRadius	Blurs the drop shadow, much like the Radius property of BlurEffect (the default is 5).
Opacity	Makes the drop shadow partially transparent, using a fractional value between 1 (fully opaque, the default) and 0 (fully transparent).
Direction	Specifies where the drop shadow should be positioned relative to the content, as an angle from 0 to 360. Use 0 to place the shadow on the right side, and increase the value to move the shadow counterclockwise. The default is 315, which places it to the lower right of the element.

Figure 14-4 shows several different drop-shadow effects on a TextBlock. Here's the markup for all of them:

```
<TextBlock FontSize="20" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Basic dropshadow</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect Color="SlateBlue"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Light blue dropshadow</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="White" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect BlurRadius="15"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Blurred dropshadow with white text</TextBlock.Text>
</TextBlock>
```

```
<TextBlock FontSize="20" Foreground="Magenta" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect ShadowDepth="0"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Close dropshadow</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="LimeGreen" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect ShadowDepth="25"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Distant dropshadow</TextBlock.Text>
</TextBlock>
```

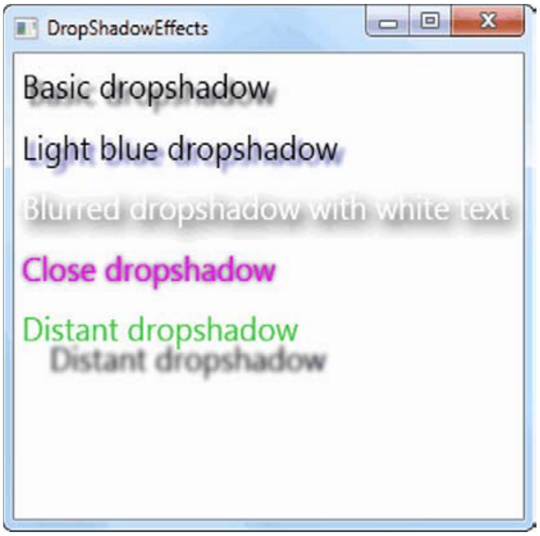

Figure 14-4. Different drop shadows

There is no class for grouping effects, which means you can apply only a single effect to an element at a time. However, you can sometimes simulate multiple effects by adding them to higher-level containers (for example, using the drop-shadow effect for a TextBlock and then placing it in a Stack Panel that uses the blur effect). In most cases, you should avoid this work-around, because it multiplies the rendering work and reduces performance. Instead, look for a single effect that can do everything you need.

ShaderEffect

The ShaderEffect class doesn't represent a ready-to-use effect. Instead, it's an abstract class from which you derive to create your own custom pixel shaders. By using ShaderEffect (or a custom effect that derives from it), you gain the ability to go far beyond mere blurs and drop shadows.

Contrary to what you may expect, the logic that implements a pixel shader isn't written in C# code directly in the effect class. Instead, pixel shaders are written using High Level Shader Language (HLSL), which was created as part of DirectX. (The benefit is obvious—because DirectX and HLSL have been around for many years, graphics developers have already created scores of pixel-shader routines that you can use in your own code.)

To create a pixel shader, you need to write the HLSL code. The first step is to install the DirectX SDK (go to `http://msdn.microsoft.com/en-us/directx/default.aspx`). This gives you enough to create and compile HLSL code to a .ps file (using the fxc.exe command-line tool), which is what you need to use a custom ShaderEffect class. But a more convenient option is to use the free Shazzam tool (`http://shazzam-tool.com`). Shazzam provides an editor for HLSL files, which includes the ability to try them on sample images. It also includes several sample pixel shaders that you can use as the basis for custom effects. More advanced users can try NVidia's free FX Composer tool (`http://developer.nvidia.com/object/fx_composer_home.html`), a shader development tool that's aimed at cutting-edge game developers and other graphics experts.

Although authoring your own HLSL files is beyond the scope of this book, using an existing HLSL file isn't. Once you've compiled your HLSL file to a .ps file, you can use it in a project. Simply add the file to an existing WPF project, select it in the Solution Explorer, and set its Build Action to Resource. Finally, you must create a custom class that derives from ShaderEffect and uses this resource.

For example, if you're using a custom pixel shader that's compiled in a file named Effect.ps, you can use the following code:

```
public class CustomEffect : ShaderEffect
{
    public CustomEffect()
    {
        // Use the URI syntax described in Chapter 7 to refer to your resource.
        // AssemblyName;component/ResourceFileName
        Uri pixelShaderUri = new Uri("Effect.ps", UriKind.Relative);

        // Load the information from the .ps file.
        PixelShader = new PixelShader();
        PixelShader.UriSource = pixelShaderUri;
    }
}
```

You can now use the custom pixel shader in any window. First, make the namespace available by adding a mapping like this:

```
<Window xmlns:local="clr-namespace:CustomEffectTest" ...>
```

Now, create an instance of the custom effect class, and use it to set the Effect property of an element:

```
<Image>
  <Image.Effect>
    <local:CustomEffect></local:CustomEffect>
  </Image.Effect>
</Image>
```

You can get a bit more complicated than this if you use a pixel shader that takes certain input arguments. In this case, you need to create the corresponding dependency properties by calling the static RegisterPixelShaderSamplerProperty() method.

A crafty pixel shader is as powerful as the plug-ins used in graphics software like Adobe Photoshop. It can do anything from adding a basic drop shadow to imposing more ambitious effects like blurs, glows, watery ripples, embossing, sharpening, and so on. Pixel shaders can also create eye-popping effects when they're combined with animation that alters their parameters in real time, as you'll see in Chapter 16.

■ **Tip** Unless you're a hard-core graphics programmer, the best way to get more advanced pixel shaders isn't to write the HLSL yourself. Instead, look for existing HLSL examples or, even better, third-party WPF components that provide custom effect classes. The gold standard is the free Windows Presentation Foundation Pixel Shader Effects Library at `http://codeplex.com/wpffx`. It includes a long list of dazzling effects like swirls, color inversion, and pixelation. Even more useful, it includes transition effects that combine pixel shaders with the animation capabilities described in Chapter 15.

The WriteableBitmap Class

WPF allows you to show bitmaps with the Image element. However, displaying a picture this way is a strictly one-way affair. Your application takes a ready-made bitmap, reads it, and displays it in the window. On its own, the Image element doesn't give you a way to create or edit bitmap information.

This is where WriteableBitmap fits in. It derives from BitmapSource, which is the class you use when setting the Image.Source property (either directly, when you set the image in code, or implicitly, when you set it in XAML). But whereas BitmapSource is a read-only reflection of bitmap data, WriteableBitmap is a modifiable array of pixels that opens up many interesting possibilities.

■ **Note** It's important to realize that the WriteableBitmap isn't the best way for most applications to draw graphical content. If you need a lower-level alternative to WPF's element system, you should begin by checking out the Visual class demonstrated earlier in this chapter. For example, the Visual class is the perfect tool for creating a charting tool or a simple animated game. The WriteableBitmap is better suited to applications that need to manipulate individual pixels—for example, a fractal generator, a sound analyzer, a visualization tool for scientific data, or an application that processes raw image data from an external hardware device (like a webcam). Although the WriteableBitmap gives you fine-grained control, it's complex and requires much more code than the other approaches.

Generating a Bitmap

To generate a bitmap with WriteableBitmap, you must supply a few key pieces of information: its width and height in pixels, its DPI resolution in both dimensions, and the image format.

Here's an example that creates an image as big as the current window:

```
WriteableBitmap wb = new WriteableBitmap((int)this.ActualWidth,
  (int)this.ActualHeight, 96, 96, PixelFormats.Bgra32, null);
```

The PixelFormats enumeration has a long list of pixel formats, but only about half are considered writeable formats and are supported by the WriteableBitmap class. Here are the ones you can use:

- **Bgra32.** This format (the one used in the current example) uses 32-bit sRGB color. That means that each pixel is represented by 32 bits, or 4 bytes. The first byte represents the contribution of the blue channel (as a number from 0 to 255). The second byte is for the green channel, the third is for the red channel, and the fourth is for the alpha value (where 0 is completely transparent and 255 is completely opaque). As you can see, the order of the colors (blue, green, red, alpha) matches the letters in the name *Bgra*32.
- **Bgr32.** This format uses 4 bytes per pixel, just like Bgra32. The difference is that the alpha channel is ignored. You can use this format when transparency is not required.
- **Pbgra32.** This format uses 4 bytes per pixel, just like Bgra32. The difference is the way it handles semitransparent pixels. In order to optimize the performance of opacity calculations, each color byte is *premultiplied* (hence the *P* in Pbgra32). This means each color byte is multiplied by the alpha value and divided by 255. So a partially transparent pixel that has the B, G, R, A values (255, 100, 0, 200) in Bgra32 would become (200, 78, 0, 200) in Pbgra32.
- **BlackWhite, Gray2, Gray4, Gray8.** These are the black-and-white and grayscale formats. The number following the word *Gray* corresponds to the number of bits per pixel. Thus, these formats are compact, but they don't support color.
- **Indexed1, Indexed2, Indexed4, Indexed8.** These are indexed formats, which means that each pixel points to a value in a color palette. When using one of these formats, you must pass the corresponding ColorPalette object as the last WriteableBitmap constructor argument. The number following the word *Indexed* corresponds to the number of bits per pixel. The indexed formats are compact, slightly more complex to use, and support far fewer colors—2, 14, 16, or 256 colors, respectively.

The top three formats—Bgra32, Bgr32, and Pbgra32—are by far the most common choices.

Writing to a WriteableBitmap

A WriteableBitmap begins with 0 values for all its bytes. Essentially, it's a big, black rectangle.

To fill a WriteableBitmap with content, you use the WritePixels() method. WritePixels() copies an array of bytes into the bitmap at the position you specify. You can call WritePixels() to set a single pixel, the entire bitmap, or a rectangular region that you choose. To get pixels out of the WriteableBitmap, you use the CopyPixels() method, which transfers the bytes you want into a byte array. Taken together, the WritePixels() and CopyPixels() methods don't give you the most convenient programming model to work with, but that's the cost of low-level pixel access.

To use WritePixels() successfully, you need to understand your image format and how it encodes pixels into bytes. For example, in a 32-bit bitmap type Bgra32, each pixel requires 4 bytes, one each for the blue, green, red, and alpha components. Here's how you can set them by hand, and then transfer them into an array:

```
byte blue =100;
byte green = 50;
byte red = 50;
byte alpha = 255;

byte[] colorData = {blue, green, red, alpha};
```

Note that the order is critical here. The byte array must follow the blue, green, red, alpha sequence set out in the Bgra32 standard.

When you call WritePixels(), you supply an Int32Rect that indicates the rectangular region of the bitmap that you want to update. The Int32Rect wraps four pieces of information: the X and Y coordinate of the top-left corner of the update region, and the width and height of the update region.

The following code takes the colorData array shown in the preceding code and uses it to set the first pixel in the WriteableBitmap:

```
// Update a single pixel. It's a region starting at (0,0)
// that's 1 pixel wide and 1 pixel high.
Int32Rect rect = new Int32Rect(0, 0, 1, 1);

// Write the 4 bytes from the array into the bitmap.
wb.WritePixels(rect, colorData, 4, 0);
```

Using this approach, you could create a code routine that generates a WriteableBitmap. It simply needs to loop over all the columns and rows in the image, updating a single pixel in each iteration.

```
for (int x = 0; x < wb.PixelWidth; x++)
{
    for (int y = 0; y < wb.PixelHeight; y++)
    {
        // Pick a pixel color using a formula of your choosing.
        byte blue = ...
        byte green = ...
        byte red = ...
        byte alpha = ...

        // Create the byte array.
        byte[] colorData = {blue, green, red, alpha};

        // Pick the position where the pixel will be drawn.
        Int32Rect rect = new Int32Rect(x, y, 1, 1);

        // Calculate the stride.
        int stride = wb.PixelWidth * wb.Format.BitsPerPixel / 8;

        // Write the pixel.
        wb.WritePixels(rect, colorData, stride, 0);
    }
}
```

This code includes one additional detail: a calculation for the *stride*, which the WritePixels() method requires. Technically, the stride is the number of bytes required for each row of pixel data. You can calculate this by multiplying the number of pixels in a row by the number of bits in a pixel for your format (usually 4, as with the Bgra32 format used in this example), and then dividing the result by 8 to convert it from bits to bytes.

After the pixel-generating process is finished, you need to display the final bitmap. Typically, you'll use an Image element to do the job:

```
img.Source = wb;
```

Even after writing and displaying a bitmap, you're still free to read and modify pixels in the WriteableBitmap. This gives you the ability to build more specialized routines for bitmap editing and bitmap hit testing.

More Efficient Pixel Writing

Although the code shown in the previous section works, it's not the best approach. If you need to write a large amount of pixel data at once—or even the entire image—you're better off using bigger chunks That's because there's a certain amount of overhead for calling WritePixels(), and the more often you call it, the longer you'll delay your application.

Figure 14-5 shows a test application that's included with the samples for this chapter. It creates a dynamic bitmap by filling pixels with a mostly random pattern interspersed with regular gridlines. The downloadable code performs this task in two different ways: using the pixel-by-pixel approach explained in the previous section and using the single-write strategy you'll see next. If you test this application, you'll find that the single-write technique is far faster.

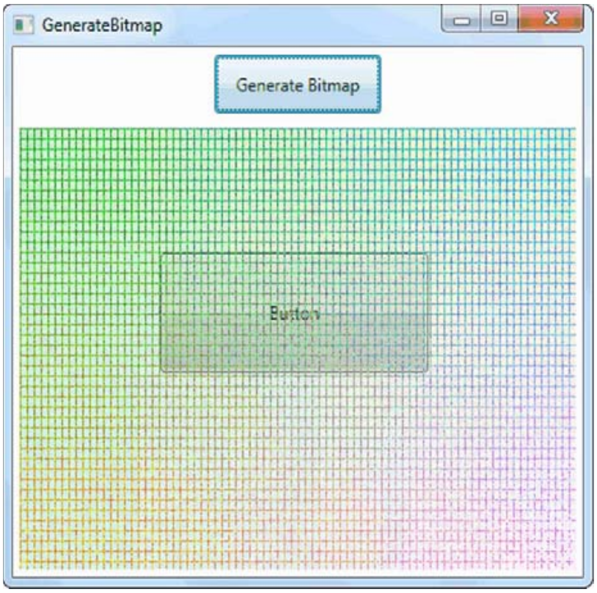

Figure 14-5. *A dynamically generated bitmap*

■ **Tip** For a more practical (and much longer) example of the WriteableBitmap at work, check out the example at `http://tinyurl.com/y8hnvsl`, which uses it to model a chemical reaction.

In order to update more than one pixel at once, you need to understand how the pixels are packaged together in your byte array. Regardless of the format you're using, your update buffer will hold a one-dimensional array of bytes. This array supplies values for the pixels in a rectangular section of the image, stretching from left to right to fill each row, and then from top to bottom.

To find a specific pixel, you need to use the following formula, which steps down the number of rows and then moves to the appropriate position in that row:

```
(y × wb.PixelWidth + x) × BytesPerPixel
```

For example, to set the pixel (40, 100) in a Bgra32 bitmap (which has 4 bytes per pixel), you use this code:

```
int pixelOffset = (40 +100 * wb.PixelWidth) * wb.Format.BitsPerPixel/8;
pixels[pixelOffset] = blue;
pixels[pixelOffset + 1] = green;
pixels[pixelOffset + 2] = red;
pixels[pixelOffset + 3] = alpha;
```

With that in mind, here's the complete code that creates the bitmap shown in Figure 14-5, by first filling all the data in a single array, and then copying it to the WriteableBitmap with just one call to WritePixels():

```
// Create the bitmap, with the dimensions of the image placeholder.
WriteableBitmap wb = new WriteableBitmap((int)img.Width,
  (int)img.Height, 96, 96, PixelFormats.Bgra32, null);

// Define the update square (which is as big as the entire image).
Int32Rect rect = new Int32Rect(0, 0, (int)img.Width, (int)img.Height);

byte[] pixels = new byte[(int)img.Width * (int)img.Height *
  wb.Format.BitsPerPixel / 8];
Random rand = new Random();
for (int y = 0; y < wb.PixelHeight; y++)
{
    for (int x = 0; x < wb.PixelWidth; x++)
    {
        int alpha = 0;
        int red = 0;
        int green = 0;
        int blue = 0;

        // Determine the pixel's color.
        if ((x % 5 == 0) || (y % 7 == 0))
        {
            red = (int)((double)y / wb.PixelHeight * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)x / wb.PixelWidth * 255);
            alpha = 255;
        }
        else
        {
            red = (int)((double)x / wb.PixelWidth * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)y / wb.PixelHeight * 255);
            alpha = 50;
        }

        int pixelOffset = (x + y * wb.PixelWidth) * wb.Format.BitsPerPixel/8;
        pixels[pixelOffset] = (byte)blue;
        pixels[pixelOffset + 1] = (byte)green;
        pixels[pixelOffset + 2] = (byte)red;
        pixels[pixelOffset + 3] = (byte)alpha;
    }

    // Copy the byte array into the image in one step.
    int stride = (wb.PixelWidth * wb.Format.BitsPerPixel) / 8;
    wb.WritePixels(rect, pixels, stride, 0);
}

// Show the bitmap in an Image element.
img.Source = wb;
```

In a realistic application, you're likely to choose an approach that falls somewhere in between. You won't write single pixels a time if you need to update large sections of a bitmap, because that approach would probably be prohibitively slow. But you won't hold all of the image data in memory at once, because it could be very large. (After all, a 1000 × 1000 pixel image that requires 4 bytes per pixel needs nearly 4MB of memory, which is not yet excessive but not trivial either.) Instead, you should aim to write large chunks of image data rather than individual pixels, especially if you're generating an entire bitmap at once.

■ **Tip** If you need to make frequent updates to the image data in a WriteableBitmap, and you want to make these updates from another thread, you can optimize the code even more using the WriteableBitmap back buffer. The basic process is this: use the Lock() method to reserve the back buffer, obtain a pointer to the back buffer, update it, indicate the changed region by calling AddDirtyRect(), and then release the back buffer by calling Unlock(). This process requires unsafe code, and is beyond the scope of this book, but you can see a basic example in the Visual Studio help under the WriteableBitmap topic.

The Last Word

In this chapter, you looked at three topics that go beyond WPF's standard 2-D drawing support. First, you tackled the lower-level visual layer, which is the most efficient way to display graphics in WPF. Using the visual layer, you saw how you could build a basic drawing application that uses sophisticated hit testing. Next, you learned about pixel shaders, a way to fuse graphical effects originally designed for next-generation games into any WPF application. Not only are pixel shaders nearly effortless to use, but there's already a huge library of free pixel shaders that you can drop into your applications right now. Finally, you considered the WriteableBitmap, a powerful but more limited tool that lets you create a bitmap image, and directly manipulate the individual pixels that comprise it.

CHAPTER 15

■ ■ ■

Animation Basics

Animation allows you to create truly *dynamic* user interfaces. It's often used to apply effects—for example, icons that grow when you move over them, logos that spin, text that scrolls into view, and so on. Sometimes, these effects seem like excessive glitz. But used properly, animations can enhance an application in a number of ways. They can make an application seem more responsive, natural, and intuitive. (For example, a button that slides in when you click it feels like a real, physical button—not just another gray rectangle.) Animations can also draw attention to important elements and guide the user through transitions to new content. (For example, an application could advertise newly downloaded content with a twinkling icon in a status bar.)

Animations are a core part of the WPF model. That means you don't need to use timers and event handling code to put them into action. Instead, you can create them declaratively, configure them using one of a handful of classes, and put them into action without writing a single line of C# code. Animations also integrate themselves seamlessly into ordinary WPF windows and pages. For example, if you animate a button so it drifts around the window, the button still behaves like a button. It can be styled, it can receive focus, and it can be clicked to fire off the typical event handling code. This is what separates animation from traditional media files, such as video. (In Chapter 26, you'll learn how to put a video window in your application. A video window is a completely separate region of your application—it's able to play video content, but it's not user interactive.)

In this chapter, you'll consider the rich set of animation classes that WPF provides. You'll see how to use them in code and (more commonly) how to construct and control them with XAML. Along the way, you'll see a wide range of animation examples, including fading pictures, rotating buttons, and expanding elements.

■ **What's New** WPF 4 adds a feature called *animation easing*, which uses mathematical formulas to create more natural animated effects. You'll learn how it works in the "Animation Easing" section. WPF 4 also supports *bitmap caching*, a form of hardware acceleration that allows you to optimize CPU usage when dealing with certain types of animation. You'll learn about this technique in the "Bitmap Caching" section.

Understanding WPF Animation

In previous Windows-based platforms (such as Windows Forms and MFC), developers had to build their own animation systems from scratch. The most common technique was to use a timer in conjunction with some custom painting logic. WPF changes the game with a new *property-based* animation system. The following two sections describe the difference.

Timer-Based Animation

Imagine you need to make a piece of text spin in the About box of a Windows Forms application. Here's the traditional way you would structure your solution:

1. Create a timer that fires periodically (say, every 50 milliseconds).
2. When the timer fires, use an event handler to calculate some animation-related details, such as the new degree of rotation. Then, invalidate part or all of the window.
3. Shortly thereafter, Windows will ask the window to repaint itself, triggering your custom painting code.
4. In your painting code, render the rotated text.

Although this timer-based solution isn't very difficult to implement, integrating it into an ordinary application window is more trouble than it's worth. Here are some of the problems:

- **It paints pixels, not controls.** To rotate text in Windows Forms, you need the lower-level GDI+ drawing support. It's easy enough to use, but it doesn't mix well with ordinary window elements, such as buttons, text boxes, labels, and so on. As a result, you need to segregate your animated content from your controls, and you can't incorporate any user-interactive elements into an animation. If you want a rotating button, you're out of luck.
- **It assumes a single animation.** If you decide you want to have two animations running at the same time, you need to rewrite all your animation code—and it could become much more complex. WPF is much more powerful in this regard, allowing you to build more complex animations out of individual, simpler animations.
- **The animation frame rate is fixed.** It's whatever the timer is set at. And if you change the timer interval, you might need to change your animation code (depending on how your calculations are performed). Furthermore, the fixed frame rate you choose is not necessarily the ideal one for the computer's video hardware.
- **Complex animations require exponentially more complex code.** The spinning text example is easy enough, but moving a small vector drawing along a path is quite a bit more difficult. In WPF, even intricate animations can be defined in XAML (and generated with a third-party design tool).

Even without WPF's animation support, you can already simplify the spinning text example. That's because WPF provides a retained graphics model, which ensures that a window is automatically rerendered when it changes. This means you don't need to worry about invalidating and repainting it yourself. Instead, the following steps work just fine:

1. Create a timer that fires periodically. (WPF provides a System.Windows.Threading.DispatcherTimer that works on the user interface thread.)
2. When the timer fires, use an event handler to calculate some animation-related details, such as the new degree of rotation. Then, modify the corresponding elements.
3. WPF notices the changes you've made to the elements in your window. It then repaints (and caches) the new window content.

With this new solution, you don't need to fiddle with low-level drawing classes, and you don't need to segregate your animated content from ordinary elements in the same window.

Although this is an improvement, timer-based animation still suffers from several flaws: it results in code that isn't very flexible, it becomes horribly messy for complex effects, and it doesn't get the best possible performance. Instead, WPF includes a higher-level model that allows you to focus on *defining* your animations, without worrying about the way they're rendered. This model is based on the dependency property infrastructure, which is described in the next section.

Property-Based Animation

Often, an animation is thought of as a series of frames. To perform the animation, these frames are shown one after the other, like a stop-motion video. WPF animations use a dramatically different model. Essentially, a WPF animation is simply a way to modify the value of a dependency property over an interval of time.

For example, to make a button that grows and shrinks, you can modify its Width property in an animation. To make it shimmer, you could change the properties of the LinearGradientBrush that it uses for its background. The secret to creating the right animation is determining what properties you need to modify.

If you want to make other changes that can't be made by modifying a property, you're out of luck. For example, you can't add or remove elements as part of animation. Similarly, you can't ask WPF to perform a transition between a starting scene and an ending scene (although some crafty workarounds can simulate this effect). And finally, you can use animation only with a dependency property, because only dependency properties use the dynamic property resolution system (described in Chapter 4) that takes animations into account.

At first glance, the property-focused nature of WPF animations seems terribly limiting. However, as you work with WPF, you'll find that it's surprisingly capable. In fact, you can create a wide range of animated effects using common properties that every element supports.

That said, there are many cases where the property-based animation system won't work. As a rule of thumb, the property-based animation is a great way to add dynamic effects to otherwise ordinary Windows applications. For example, if you want a slick front end for your interactive shopping tool, property-based animations will work perfectly well. However, if you need to use animations as part of the core purpose of your application and you want them to continue running over the lifetime of your application, you probably need something more flexible and more powerful. For example, if you're creating a basic arcade game or using complex physics calculations to model collisions, you'll need greater control over the animation. In these situations, you'll be forced to do most of the work yourself using WPF's lower-level frame-based rendering support, which is described in Chapter 16.

Basic Animation

You've already learned the first rule of WPF animation—every animation acts on a single dependency property. However, there's another restriction. To animate a property (in other words, change its value in a time-dependent way), you need to have an animation class that supports its data type. For example, the Button.Width property uses the double data type. To animate it, you use the DoubleAnimation class. However, Button.Padding uses the Thickness structure, so it requires the ThicknessAnimation class.

This requirement isn't as absolute as the first rule of WPF animation, which limits animations to dependency properties. That's because you can animate a dependency property that doesn't have a corresponding animation class by creating your *own* animation class for that data type. However, you'll find that the System.Windows.Media.Animation namespace includes animation classes for most of the data types that you'll want to use.

Many data types don't have a corresponding animation class because it wouldn't be practical. A prime example is enumerations. For example, you can control how an element is placed in a layout panel using the HorizontalAlignment property, which takes a value from the HorizontalAlignment enumeration. However, the HorizontalAlignment enumeration allows you to choose between only four values (Left, Right, Center, and Stretch), which greatly limits its use in an animation. Although you can swap between one orientation and another, you can't smoothly transition an element from one alignment to another. For that reason, there's no animation class for the HorizontalAlignment data type. You can build one yourself, but you're still constrained by the four values of the enumeration.

Reference types are not usually animated. However, their subproperties are. For example, all content controls sport a Background property that allows you to set a Brush object that's used to paint the background. It's rarely efficient to use animation to switch from one brush to another, but you can use animation to vary the properties of a brush. For example, you could vary the Color property of a SolidColorBrush (using the ColorAnimation class) or the Offset property of a GradientStop in a LinearGradientBrush (using the DoubleAnimation class). This extends the reach of WPF animation, allowing you to animate specific aspects of an element's appearance.

The Animation Classes

Based on the animation types mentioned so far—DoubleAnimation and ColorAnimation—you might assume all animation classes are named in the form *TypeName*Animation. This is close but not exactly true.

There are actually two types of animations—those that vary a property incrementally between the starting and finishing values (a process called *linear interpolation*) and those that abruptly change a property from one value to another. DoubleAnimation and ColorAnimation are examples of the first category; they use interpolation to smoothly change the value. However, interpolation doesn't make sense when changing certain data types, such as strings and reference type objects. Rather than use interpolation, these data types are changed abruptly at specific times using a technique called *key frame animation*. All key frame animation classes are named in the form *TypeName*AnimationUsingKeyFrames, as in StringAnimationUsingKeyFrames and ObjectAnimationUsingKeyFrames.

Some data types have a key frame animation class but no interpolation animation class. For example, you can animate a string using key frames, but you can't animate a string using interpolation. However, *every* data type supports key frame animations, unless they have no animation support at all. In other words, every data type that has a normal animation class that uses interpolation (such as DoubleAnimation and ColorAnimation) also has a corresponding animation type for key frame animation (such as DoubleAnimationUsingKeyFrames and ColorAnimationUsingKeyFrames).

Truthfully, there's still one more type of animation. The third type is called a *path-based animation*, and it's much more specialized than animation that uses interpolation or key frames. A path-based animation modifies a value to correspond with the shape that's described by a PathGeometry object, and it's primarily useful for moving an element along a path. The classes for path-based animations have names in the form *TypeName*AnimationUsingPath, such as DoubleAnimationUsingPath and PointAnimationUsingPath.

■ **Note** Although WPF currently uses three approaches to animation (linear interpolation, key frames, and paths), there's no reason you can't create more animation classes that modify values using a completely different approach. The only requirement is that your animation class must modify values in a time-dependent way.

All in all, you'll find the following in the System.Windows.Media.Animation namespace:

- Seventeen *TypeName*Animation classes, which use interpolation
- Twenty-two *TypeName*AnimationUsingKeyFrames classes, which use key frame animation
- Three *TypeName*AnimationUsingPath classes, which use path-based animation

Every one of these animation classes derives from an abstract *TypeName*AnimationBase class that implements a few fundamentals. This gives you a shortcut to creating your own animation classes. If a data type supports more than one type of animation, both animation classes derive from the abstract animation base class. For example, DoubleAnimation and DoubleAnimationUsingKeyFrames both derive from DoubleAnimationBase.

■ **Note** These 42 classes aren't the only things you'll find in the System.Windows.Media.Animation namespace. Every key frame animation also works with its own key frame class and key frame collection classes, which adds to the clutter. In total, there are more than 100 classes in System.Windows.Media.Animation.

You can quickly determine what data types have native support for animation by reviewing these 42 classes. The following is the complete list:

BooleanAnimationUsingKeyFrames	ByteAnimation
ByteAnimationUsingKeyFrames	CharAnimationUsingKeyFrames
ColorAnimation	ColorAnimationUsingKeyFrames
DecimalAnimation	DecimalAnimationUsingKeyFrames
DoubleAnimation	DoubleAnimationUsingKeyFrames

DoubleAnimationUsingPath	Int16Animation
Int16AnimationUsingKeyFrames	Int32Animation
Int32AnimationUsingKeyFrames	Int64Animation
Int64AnimationUsingKeyFrames	MatrixAnimationUsingKeyFrames
MatrixAnimationUsingPath	ObjectAnimationUsingKeyFrames
PointAnimation	PointAnimationUsingKeyFrames
PointAnimationUsingPath	Point3DAnimation
Point3DAnimationUsingKeyFrames	QuarternionAnimation
QuarternionAnimationUsingKeyFrames	RectAnimation
RectAnimationUsingKeyFrames	Rotation3DAnimation
Rotation3DAnimationUsingKeyFrames	SingleAnimation
SingleAnimationUsingKeyFrames	SizeAnimation
SizeAnimationUsingKeyFrames	StringAnimationUsingKeyFrames
ThicknessAnimation	ThicknessAnimationUsingKeyFrames
VectorAnimation	VectorAnimationUsingKeyFrames
Vector3DAnimation	Vector3DAnimationUsingKeyFrames

Many of these types are self-explanatory. For example, once you master the DoubleAnimation class, you won't think twice about SingleAnimation, Int16Animation, Int32Animation, and all the other animation classes for simple numeric types, which work in the same way. Along with the animation classes for numeric types, you'll find a few that work with other basic data types (byte, bool, string, and char) and many more that deal with two-dimensional and three-dimensional Drawing primitives (Point, Size, Rect, Vector, and so on). You'll also find an animation class for the Margin and Padding properties of any element (ThicknessAnimation), one for color (ColorAnimation), and one for any reference type object (ObjectAnimationUsingKeyFrames). You'll consider many of these animation types as you work through the examples in this chapter.

The Cluttered Animation Namespace

If you look in the System.Windows.Media.Animation namespace, you may be a bit shocked. It's packed full with different animation classes for different data types. The effect is a bit overwhelming. It would be nice if there were a way to combine all the animation features into a few core classes. And what developer wouldn't appreciate a generic Animate<T> class that could work with any data type? However, this model isn't currently possible, for a variety of reasons. First, different animation classes may perform their work in slightly different ways, which means the code required will differ. For example, the way a color value is blended from one shade to another by the ColorAnimation class differs from the way a single numeric value is modified by the DoubleAnimation class. In other words, although the animation classes expose the same public interface for you to use, their internal workings may differ. Their interface is standardized through inheritance, because all animation classes derive from the same base classes (beginning with Animatable).

However, this isn't the full story. Certainly, many animation classes *do* share a significant amount of code, and a few areas absolutely cry out for a dash of generics, such as the 100 or so classes used to represent key frames and key frame collections. In an ideal world, animation classes would be distinguished by the type of animation they perform, so you could use classes such as NumericAnimation<T>, KeyFrameAnimation<T>, or LinearInterpolationAnimation<T>. One can only assume that the deeper reason that prevents solutions like these is that XAML lacks direct support for generics.

Animations in Code

As you've already learned, the most common animation technique is linear interpolation, which modifies a property smoothly from its starting point to its end point. For example, if you set a starting value of 1 and an ending value of 10, your property might be rapidly changed from 1 to 1.1, 1.2, 1.3, and so on, until the value reaches 10.

At this point, you're probably wondering how WPF determines the increments it will use when performing interpolation. Happily, this detail is taken care of automatically. WPF uses whatever increment it needs to ensure a smooth animation at the currently configured frame rate. The standard frame rate WPF uses is 60 frames per second. (You'll learn how to tweak this detail later in this chapter.) In other words, every 1/60th of a second WPF calculates all animated values and updates the corresponding properties.

The simplest way to use an animation is to instantiate one of the animation classes listed earlier, configure it, and then use the BeginAnimation() of the element you want to modify. All WPF elements inherit BeginAnimation(), which is part of the IAnimatable interface, from the base UIElement class. Other classes that implement IAnimatable include ContentElement (the base class for bits of document flow content) and Visual3D (the base class for 3D visuals).

■ **Note** This isn't the most common approach—it most situations, you'll create animations declaratively using XAML, as described later in the "Storyboards" section. However, using XAML is slightly more involved because you need another object—called a *storyboard*—to connect the animation to the appropriate property. Code-based animations are also useful in certain scenarios where you need to use complex logic to determine the starting and ending values for your animation.

Figure 15-1 shows an extremely simple animation that widens a button. When you click the button, WPF smoothly extends both sides until the button fills the window.

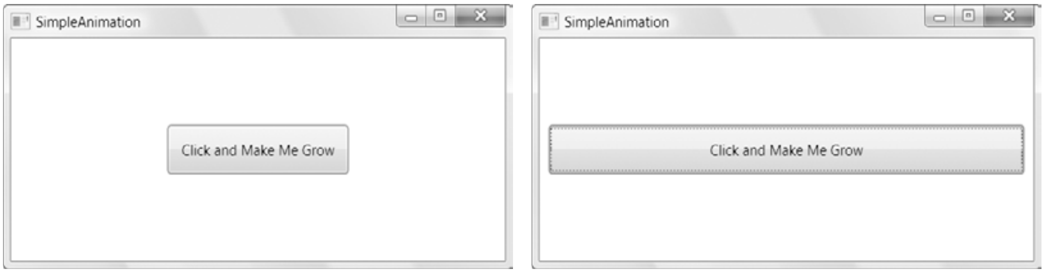

Figure 15-1. *An animated button*

To create this effect, you use an animation that modifies the Width property of the button. Here's the code that creates and launches this animation when the button is clicked:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 160;
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

Three details are the bare minimum of any animation that uses linear interpolation: the starting value (From), the ending value (To), and the time that the entire animation should take (Duration). In this example, the ending value is based on the current width of the containing window. These three properties are found in all the animation classes that use interpolation.

The From, To, and Duration properties seem fairly straightforward, but you should note a few important details. The following sections explore these properties more closely.

From

The From value is the starting value for the Width property. If you click the button multiple times, each time you click it the Width is reset to 160, and the animation runs again. This is true even if you click the button while an animation is already underway.

■ **Note** This example exposes another detail about WPF animations; namely, every dependency property can be acted on by only one animation at a time. If you start a second animation, the first one is automatically discarded.

In many situations, you don't want an animation to begin at the original From value. There are two common reasons:

- **You have an animation that can be triggered multiple times in a row for a cumulative effect.** For example, you might want to create a button that grows a bit more each time it's clicked.
- **You have animations that may overlap.** For example, you might use the MouseEnter event to trigger an animation that expands a button and the MouseLeave event to trigger a complementary animation that shrinks it back. (This is often known as a "fish-eye" effect.) If you move the mouse over and off this sort of button several times in quick succession, each new animation will interrupt the previous one, causing the button to "jump" back to the size that's set by the From property.

The current example falls into the second category. If you click the button while it's already growing, the width is reset to 160 pixels—which can be a bit jarring. To correct the problem, just leave out the code statement that sets the From property:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

There's one catch. For this technique to work, the property you're animating must have a previously set value. In this example, that means the button must have a hard-coded width (whether it's defined directly in the button tag or applied through a style setter). The problem is that in many layout containers, it's common not to specify a width and to allow the container to control it based on the element's alignment properties. In this case, the default width applies, which is the special value Double.NaN (where NaN stands for "not a number"). You can't animate a property that has this value using linear interpolation.

So, what's the solution? In many cases, the answer is to hard-code the button's width. As you'll see, animations often require a more fine-grained control of element sizing and positioning than you'd otherwise use. In fact, the most common layout container for "animatable" content is the Canvas, because it makes it easy to move content around (with possible overlap) and resize it. The Canvas is also the most lightweight layout container, because no extra layout work is needed when a property like Width is changed.

In the current example, there's another option. You could retrieve the current value of the button using its ActualWidth property, which indicates the current rendered width. You can't animate ActualWidth (it's read-only), but you can use it to set the From property of your animation:

```
widthAnimation.From = cmdGrow.ActualWidth;
```

This technique works for both code-based animations (like the current example) and the declarative animations you'll see later (which require the use of a binding expression to get the ActualWidth value).

■ **Note** It's important to use the ActualWidth property in this example rather than the Width property. That's because Width reflects the desired width that you choose, while ActualWidth indicates the rendered width that was used. If you're using automatic layout, you probably won't set a hard-coded Width at all, so the Width property will simply return Double.NaN, and an exception will be raised when you attempt to start the animation.

You need to be aware of another issue when you use the current value as a starting point for an animation—it may change the speed of your animation. That's because the duration isn't adjusted to take into account that there's a smaller spread between the initial value and the final value. For example, imagine you create a button that doesn't use the From value and instead animates from its current position. If you click the button when it has almost reached its maximum width, a new animation begins. This animation is configured to take five seconds (through the Duration property), even though there are only a few more pixels to go. As a result, the growth of the button will appear to slow down.

This effect appears only when you restart an animation that's almost complete. Although it's a bid odd, most developers don't bother trying to code around it. Instead, it's considered to be an acceptable quirk.

■ **Note** You could compensate for this problem by writing some custom logic that modifies the animation duration, but it's seldom worth the effort. To do so, you'd need to make assumptions about the standard size of the button (which limits the reusability of your code), and you'd need to create your animations programmatically so that you could run this code (rather than declaratively, which is the more common approach you'll see a bit later).

To

Just as you can omit the From property, you can omit the To property. In fact, you could leave out both the From and To properties to create an animation like this:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.Duration = TimeSpan.FromSeconds(5);
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

At first glance, this animation seems like a long-winded way to do nothing at all. It's logical to assume that because both the To and From properties are left out, they'll both use the same value. But there's a subtle and important difference.

When you leave out From, the animation uses the current value and takes animation into account. For example, if the button is midway through a grow operation, the From value uses the expanded width. However, when you leave out To, the animation uses the current value *without taking animation into account*. Essentially, that means the To value becomes the *original* value—whatever you last set in code, on the element tag, or through a style. (This works thanks to WPF's property resolution system, which is able to calculate a value for a property based on several overlapping property providers, without discarding any information. Chapter 4 describes this system in more detail.)

In the button example, that means if you start a grow animation and then interrupt it with the animation shown previously (perhaps by clicking another button), the button will shrink from its