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 several 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 by 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.

Understanding WPF Animation

In many user frameworks (particularly ones that predate WPF, such as Windows Forms and MFC), developers need to build their own animation systems from scratch. The most common technique is to use a timer in conjunction with some custom painting logic. WPF is different—it includes a built-in *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 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).

Timer-based animation still suffers from several flaws: it results in code that isn't very flexible, it becomes messy for complex effects, and it doesn't get the best possible performance.

Property-Based Animation

WPF uses 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. 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 by using common properties that every element supports.

That said, in many cases the property-based animation system won't work. As a rule of thumb, 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 by 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 by 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 that 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 by 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 by using key frames, but you can't animate a string by 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 *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
ByteAnimationUsingKeyFrames
ColorAnimation
DecimalAnimation
DoubleAnimation
DoubleAnimationUsingPath
Int16AnimationUsingKeyFrames
Int32AnimationUsingKeyFrames
Int64AnimationUsingKeyFrames
MatrixAnimationUsingPath
PointAnimation
PointAnimationUsingPath
Point3DAnimationUsingKeyFrames
QuarternionAnimationUsingKeyFrames
RectAnimationUsingKeyFrames
Rotation3DAnimationUsingKeyFrames
SingleAnimationUsingKeyFrames
SizeAnimationUsingKeyFrames
ThicknessAnimation
VectorAnimation
Vector3DAnimation
ByteAnimation
CharAnimationUsingKeyFrames
ColorAnimationUsingKeyFrames
DecimalAnimationUsingKeyFrames
DoubleAnimationUsingKeyFrames
Int16Animation
Int32Animation
Int64Animation
MatrixAnimationUsingKeyFrames
ObjectAnimationUsingKeyFrames
PointAnimationUsingKeyFrames
Point3DAnimation
QuarternionAnimation
RectAnimation
Rotation3DAnimation
SingleAnimation
SizeAnimation
StringAnimationUsingKeyFrames
ThicknessAnimationUsingKeyFrames
VectorAnimationUsingKeyFrames
Vector3DAnimationUsingKeyFrames

Many of these types are self-explanatory. For example, after 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() method 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** Using the BeginAnimation() method isn't the most common approach—it most situations, you'll create animations declaratively by using XAML, as described later in the "Working with Storyboards and Event Triggers" 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 in which 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.

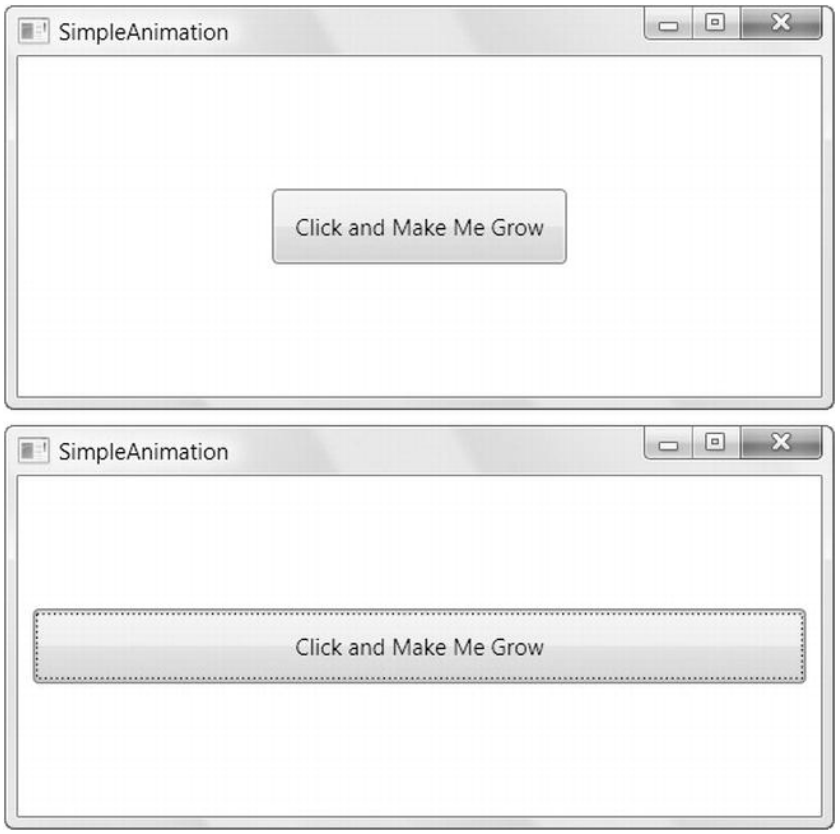

Figure 15-1. *An animated button*

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 by 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 such as Width is changed.

In the current example, there's another option. You could retrieve the current value of the button by 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 (such as 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 that 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 5 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 bit 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 half-

grown size until it reaches the original width that's set in the XAML markup. On the other hand, if you run this code while no other animation is underway, nothing will happen. That's because the From value (the animated width) and the To value (the original width) are the same.

By

Instead of using To, you can use the By property. The By property is used to create an animation that changes a value *by* a set amount, rather than *to* a specific target. For example, you could create an animation that enlarges a button by 10 units more than its current size, as shown here:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.By = 10;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
cmdGrowIncrementally.BeginAnimation(Button.WidthProperty, widthAnimation);
```

This approach isn't necessary in the button example, because you could achieve the same result by using a simple calculation to set the To property, like this:

```
widthAnimation.To = cmdGrowIncrementally.Width + 10;
```

However, the By value makes more sense when you're defining your animation in XAML, because XAML doesn't provide a way to perform simple calculations.

■ **Note** You can use By and From in combination, but it doesn't save you any work. The By value is simply added to the From value to arrive at the To value.

The By property is offered by most, but not all, animation classes that use interpolation. For example, it doesn't make sense with non-numeric data types, such as a Color structure (as used by ColorAnimation).

There's one other way to get similar behavior without using the By property—you can create an *additive* animation by setting the IsAdditive property. When you do, the current value is added to both the From and To values automatically. For example, consider this animation:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 0;
widthAnimation.To = -10;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
widthAnimation.IsAdditive = true;
```

It starts from the current value and finishes at a value that's reduced by 10 units. On the other hand, if you use this animation:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 10;
widthAnimation.To = 50;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
widthAnimation.IsAdditive = true;
```

the property jumps to the new value (which is 10 units greater than the current value) and then increases until it reaches a final value that is 50 units more than the current value before the animation began.

Duration

The Duration property is straightforward enough—it takes the time interval (in milliseconds, minutes, hours, or whatever else you'd like to use) between the time the animation starts and the time it ends. Although the duration of the animations in the previous examples is set by using a TimeSpan, the Duration property actually requires a Duration object. Fortunately, Duration and TimeSpan are quite similar, and the Duration structure defines an implicit cast that can convert System.TimeSpan to System.Windows.Duration as needed. That's why this line of code is perfectly reasonable:

```
widthAnimation.Duration = TimeSpan.FromSeconds(5);
```

So, why bother introducing a whole new type? The Duration also includes two special values that can't be represented by a TimeSpan object—Duration.Automatic and Duration.Forever. Neither of these values is useful in the current example. (Automatic simply sets the animation to a 1-second duration, and Forever makes the animation infinite in length, which prevents it from having any effect.) However, these values become useful when creating more-complex animations.

Simultaneous Animations

You can use BeginAnimation() to launch more than one animation at a time. The BeginAnimation() method returns almost immediately, allowing you to use code like this to animate two properties simultaneously:

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

DoubleAnimation heightAnimation = new DoubleAnimation();
heightAnimation.From = 40;
heightAnimation.To = this.Height - 50;
heightAnimation.Duration = TimeSpan.FromSeconds(5);

cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
cmdGrow.BeginAnimation(Button.HeightProperty, heightAnimation);
```

In this example, the two animations are not synchronized. That means the width and height won't grow at exactly the same intervals. (Typically, you'll see the button grow wider and then grow taller just after.) You can overcome this limitation by creating animations that are bound to the same timeline. You'll learn this technique later in this chapter, when you consider storyboards.

Animation Lifetime

Technically, WPF animations are *temporary*, which means they don't change the value of the underlying property. While an animation is active, it simply overrides the property value. This is because of the way that dependency properties work (as described in Chapter 4), and it's an often overlooked detail that can cause significant confusion.

A one-way animation (such as the button-growing animation) remains active after it finishes running. That's because the animation needs to hold the button's width at the new size. This can lead to an unusual problem—namely, if you try to modify the value of the property by using code after the animation has

completed, your code will appear to have no effect. That's because your code simply assigns a new local value to the property, but the animated value still takes precedence.

You can solve this problem in several ways, depending on what you're trying to accomplish:

- Create an animation that resets your element to its original state. You do this by not setting the To property. For example, the button-shrinking animation reduces the width of the button to its last set size, after which you can change it in your code.
- Create a reversible animation. You do this by setting the AutoReverse property to true. For example, when the button-growing animation finishes widening the button, it will play out the animation in reverse, returning it to its original width. The total duration of your animation will be doubled.
- Change the FillBehavior property. Ordinarily, FillBehavior is set to HoldEnd, which means that when an animation ends, it continues to apply its final value to the target property. If you change FillBehavior to Stop, as soon as the animation ends, the property reverts to its original value.
- Remove the animation object when the animation is complete by handling the Completed event of the animation object.

The first three options change the behavior of your animation. One way or another, they return the animated property to its original value. If this isn't what you want, you need to use the last option.

First, before you launch the animation, attach an event handler that reacts when the animation finishes:

```
widthAnimation.Completed += animation_Completed;
```

■ **Note** The Completed event is a normal .NET event that takes an ordinary EventArgs object with no additional information. It's not a routed event.

When the Completed event fires, you can render the animation inactive by calling the BeginAnimation() method. You simply need to specify the property and pass in a null reference for the animation object:

```
cmdGrow.BeginAnimation(Button.WidthProperty, null);
```

When you call BeginAnimation(), the property returns to the value it had before the animation started. If this isn't what you want, you can take note of the current value that's being applied by the animation, remove the animation, and then manually set the new property, like so:

```
double currentWidth = cmdGrow.Width;
cmdGrow.BeginAnimation(Button.WidthProperty, null);
cmdGrow.Width = currentWidth;
```

Keep in mind that this changes the local value of the property. That may affect how other animations work. For example, if you animate this button with an animation that doesn't specify the From property, it uses this newly applied value as a starting point. In most cases, this is the behavior you want.

The Timeline Class

As you've seen, every animation revolves around a few key properties. You've seen several of these properties: From and To (which are provided in animation classes that use interpolation) and Duration and FillBehavior (which are provided in all animation classes). Before going any further, it's worth taking a closer look at the properties you have to work with.

Figure 15-2 shows the inheritance hierarchy of the WPF animation types. It includes all the base classes, but it leaves out the full 42 animation types (and the corresponding *TypeName*AnimationBase classes).

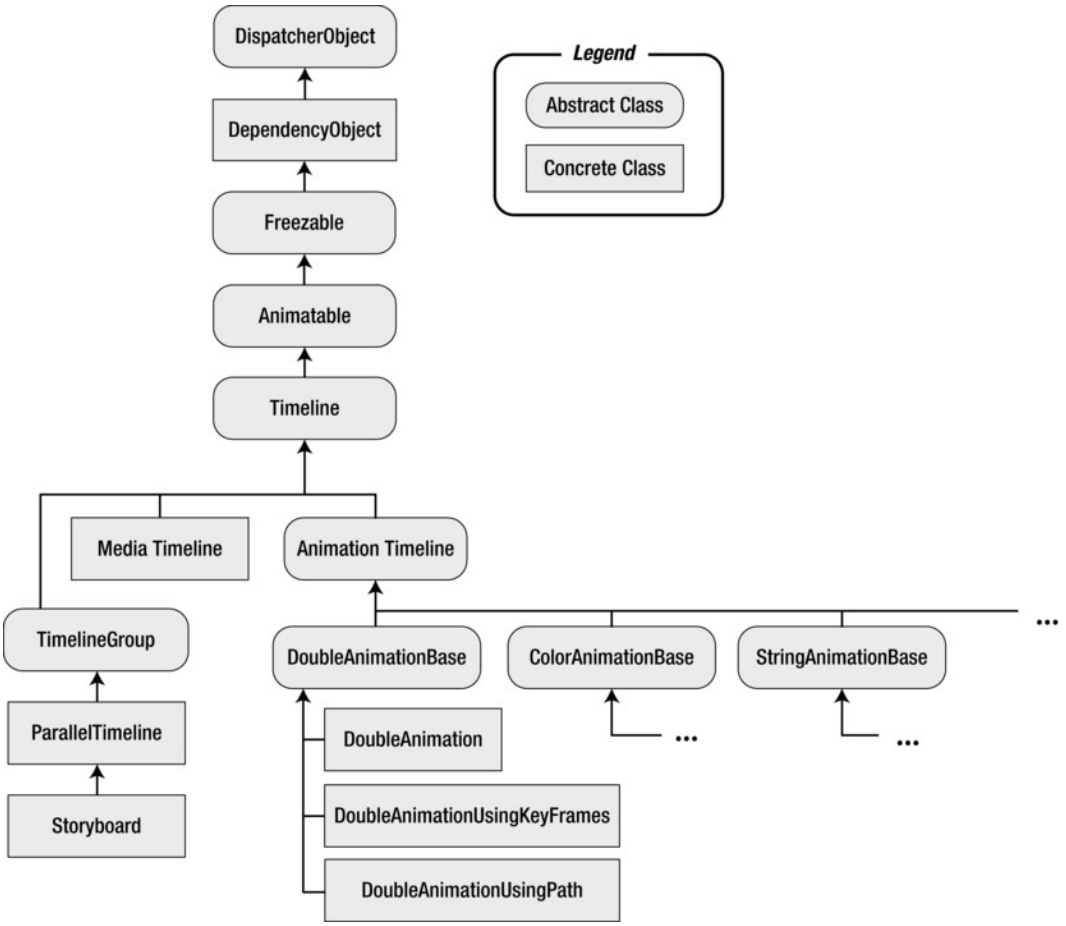

Figure 15-2. *The animation class hierarchy*

The class hierarchy includes three main branches that derive from the abstract Timeline class. MediaTimeline is used when playing audio or video files—it's described in Chapter 26. AnimationTimeline is used for the property-based animation system you've considered so far. And TimelineGroup allows you to synchronize timelines and control their playback. It's described later in this chapter in the "Synchronized Animations" section, when you tackle storyboards.

The first useful members appear in the Timeline class, which defines the Duration property you've already considered and a few more. Table 15-1 lists its properties.

Table 15-1. *Timeline Properties*

Name	Description
BeginTime	Sets a delay that will be added before the animation starts (as a TimeSpan). This delay is added to the total time, so a 5-second animation with a 5-second delay takes 10 seconds. BeginTime is useful when synchronizing different animations that start at the same time but should apply their effects in sequence.
Duration	Sets the length of time the animation runs, from start to finish, as a Duration object.
SpeedRatio	Increases or decreases the speed of the animation. Ordinarily, SpeedRatio is 1. If you increase it, the animation completes more quickly (for example, a SpeedRatio of 5 completes five times faster). If you decrease it, the animation is slowed down (for example, a SpeedRatio of 0.5 takes twice as long). You can change the Duration of your animation for an equivalent result. The SpeedRatio is not taken into account when applying the BeginTime delay.
AccelerationRatio and DecelerationRatio	Makes an animation nonlinear, so it starts off slow and then speeds up (by increasing the AccelerationRatio) or slows down at the end (by increasing the DecelerationRatio). Both values are set from 0 to 1 and begin at 0. Furthermore, the total of both values cannot exceed 1.
AutoReverse	If true, the animation will play out in reverse after it's complete, reverting to the original value. This also doubles the time the animation takes. If you've increased the SpeedRatio, it applies to both the initial playback of the animation and the reversal. The BeginTime applies only to the very beginning of the animation—it doesn't delay the reversal.
FillBehavior	Determines what happens when the animation ends. Usually, it keeps the property fixed at the ending value (FillBehavior.HoldEnd), but you can also choose to return it to its original value (FillBehavior.Stop).
RepeatBehavior	Allows you to repeat an animation a specific number of times or for a specific time interval. The RepeatBehavior object that you use to set this property determines the exact behavior.

Although BeginTime, Duration, SpeedRatio, and AutoReverse are all fairly straightforward, some of the other properties warrant closer examination. The following sections delve into AccelerationRatio, DecelerationRatio, and RepeatBehavior.

AccelerationRatio and DecelerationRatio

AccelerationRatio and DecelerationRatio allow you to compress part of the timeline so it passes by more quickly. The rest of the timeline is stretched to compensate so that the total time is unchanged.

Both of these properties represent a percentage value. For example, an AccelerationRatio of 0.3 indicates that you want to spend the first 30 percent of the duration of the animation accelerating. For example, in a 10-second animation, the first 3 seconds would be taken up with acceleration, and the remaining 7 seconds would pass at a consistent speed. (Obviously, the speed in the last 7 seconds is faster than the speed of a nonaccelerated animation, because the animation needs to make up for the slow start.) If you set AccelerationRatio to 0.3 and DecelerationRatio to 0.3, acceleration takes place for the first 3 seconds, the middle 4 seconds are at a fixed maximum speed, and deceleration takes place for the last 3 seconds. Viewed this way, it's obvious that the total of AccelerationRatio and DecelerationRatio can't top 1,

because that would require more than 100 percent of the available time to perform the requested acceleration and deceleration. Of course, you could set AccelerationRatio to 1 (in which case the animation speeds up from start to finish) or DecelerationRatio to 1 (in which case the animation slows down from start to finish).

Animations that accelerate and decelerate are often used to give a more natural appearance. However, AccelerationRatio and DecelerationRatio give you only relatively crude control. For example, they don't let you vary the acceleration or set it specifically. If you want to have an animation that uses varying degrees of acceleration, you'll need to define a series of animations, one after the other, and set the AccelerationRatio and DecelerationRatio property of each one, or you'll need to use a key-frame animation with key spline frames (as described in Chapter 16). Although this technique gives you plenty of flexibility, keeping track of all the details is a headache, and it's a perfect case for using a design tool to construct your animations.

RepeatBehavior

The RepeatBehavior property allows you to control how an animation is repeated. If you want to repeat it a fixed number of times, pass the appropriate number of times to the RepeatBehavior constructor. For example, this animation repeats twice:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
widthAnimation.RepeatBehavior = new RepeatBehavior(2);
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

When you run this animation, the button will increase in size (over 5 seconds), jump back to its original value, and then increase in size again (over 5 seconds), ending at the full width of the window. If you've set AutoReverse to true, the behavior is slightly different—the entire animation is completed forward and backward (meaning the button expands and then shrinks), and *then* it's repeated again.

■ **Note** Animations that use interpolation provide an IsCumulative property, which tells WPF how to deal with each repetition. If IsCumulative is true, the animation isn't repeated from start to finish. Instead, each subsequent animation adds to the previous one. For example, if you use IsCumulative with the animation shown earlier, the button will expand twice as wide over twice as much time. To put it another way, the first iteration is treated normally, but every repetition after that is treated as though you set IsAdditive to true.

Rather than using RepeatBehavior to set a repeat count, you can use it to set a repeat *interval*. To do so, simply pass a TimeSpan to the RepeatBehavior constructor. For example, the following animation repeats itself for 13 seconds:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
widthAnimation.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(13));
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

In this example, the Duration property specifies that the entire animation takes 5 seconds. As a result, the RepeatBehavior of 13 seconds will trigger two repeats and then leave the button halfway through a third repeat (at the 3-second mark).

■ **Tip** You can use RepeatBehavior to perform just part of an animation. To do so, use a fractional number of repetitions, or use a TimeSpan that's less than the duration.

Finally, you can cause an animation to repeat itself endlessly with the RepeatBehavior.Forever value:

```
widthAnimation.RepeatBehavior = RepeatBehavior.Forever;
```

Storyboards

As you've seen, WPF animations are represented by a group of animation classes. You set the relevant information, such as the starting value, ending value, and duration, using a handful of properties. This obviously makes them a great fit for XAML. What's less clear is how you wire an animation up to a particular element and property and how you trigger it at the right time.

It turns out that two ingredients are at work in any declarative animation:

A storyboard: It's the XAML equivalent of the BeginAnimation() method. It allows you to direct an animation to the right element and property.

An event trigger: It responds to a property change or event (such as the Click event of a button) and controls the storyboard. For example, to start an animation, the event trigger must *begin* the storyboard.

You'll learn how both pieces work in the following sections.

The Storyboard

A storyboard is an enhanced timeline. You can use it to group multiple animations as well as to control the playback of animation—pausing it, stopping it, and changing its position. However, the most basic feature provided by the Storyboard class is its ability to point to a specific property and specific element by using the TargetProperty and TargetName properties. In other words, the storyboard bridges the gap between your animation and the property you want to animate.

Here's how you might define a storyboard that manages a DoubleAnimation:

```
<Storyboard TargetName="cmdGrow" TargetProperty="Width">
  <DoubleAnimation From="160" To="300" Duration="0:0:5"></DoubleAnimation>
</Storyboard>
```

Both TargetName and TargetProperty are attached properties. That means you can apply them directly to the animation, as shown here:

```
<Storyboard>
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   From="160" To="300" Duration="0:0:5"></DoubleAnimation>
</Storyboard>
```

This syntax is more common, because it allows you to put several animations in the same storyboard but allow each animation to act on a different element and property.

Defining a storyboard is the first step to creating an animation. To actually put this storyboard into action, you need an event trigger.

Event Triggers

You first learned about event triggers in Chapter 11, when you considered styles. Styles give you one way to attach an event trigger to an element. However, you can define an event trigger in four places:

- In a style (the Styles.Triggers collection)
- In a data template (the DataTemplate.Triggers collection)
- In a control template (the ControlTemplate.Triggers collection)
- In an element directly (the FrameworkElement.Triggers collection)

When creating an event trigger, you need to indicate the routed event that starts the trigger and the action (or actions) that are performed by the trigger. With animations, the most common action is BeginStoryboard, which is equivalent to calling BeginAnimation().

The following example uses the Triggers collection of a button to attach an animation to the Click event. When the button is clicked, it grows.

```
<Button Padding="10" Name="cmdGrow" Height="40" Width="160"
 HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="Width"
             To="300" Duration="0:0:5"></DoubleAnimation>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>

  <Button.Content>
    Click and Make Me Grow
  </Button.Content>
</Button>
```

■ **Tip** To create an animation that fires when the window first loads, add an event trigger in the Window.Triggers collection that responds to the Window.Loaded event.

The Storyboard.TargetProperty property identifies the property you want to change (in this case, Width). If you don't supply a class name, the storyboard uses the parent element, which is the button you want to expand. If you want to set an attached property (for example, Canvas.Left or Canvas.Top), you need to wrap the entire property in brackets, like this:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Left)" ... />
```

The Storyboard.TargetName property isn't required in this example. When you leave it out, the storyboard uses the parent element, which is the button.

■ **Note** All an event trigger is able to do is launch *actions*. All actions are represented by classes that derive from System.Windows.TriggerAction. Currently, WPF includes a very small set of actions that are designed for interacting with a storyboard and controlling media playback.

There's one difference between the declarative approach shown here and the code-only approach demonstrated earlier. Namely, the To value is hard-coded at 300 units, rather than set relative to the size of the containing window. If you wanted to use the window width, you'd need to use a data-binding expression, like so:

```
<DoubleAnimation Storyboard.TargetProperty="Width"
 To="{Binding ElementName=window,Path=Width}" Duration="0:0:5">
</DoubleAnimation>
```

This still doesn't produce exactly the result you need. Here, the button grows from its current size to the full width of the window. The code-only approach enlarges the button to 30 units less than the full size, using a trivial calculation. Unfortunately, XAML doesn't support inline calculations. One solution is to build an IValueConverter that does the work for you. Fortunately, this odd trick is easy to implement (and many developers have). You can find one example at `http://tinyurl.com/y9lglyu`, or check out the downloadable examples for this chapter.

■ **Note** Another option is to create a custom dependency property in your window class that performs the calculation. You can then bind your animation to the custom dependency property. For more information about creating dependency properties, see Chapter 4.

You can now duplicate all the examples you've seen so far by creating triggers and storyboards and setting the appropriate properties of the DoubleAnimation object.

Attaching Triggers with a Style

The FrameworkElement.Triggers collection is a bit of an oddity. It supports only event triggers. The other trigger collections (Styles.Triggers, DataTemplate.Triggers, and ControlTemplate.Triggers) are more capable. They support the three basic types of WPF triggers: property triggers, data triggers, and event triggers.

■ **Note** There's no technical reason why the FrameworkElement.Triggers collection shouldn't support additional trigger types, but this functionality wasn't implemented in time for the first version of WPF.

Using an event trigger is the most common way to attach an animation. However, it's not your only option. If you're using the Triggers collection in a style, data template, or control template, you can also create a property trigger that reacts when a property value changes. For example, here's a style that duplicates the example shown earlier. It triggers a storyboard when IsPressed is true:

```
<Window.Resources>
  <Style x:Key="GrowButtonStyle">
    <Style.Triggers>
      <Trigger Property="Button.IsPressed" Value="True">
        <Trigger.EnterActions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation Storyboard.TargetProperty="Width"
               To="250" Duration="0:0:5"></DoubleAnimation>
            </Storyboard>
          </BeginStoryboard>
        </Trigger.EnterActions>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

You can attach actions to a property trigger in two ways. You can use Trigger.EnterActions to set actions that will be performed when the property changes to the value you specify (in the previous example, when IsPressed becomes true) and use Trigger.ExitActions to set actions that will be performed when the property changes back (when the value of IsPressed returns False false). This is a handy way to wrap together a pair of complementary animations.

Here's the button that uses the style shown earlier:

```
<Button Padding="10" Name="cmdGrow" Height="40" Width="160"
 Style="{StaticResource GrowButtonStyle}"
 HorizontalAlignment="Center" VerticalAlignment="Center">
 Click and Make Me Grow
</Button>
```

Remember, you don't need to use property triggers in a style. You can also use event triggers, as you saw in the previous section. Finally, you don't need to define a style separately from the button that uses it (you can set the Button.Style property with an inline style), but this two-part separation is more common, and it gives you the flexibility to apply the same animation to multiple elements.

■ **Note** Triggers are also handy when you fuse them into a control template, which allows you to add visual pizzazz to a standard WPF control. Chapter 17 shows numerous examples of control templates that use animations, including a ListBox that animates its child items with triggers.

Overlapping Animations

The storyboard gives you the ability to change the way you deal with animations that overlap—in other words, when a second animation is applied to a property that is already being animated. You do this using the BeginStoryboard.HandoffBehavior property.

Ordinarily, when two animations overlap, the second animation overrides the first one immediately. This behavior is known as *snapshot-and-replace* (and represented by the SnapshotAndReplace value in the HandoffBehavior enumeration). When the second animation starts, it takes a snapshot of the property as it currently is (based on the first animation), stops the animation, and replaces it with the new animation.

The only other HandoffBehavior option is Compose, which fused the second animation into the first animation's timeline. For example, consider a revised version of the ListBox example that uses HandoffBehavior.Compose when shrinking the button:

```
<EventTrigger RoutedEvent="ListBoxItem.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard HandoffBehavior="Compose">
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="FontSize"
         BeginTime="0:0:0.5" Duration="0:0:0.2"></DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

Now, if you move the mouse onto a ListBoxItem and off it, you'll see a different behavior. When you move the mouse off the item, it will continue expanding, which will be clearly visible until the second animation reaches its begin time delay of 0.5 seconds. Then, the second animation will shrink the button. Without the Compose behavior, the button would simply wait, fixed at its current size, for the 0.5-second time interval before the second animation kicks in.

Using a HandoffBehavior of compose requires more overhead. That's because the clock that's used to run the original animation won't be released when the second animation starts. Instead, it will stay alive until the ListBoxItem is garbage collected or a new animation is used on the same property.

■ **Tip** If performance becomes an issue, the WPF team recommends that you manually release the animation clock for your animations as soon as they are complete (rather than waiting for the garbage collector to find them). To do this, you need to handle an event such as Storyboard.Completed. Then, call BeginAnimation() on the element that has just finished its animation, supplying the appropriate property and a null reference in place of an animation.

Synchronized Animations

The Storyboard class derives indirectly from TimelineGroup, which gives it the ability to hold more than one animation. Best of all, these animations are managed as one group—meaning they're started at the same time.

To see an example, consider the following storyboard. It starts two animations, one that acts on the Width property of a button and the other that acts on the Height property. Because the animations are grouped into one storyboard, they increment the button's dimensions in unison, which gives a more synchronized effect than simply calling BeginAnimation() multiple times in your code.

```
<EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="Width"
         To="300" Duration="0:0:5"></DoubleAnimation>
        <DoubleAnimation Storyboard.TargetProperty="Height"
         To="300" Duration="0:0:5"></DoubleAnimation>
      </Storyboard>
```

```
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

In this example, both animations have the same duration, but this isn't a requirement. The only consideration with animations that end at different times is their FillBehavior. If an animation's FillBehavior property is set to HoldEnd, it holds the value until all the animations in the storyboard are completed. If the storyboard's FillBehavior property is HoldEnd, the final animated values are held indefinitely (until a new animation replaces this one or until you manually remove the animation).

It's at this point that the Timeline properties you learned about in Table 15-1 start to become particularly useful. For example, you can use SpeedRatio to make one animation in a storyboard run faster than the other. Or you can use BeginTime to offset one animation relative to another so that it starts at a specific point.

■ **Note** Because Storyboard derives from Timeline, you can use all the properties that were described in Table 15-1 to configure its speed, use acceleration or deceleration, introduce a delay time, and so on. These properties will affect all the contained animations, and they're cumulative. For example, if you set the Storyboard.SpeedRatio to 2 and the DoubleAnimation.SpeedRatio to 2, that animation will run four times faster than usual.

Controlling Playback

So far, you've been using one action in your event triggers—the BeginStoryboard action that launches an animation. However, you can use several other actions to control a storyboard after it's created. These actions, which derive from the ControllableStoryboardAction class, are listed in Table 15-2.

Table 15-2. *Action Classes for Controlling a Storyboard*

Name	Description
PauseStoryboard	Stops playback of an animation and keeps it at the current position.
ResumeStoryboard	Resumes playback of a paused animation.
StopStoryboard	Stops playback of an animation and resets the animation clock to the beginning.
SeekStoryboard	Jumps to a specific position in an animation's timeline. If animation is currently playing, it continues playback from the new position. If the animation is currently paused, it remains paused.
SetStoryboardSpeedRatio	Changes the SpeedRatio of the entire storyboard (rather than just one animation inside).
SkipStoryboardToFill	Moves the storyboard to the end of its timeline. Technically, this period is known as the *fill region*. For a standard animation, with FillBehavior set to HoldEnd, the animation continues to hold the final value.
RemoveStoryboard	Removes a storyboard, halting any in-progress animation and returning the property to its original, last-set value. This has the same effect as calling BeginAnimation() on the appropriate element with a null animation object.

■ **Note** Stopping an animation is not equivalent to completing an animation (unless FillBehavior is set to Stop). That's because even when an animation reaches the end of its timeline, it continues to apply its final value. Similarly, when an animation is paused, it continues to apply the most recent intermediary value. However, when an animation is stopped, it no longer applies any value, and the property reverts to its preanimation value.

There's an undocumented stumbling block to using these actions. For them to work successfully, you must define all the triggers in one Triggers collection. If you place the BeginStoryboard action in a different trigger collection than the PauseStoryboard action, the PauseStoryboard action won't work. To see the design you need to use, it helps to consider an example.

Consider the window shown in Figure 15-3. It superimposes two Image elements in exactly the same position, using a grid. Initially, only the topmost image—a day scene of a Toronto city landmark—is visible. But as the animation runs, the opacity is reduced from 1 to 0, eventually allowing the night scene to show through completely. The effect is as if the image is changing from day to night, like a sequence of time-lapse photography.

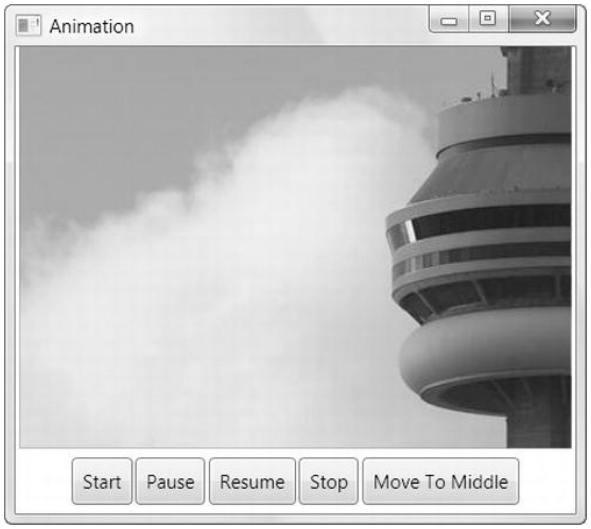

Figure 15-3. *A controllable animation*

Here's the markup that defines the Grid with its two images:

```
<Grid>
  <Image Source="night.jpg"></Image>
  <Image Source="day.jpg" Name="imgDay"></Image>
</Grid>
```

and here's the animation that fades from one to the other:

```
<DoubleAnimation
  Storyboard.TargetName="imgDay" Storyboard.TargetProperty="Opacity"
  From="1" To="0" Duration="0:0:10">
</DoubleAnimation>
```

To make this example more interesting, it includes several buttons at the bottom that allow you to control the playback of this animation. Using these buttons, you can perform the typical media player actions, such as pausing, resuming, and stopping. (You could add other buttons to change the speed ratio and seek out specific times.)

Here's the markup that defines these buttons:

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="5">
  <Button Name="cmdStart">Start</Button>
  <Button Name="cmdPause">Pause</Button>
  <Button Name="cmdResume">Resume</Button>
  <Button Name="cmdStop">Stop</Button>
  <Button Name="cmdMiddle">Move To Middle</Button>
</StackPanel>
```

Ordinarily, you might choose to place the event trigger in the Triggers collection of each individual button. However, as explained earlier, that doesn't work for animations. The easiest solution is to define all the event triggers in one place, such as the Triggers collection of a containing element, and wire them up using the EventTrigger.SourceName property. As long as the SourceName matches the Name property you've given the button, the trigger will be applied to the appropriate button.

In this example, you could use the Triggers collection of the StackPanel that holds the buttons. However, it's often easier to use the Triggers collection of the top-level element, which is the window in this case. That way, you can move your buttons to different places in your user interface without disabling their functionality.

```
<Window.Triggers>
  <EventTrigger SourceName="cmdStart" RoutedEvent="Button.Click">
    <BeginStoryboard Name="fadeStoryboardBegin">
      <Storyboard>
        <DoubleAnimation
          Storyboard.TargetName="imgDay" Storyboard.TargetProperty="Opacity"
          From="1" To="0" Duration="0:0:10">
        </DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>

  <EventTrigger SourceName="cmdPause" RoutedEvent="Button.Click">
    <PauseStoryboard BeginStoryboardName="fadeStoryboardBegin"></PauseStoryboard>
  </EventTrigger>
  <EventTrigger SourceName="cmdResume" RoutedEvent="Button.Click">
    <ResumeStoryboard BeginStoryboardName="fadeStoryboardBegin"></ResumeStoryboard>
  </EventTrigger>
  <EventTrigger SourceName="cmdStop" RoutedEvent="Button.Click">
    <StopStoryboard BeginStoryboardName="fadeStoryboardBegin"></StopStoryboard>
  </EventTrigger>
  <EventTrigger SourceName="cmdMiddle" RoutedEvent="Button.Click">
    <SeekStoryboard BeginStoryboardName="fadeStoryboardBegin"
     Offset="0:0:5"></SeekStoryboard>
  </EventTrigger>
</Window.Triggers>
```

Notice that you must give a name to the BeginStoryboard action. (In this example, it's fadeStoryboardBegin). The other triggers specify this name in the BeginStoryboardName property to link up to the same storyboard.

You'll encounter one limitation when using storyboard actions. The properties they provide (such as SeekStoryboard.Offset and SetStoryboardSpeedRatio.SpeedRatio) are not dependency properties. That limits your ability to use data-binding expressions. For example, you can't automatically read the Slider.Value property and apply it to the SetStoryboardSpeedRatio.SpeedRatio action, because the SpeedRatio property doesn't accept a data-binding expression. You might think you could code around this problem by using the SpeedRatio property of the Storyboard object, but this won't work. When the animation starts, the SpeedRatio value is read and used to create an animation clock. If you change it after that point, the animation continues at its normal pace.

If you want to adjust the speed or position dynamically, the only solution is to use code. The Storyboard class exposes methods that provide the same functionality as the triggers described in Table 15-2, including Begin(), Pause(), Resume(), Seek(), Stop(), SkipToFill(), SetSpeedRatio(), and Remove().

To access the Storyboard object, you need to make sure you set its Name property in the markup:

```
<Storyboard Name="fadeStoryboard">
```

■ **Note** Don't confuse the name of the Storyboard object (which is required to use the storyboard in your code) with the name of the BeginStoryboard action (which is required to wire up other trigger actions that manipulate the storyboard). To prevent confusion, you may want to adopt a convention such as adding the word *Begin* to the end of the BeginStoryboard name.

Now you simply need to write the appropriate event handler and use the methods of the Storyboard object. (Remember, simply changing storyboard properties such as SpeedRatio won't have any effect. They simply configure the settings that will be used when the animation starts.)

Here's an event handler that reacts when you drag the thumb on a Slider. The code then takes the value of the slider (which ranges from 0 to 3) and uses it to apply a new speed ratio:

```
private void sldSpeed_ValueChanged(object sender, RoutedEventArgs e)
{
    fadeStoryboard.SetSpeedRatio(this, sldSpeed.Value);
}
```

Notice that the SetSpeedRatio() requires two arguments. The first argument is the top-level animation container (in this case, the current window). All the storyboard methods require this reference. The second argument is the new speed ratio.

THE WIPE EFFECT

The previous example provides a gradual transition between the two images you're using by varying the Opacity of the topmost image. Another common way to transition between images is to perform a "wipe" that unveils the new image on top of the existing one.

The trick to using this technique is to create an opacity mask for the topmost image. Here's an example:

```
<Image Source="day.jpg" Name="imgDay">
  <Image.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
```

```
      <GradientStop Offset="0" Color="Transparent" x:Name="transparentStop" />
      <GradientStop Offset="0" Color="Black" x:Name="visibleStop" />
    </LinearGradientBrush>
  </Image.OpacityMask>
</Image>
```

This opacity mask uses a gradient that defines two gradient stops, Black (where the image will be completely visible) and Transparent (where the image will be completely transparent). Initially, both stops are positioned at the left edge of the image. Because the visible stop is declared last, it takes precedence, and the image will be completely opaque. Notice that both stops are named so they can be easily accessed by your animation.

Next, you need to perform your animation on the offsets of the LinearGradientBrush. In this example, both offsets are moved from the left side to the right side, allowing the image underneath to appear. To make this example a bit fancier, the offsets don't occupy the same position while they move. Instead, the visible offset leads the way, followed by the transparent offset after a short delay of 0.2 seconds. This creates a blended fringe at the edge of the wipe while the animation is underway.

```
<Storyboard>
  <DoubleAnimation
    Storyboard.TargetName="visibleStop"
    Storyboard.TargetProperty="Offset"
    From="0" To="1.2" Duration="0:0:1.2" ></DoubleAnimation>
  <DoubleAnimation
    Storyboard.TargetName="transparentStop"
    Storyboard.TargetProperty="Offset" BeginTime="0:0:0.2"
    From="0" To="1" Duration="0:0:1" ></DoubleAnimation>
</Storyboard>
```

There's one odd detail here. The visible stop moves to 1.2 rather than simply 1, which denotes the right edge of the image. This ensures that both offsets move at the same speed, because the total distance each one must cover is proportional to the duration of its animation.

Wipes commonly work from left to right or top to bottom, but more-creative effects are possible by using different opacity masks. For example, you could use a DrawingBrush for your opacity mask and modify its geometry to let the content underneath show through in a tiled pattern. You'll see more examples that animate brushes in Chapter 16.

Monitoring Progress

The animation player shown in Figure 15-3 still lacks one feature that's common in most media players—the ability to determine your current position. To make it a bit fancier, you can add some text that shows the time offset and a progress bar that provides a visual indication of how far you are in the animation. Figure 15-4 shows a revised animation player with both details (along with the Slider for controlling speed that was explained in the previous section).

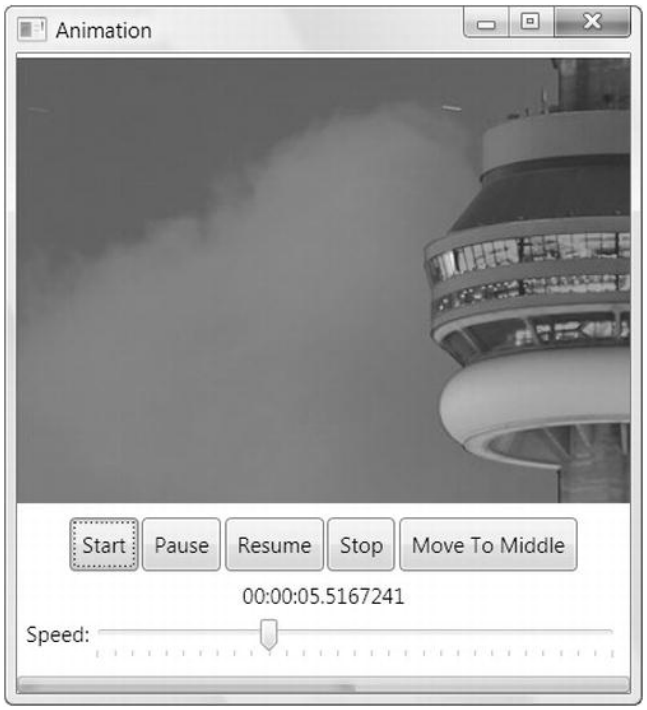

Figure 15-4. *Displaying position and progress in an animation*

Adding these details is fairly straightforward. First you need a TextBlock element to show the time and a ProgressBar control to show the graphical bar. You might assume you could set the TextBlock value and the ProgressBar content by using a data-binding expression, but this isn't possible. That's because the only way to retrieve the information about the current animation clock from the Storyboard is to use methods such as GetCurrentTime() and GetCurrentProgress(). There isn't any way to get the same information from properties.

The easiest solution is to react to one of the storyboard events listed in Table 15-3.

Table 15-3. *Storyboard Events*

Name	Description
Completed	The animation has reached its ending point.
CurrentGlobalSpeedInvalidated	The speed has changed, or the animation has been paused, resumed, stopped, or moved to a new position. This event also occurs when the animation clock reverses (at the end of a reversible animation) and when it accelerates or decelerates.
CurrentStateInvalidated	The animation has started or ended.
CurrentTimeInvalidated	The animation clock has moved forward an increment, changing the animation. This event also occurs when the animation starts, stops, or ends.
RemoveRequested	The animation is being removed. The animated property will subsequently return to its original value.

In this case, the event you need is CurrentTimeInvalidated, which fires every time the animation clock moves forward. (Typically, this will be 60 times per second, but if your code takes more time to execute, you may miss clock ticks.)

When the CurrentTimeInvalidated event fires, the sender is a Clock object (from the System.Windows.Media.Animation namespace). The Clock object allows you to retrieve the current time as a TimeSpan and the current progress as a value from 0 to 1.

Here's the code that updates the label and the progress bar:

```
private void storyboard_CurrentTimeInvalidated(object sender, EventArgs e)
{
    Clock storyboardClock = (Clock)sender;

    if (storyboardClock.CurrentProgress == null)
    {
        lblTime.Text = "[[ stopped ]]";
        progressBar.Value = 0;
    }
    else
    {
        lblTime.Text = storyboardClock.CurrentTime.ToString();
        progressBar.Value = (double)storyboardClock.CurrentProgress;
    }
}
```

■ **Tip** If you use the Clock.CurrentProgress property, you don't need to perform any calculation to determine the value for your progress bar. Instead, simply configure your progress bar with a minimum of 0 and a maximum of 1. That way, you can simply use the Clock.CurrentProgress to set the ProgressBar.Value, as in this example.

Animation Easing

One of the shortcomings of linear animation is that it often feels mechanical and unnatural. By comparison, sophisticated user interfaces have animated effects that model real-world systems. For example, they may use tactile push-buttons that jump back quickly when clicked but slow down as they come to rest, creating the illusion of true movement. Or they may use maximize and minimize effects as in the Windows operating system, where the speed at which the window grows or shrinks accelerates as the window nears its final size. These details are subtle, and you're not likely to notice them when they're implemented well. However, you'll almost certainly notice the clumsy feeling of less-refined animations that lack these finer points.

The secret to improving your animations and creating more-natural animations is to vary the rate of change. Instead of creating animations that change properties at a fixed, unchanging rate, you need to design animations that speed up or slow down along the way. WPF gives you several options. In the next chapter, you'll learn about frame-based animation and key-frame animation, two techniques that give you more-nuanced control over your animations (and require significantly more work). But the simplest way to make a more natural animation is to use a prebuilt *easing function*.

When using an easing function, you still define your animation normally by specifying the starting and ending property values. But in addition to these details, you add a ready-made mathematical function

that alters the progression of your animation, causing it to accelerate or decelerate at different points. This is the technique you'll study in the following sections.

Using an Easing Function

The best part about animation easing is that it requires much less work than other approaches such as frame-based animation and key frames. To use animation easing, you set the EasingFunction property of an animation object with an instance of an easing function class (a class that derives from EasingFunctionBase). You'll usually need to set a few properties on the easing function, and you may be forced to play around with different settings to get the effect you want, but you'll need no code and very little additional XAML.

For example, consider the two animations shown here, which act on a button. When the user moves the mouse over the button, a small snippet of code calls the growStoryboard animation into action, stretching the button to 400 units. When the user moves the mouse off the button, the buttons shrinks back to its normal size.

```
<Storyboard x:Name="growStoryboard">
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   To="400" Duration="0:0:1.5"></DoubleAnimation>
</Storyboard>

<Storyboard x:Name="revertStoryboard">
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   Duration="0:0:3"></DoubleAnimation>
</Storyboard>
```

Right now, the animations use linear interpolation, which means the growing and shrinking happen in a steady, mechanical way. For a more natural effect, you can add an easing function. The following example adds an easing function named ElasticEase. The end result is that the button springs beyond its full size, snaps back to a value that's somewhat less, swings back over its full size again (but a little less than before), snaps back a bit less, and so on, repeating its bouncing pattern as the movement diminishes. It gradually comes to rest ten oscillations later. The Oscillations property controls the number of bounces at the end. The ElasticEase class provides one other property that's not used in this example: Springiness. This higher this value, the more each subsequent oscillation dies down (the default value is 3).

```
<Storyboard x:Name="growStoryboard">
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   To="400" Duration="0:0:1.5">
    <DoubleAnimation.EasingFunction>
      <ElasticEase EasingMode="EaseOut" Oscillations="10"></ElasticEase>
    </DoubleAnimation.EasingFunction>
  </DoubleAnimation>
</Storyboard>
```

To really appreciate the difference between this markup and the earlier example that didn't use an easing function, you need to try this animation (or run the companion examples for this chapter). It's a remarkable change. With one line of XAML, a simple animation changes from amateurish to a slick effect that would feel at home in a professional application.

■ **Note** Because the EasingFunction property accepts a single easing function object, you can't combine different easing functions for the same animation.

Easing In and Easing Out

Before you consider the different easing functions, it's important to understand *when* an easing function is applied. Every easing function class derives from EasingFunctionBase and inherits a single property named EasingMode. This property has three possible values: EaseIn (which means the effect is applied to the beginning of the animation), EaseOut (which means it's applied to the end), and EaseInOut (which means it's applied at both the beginning and the end—the easing in takes place in the first half of the animation, and the easing out takes place in the second half).

In the previous example, the animation in the growStoryboard animation uses EaseOut mode. Thus, the sequence of gradually diminishing bounces takes place at the end of the animation. If you were to graph the changing button width as the animation progresses, you'd see something like the graph shown in Figure 15-5.

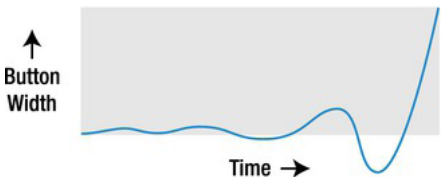

Figure 15-5. *Oscillating to a stop by using EaseOut with ElasticEase*

■ **Note** The duration of an animation doesn't change when you apply an easing function. In the case of the growStoryboard animation, the ElasticEase function doesn't just change the way the animation ends—it also makes the initial portion of the animation (when the button expands normally) run more quickly so that there's more time left for the oscillations at the end.

If you switch the ElasticEase function to use EaseIn mode, the bounces happen at the beginning of the animation. The button shrinks below its starting value a bit, expands a bit over, shrinks back a little more, and continues this pattern of gradually increasing oscillations until it finally breaks free and expands the rest of the way. (You use the ElasticEase.Oscillations property to control the number of bounces.) Figure 15-6 shows this very different pattern of movement.

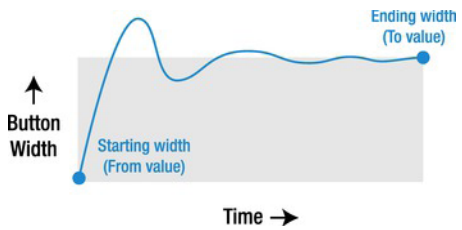

Figure 15-6. *Oscillating to a start by using EaseIn with ElasticEase*

Finally, EaseInOut creates a stranger effect, with oscillations that start the animation in its first half followed by oscillations that stop it in the second half. Figure 15-7 illustrates.

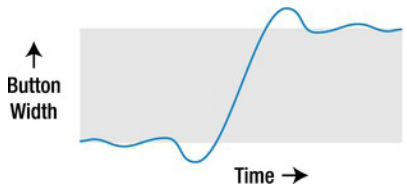

Figure 15-7. *Oscillating to a start and to a stop by using EaseInOut with ElasticEase*

Easing Function Classes

WPF has 11 easing functions, all of which are found in the familiar System.Windows.Media.Animation namespace. Table 15-4 describes them all and lists their important properties. Remember, every animation also provides the EasingMode property, which allows you to control whether it affects that animation as it starts (EaseIn), ends (EaseOut), or both (EaseInOut).

Table 15-4. *Easing Functions*

Name	Description	Properties
BackEase	When applied with EaseIn, pulls the animation back before starting it. When applied with EaseOut, this function allows the animation to overshoot slightly and then pulls it back.	Amplitude determines the amount of pullback or overshoot. The default value is 1, and you can decrease it (to any value greater than 0) to reduce the effect or increase it to amplify the effect.
ElasticEase	When applied with EaseOut, makes the animation overshoot its maximum and swing back and forth, gradually slowing. When applied with EaseIn, the animation swings back and forth around its starting value, gradually increasing.	Oscillations controls the number of times the animation swings back and forth (the default is 3), and Springiness controls how quickly the oscillations increase or diminish (the default is 3).
BounceEase	Performs an effect similar to ElasticEase, except the bounces never overshoot the initial or final values.	Bounces controls the number of times the animation bounces back (the default is 2), and Bounciness determines how quickly the bounces increase or diminish (the default is 2).

CircleEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a circular function.	None
CubicEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a function based on the cube of time. The effect is similar to CircleEase, but the acceleration is more gradual.	None
QuadraticEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a function based on the square of time. The effect is similar to CubicEase but even more gradual.	None
QuarticEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a function based on time to the power of 4. The effect is similar to CubicEase and QuadraticEase, but the acceleration is more pronounced.	None
QuinticEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a function based on time to the power of 5. The effect is similar to CubicEase, QuadraticEase, and QuinticEase, but the acceleration is more pronounced.	None
SineEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using a function that includes a sine calculation. The acceleration is very gradual and closer to linear interpolation than any of the other easing functions.	None
PowerEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using the power function $f(t) = t^p$. Depending on the value you use for the exponent p, you can duplicate the effect of the Cubic, QuadraticEase, QuarticEase, and QuinticEase functions.	Power sets the value of the exponent in the formula. Use 2 to duplicate QuadraticEase ($f(t) = t^2$), 3 for CubicEase ($f(t) = t^3$), 4 for QuarticEase ($f(t) = t^4$), and 5 for QuinticEase ($f(t) = t^5$), or choose something different. The default is 2.
ExponentialEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation by using the exponential function $f(t)=(e(at) - 1)/(e(a) - 1)$.	Exponent allows you to set the value of the exponent (2 is the default).

Many of the easing functions provide similar but subtly different results. To use animation easing successfully, you need to decide which easing function to use and how to configure it. Often this process requires a bit of trial-and-error experimentation. Two good resources can help you.

First, the WPF documentation charts example behavior for each easing function, showing how the animated value changes as time progresses. Reviewing these charts is a good way to develop a sense of what the easing function does. Figure 15-8 shows the charts for the most popular easing functions.

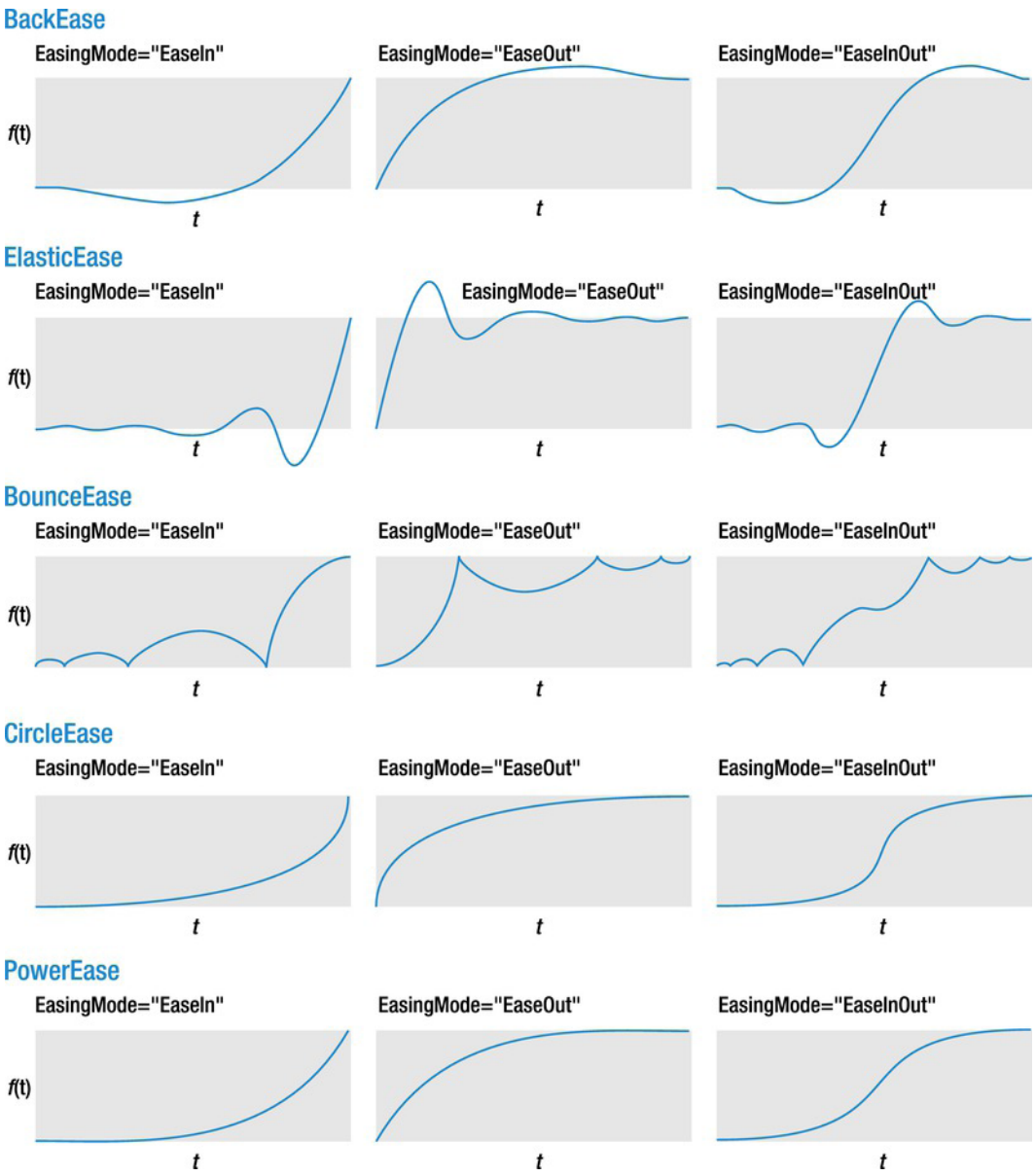

Figure 15-8. *The effect of different easing functions*

Second, Microsoft provides several sample applications that you can use to play with the easing functions and try different property values. One of the handiest is a Silverlight application that you can run in the browser by surfing to `http://tinyurl.com/animationeasing`. It allows you to observe the effect of any easing function on a falling square, and it shows the automatically generated XAML markup needed to duplicate the effect.

Creating a Custom Easing Function

You can create a custom easing effect by deriving your own class from EasingFunctionBase and overriding the EaseInCore() and CreateInstanceCore() methods. This is a fairly specialized technique, because most developers will be able to get the results they want by configuring the standard easing functions (or by using key spline animations, as described in the next chapter). However, if you do decide to create a custom easing function, you'll find that it's surprisingly easy.

Virtually all the logic you need to write runs in the EaseInCore() method. It accepts a normalized time value—essentially, a value from 0 to 1 that represents the progress of the animation. When the animation first begins, the normalized time is 0. It increases from that point on, until it reaches 1 at the end of the animation.

```
protected override double EaseInCore(double normalizedTime)
{ ... }
```

During an animation, WPF calls the EaseInCore() method each time it updates the animated value. The exact frequency depends on the animation's frame rate, but you can expect it to call EaseInCore() close to 60 times each second.

To perform easing, the EaseInCore() method takes the normalized time and adjusts it in some way. The adjusted value that EaseInCore() returns is then used to adjust the progress of the animation. For example, if EaseInCore() returns 0, the animation is returned to its starting point. If EaseInCore() returns 1, the animation jumps to its ending point. However, EaseInCore() isn't limited to this range—for example, it can return 1.5 to cause the animation to overrun itself by an additional 50 percent. (You've already seen this effect with easing functions such as ElasticEase.)

Here's a version of EaseInCore() that does nothing at all. It returns the normalized time, meaning the animation will unfold evenly, just as if there were no easing:

```
protected override double EaseInCore(double normalizedTime)
{
    return normalizedTime;
}
```

And here's a version of EaseInCore() that duplicates the CubicEase function, by cubing the normalized time. Because the normalized time is a fractional value, cubing it produces a smaller fraction. Thus, this method has the effect of initially slowing down the animation and causing it to accelerate as the normalized time (and its cubed value) approaches 1.

```
protected override double EaseInCore(double normalizedTime)
{
    return Math.Pow(normalizedTime, 3);
}
```

■ **Note** The easing you perform in the EaseInCore() method is what you'll get when you use an EasingMode of EaseIn. Interestingly, that's all the work you need to do, because WPF is intelligent enough to calculate complementary behavior for the EaseOut and EaseInOut settings.

Finally, here's a custom easing function that does something more interesting—it offsets the normalized value a random amount, causing a sporadic jittering effect. You can adjust the magnitude of the jitter (within a narrow range) by using the provide Jitter dependency property, which accepts a value from 0 to 2000.

```
public class RandomJitterEase : EasingFunctionBase
{
    // Store a random number generator.
    private Random rand = new Random();

    // Allow the amount of jitter to be configured.
    public static readonly DependencyProperty JitterProperty =
      DependencyProperty.Register("Jitter", typeof(int), typeof(RandomJitterEase),
      new UIPropertyMetadata(1000), new ValidateValueCallback(ValidateJitter));

    public int Jitter
    {
        get { return (int)GetValue(JitterProperty); }
        set { SetValue(JitterProperty, value); }
    }

    private static bool ValidateJitter(object value)
    {
        int jitterValue = (int)value;
        return ((jitterValue <= 2000) && (jitterValue >= 0));
    }

    // Perform the easing.
    protected override double EaseInCore(double normalizedTime)
    {
        // Make sure there's no jitter in the final value.
        if (normalizedTime == 1) return 1;

        // Offset the value by a random amount.
        return Math.Abs(normalizedTime -
          (double)rand.Next(0,10)/(2010 - Jitter));
    }

    // This required override simply provides a live instance of your
    // easing function.
    protected override Freezable CreateInstanceCore()
    {
        return new RandomJitterEase();
    }
}
```

■ **Tip** If you want to see the eased values that you're calculating as your animation runs, use the WriteLine() method of the System.Diagnostics.Debug class in the EaseInCore() method. This writes the value you supply to the Output window while you're debugging your application in Visual Studio.

Using this easing function is easy. First, map the appropriate namespace in your XAML:

```
<Window x:Class="Animation.CustomEasingFunction"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="CustomEasingFunction" Height="300" Width="600"
 xmlns:local="clr-namespace:Animation">
```

Then you can create a RandomJitterEase object in your markup, like this:

```
<DoubleAnimation
 Storyboard.TargetName="ellipse2" Storyboard.TargetProperty="(Canvas.Left)"
 To="500" Duration="0:0:10">
  <DoubleAnimation.EasingFunction>
    <local:RandomJitterEase EasingMode="EaseIn" Jitter="1000">
    </local:RandomJitterEase>
  </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

The online samples for this chapter feature an example that compares an animation with no easing (the movement of a small ellipse across a Canvas) to one that uses the RandomJitterEase.

Animation Performance

Often an animated user interface requires little more than creating and configuring the right animation and storyboard objects. But in other scenarios, particularly ones in which you have multiple animations taking place at the same time, you may need to pay more attention to performance. Certain effects are more likely to cause these issues—for example, those that involve video, large bitmaps, and multiple levels of transparency typically demand more from the computer's CPU. If they're not implemented carefully, they may run with notable jerkiness, or they may steal CPU time away from other applications that are running at the same time.

Fortunately, WPF has a few tricks that can help you. In the following sections, you'll learn to slow down the maximum frame rate and cache bitmaps on the computer's video card, two techniques that can lessen the load on the CPU.

Desired Frame Rate

As you learned earlier in this chapter, WPF attempts to keep animations running at 60 frames per second. This ensures smooth, fluid animations from start to finish. Of course, WPF might not be able to deliver on its intentions. If you have multiple complex animations running at once and the CPU or video card can't keep up, the overall frame rate may drop (in the best-case scenario), or it may jump to catch up (in the worst-case scenario).

Although it's rare to increase the frame rate, you may choose to *decrease* the frame rate. You might take this step for one of two reasons:

- Your animation looks good at a lower frame rate, so you don't want to waste the extra CPU cycles.
- Your application is running on a less powerful CPU or video card, and you know your complete animation won't be rendered as well at a high frame rate as it would at a lower rate.

■ **Note** Developers sometimes assume that WPF includes code that scales the frame rate down based on the video card hardware. It does not. Instead, WPF always attempts 60 frames per second, unless you tell it otherwise. To evaluate how your animations are performing and whether WPF is able to achieve 60 frames per second on a specific computer, you can use the Perforator tool, which is included as part of the Microsoft Windows SDK v7.0. For a download link, installation instructions, and documentation, see `http://tinyurl.com/9kzmv9s`.

Adjusting the frame rate is easy. You simply use the Timeline.DesiredFrameRate attached property on the storyboard that contains your animations. Here's an example that halves the frame rate:

```
<Storyboard Timeline.DesiredFrameRate="30">
```

Figure 15-9 shows a simple test application that animates a circle so that it arcs across a Canvas.

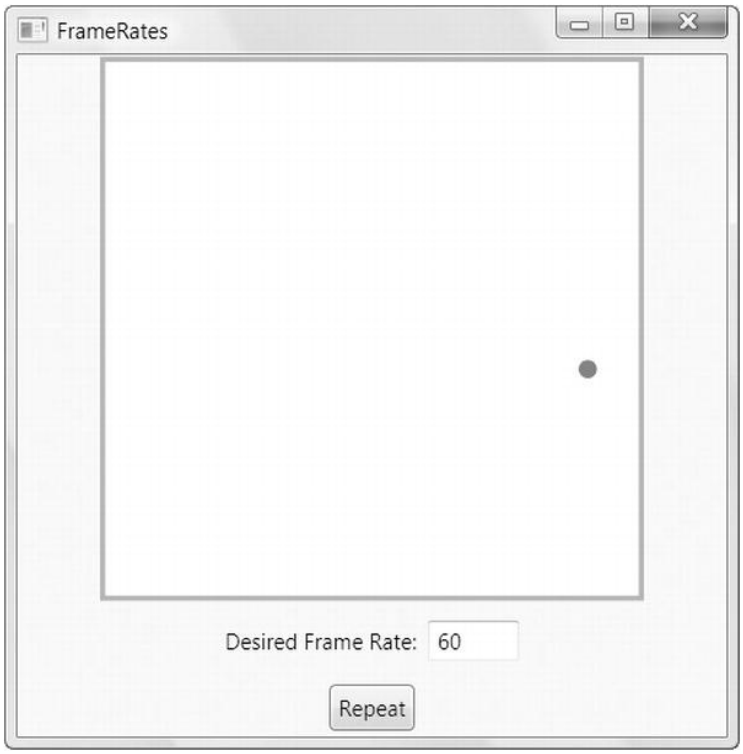

Figure 15-9. *Testing frame rates with a simple animation*

The application begins with an Ellipse object in a Canvas. The Canvas.ClipToBounds property is set to true so the edges of the circle won't leak over the edge of the Canvas into the rest of the window.

```
<Canvas ClipToBounds="True">
  <Ellipse Name="ellipse" Fill="Red" Width="10" Height="10"></Ellipse>
</Canvas>
```

To move the circle across the Canvas, two animations take place at once—one that updates the Canvas.Left property (moving it from left to right) and one that changes the Canvas.Top property (causing it to rise up and then fall back down). The Canvas.Top animation is reversible—after the circle reaches its highest point, it falls back down. The Canvas.Left animation is not, but it takes twice as long, so both animations move the circle simultaneously. The final trick is using the DecelerationRatio property on the Canvas.Top animation. That way, the circle rises more slowly as it reaches the summit, which creates a more realistic effect.

Here's the complete markup for the animation:

```
<Window.Resources>
  <BeginStoryboard x:Key="beginStoryboard">
    <Storyboard Timeline.DesiredFrameRate=
     "{Binding ElementName=txtFrameRate,Path=Text}">
      <DoubleAnimation Storyboard.TargetName="ellipse"
       Storyboard.TargetProperty="(Canvas.Left)"
       From="0" To="300" Duration="0:0:5">
      </DoubleAnimation>
      <DoubleAnimation Storyboard.TargetName="ellipse"
       Storyboard.TargetProperty="(Canvas.Top)"
       From="300" To="0" AutoReverse="True" Duration="0:0:2.5"
       DecelerationRatio="1">
      </DoubleAnimation>
    </Storyboard>
  </BeginStoryboard>
</Window.Resources>
```

Notice that the Canvas.Left and Canvas.Top properties are wrapped in brackets—this indicates that they aren't found on the target element (the ellipse) but are attached properties. You'll also see that the animation is defined in the Resources collection for the window. This allows the animation to be started in more than one way. In this example, the animation is started when the Repeat button is clicked and when the window is first loaded, using code like this:

```
<Window.Triggers>
  <EventTrigger RoutedEvent="Window.Loaded">
    <EventTrigger.Actions>
      <StaticResource ResourceKey="beginStoryboard"></StaticResource>
    </EventTrigger.Actions>
  </EventTrigger>
</Window.Triggers>
```

The real purpose of this example is to try different frame rates. To see the effect of a particular frame rate, you simply need to type the appropriate number in the text box and click Repeat. The animation is then triggered with the new frame rate (which it picks up through a data-binding expression), and you can watch the results. At lower frame rates, the ellipse won't appear to move evenly—instead, it will hop across the Canvas.

You can also adjust the Timeline.DesiredFrame property in code. For example, you may want to read the static RenderCapability.Tier to determine the level of video card support.

■ **Note** With a little bit of work, you can also create a helper class that lets you put the same logic into work in your XAML markup. You'll find one example at `http://tinyurl.com/yata5eu`, which demonstrates how you can lower the frame rate declaratively based on the tier.

Bitmap Caching

Bitmap caching tells WPF to take a bitmap image of your content as it currently is and copy that to the memory on your video card. From this point on, the video card can take charge of manipulating the bitmap and refreshing the display. This process is far faster than getting WPF to do all the work and communicate continuously with the video card.

In the right situation, bitmap caching improves the drawing performance of your application. But in the wrong situation, it wastes video card memory and actually *slows* performance. Thus, before you use bitmap caching, you need to make sure that it's truly suitable. Here are some guidelines:

- If the content you're painting needs to be redrawn frequently, bitmap caching may make sense. That's because each subsequent redraw will happen much faster. One example is using a BitmapCacheBrush to paint the surface of a shape, while some other animated objects float on top. Even though your shape isn't changing, different parts of it are being obscured or revealed, necessitating a redraw.
- If the content in your element changes often, bitmap caching probably doesn't make sense. That's because each time the visual changes, WPF needs to rerender the bitmap and send it to the video card cache, which takes time. This rule is a bit tricky, because certain changes won't invalidate the cache. Examples of safe operations include rotating and rescaling your element with a transform, clipping it, changing its opacity, or applying an effect. On the other hand, changing its content, layout, and formatting will force the bitmap to be rerendered.
- Cache the smallest amount of content possible. The larger the bitmap, the longer WPF takes to store the cached copy, and the more video card memory it requires. After the video card memory is exhausted, WPF will be forced to fall back on slower software rendering.

■ **Tip** A poor caching strategy can cause more performance problems that an application that isn't fully optimized. So don't apply caching unless you're sure you meet these guidelines. Also, use a profiling tool such as Perforator (`http://tinyurl.com/9kzmv9s`) to verify that your strategy is improving performance.

To get a better understanding, it helps to play with a simple example. Figure 15-10 shows a project that's included with the downloadable samples for this chapter. Here, an animation pushes a simple shape—a square—over a Canvas that contains a Path with a complex geometry. As the square moves over its surface, WPF is forced to recalculate the path and fill in the missing sections. This imposes a surprisingly heavy CPU load, and the animation may even begin to become choppy.

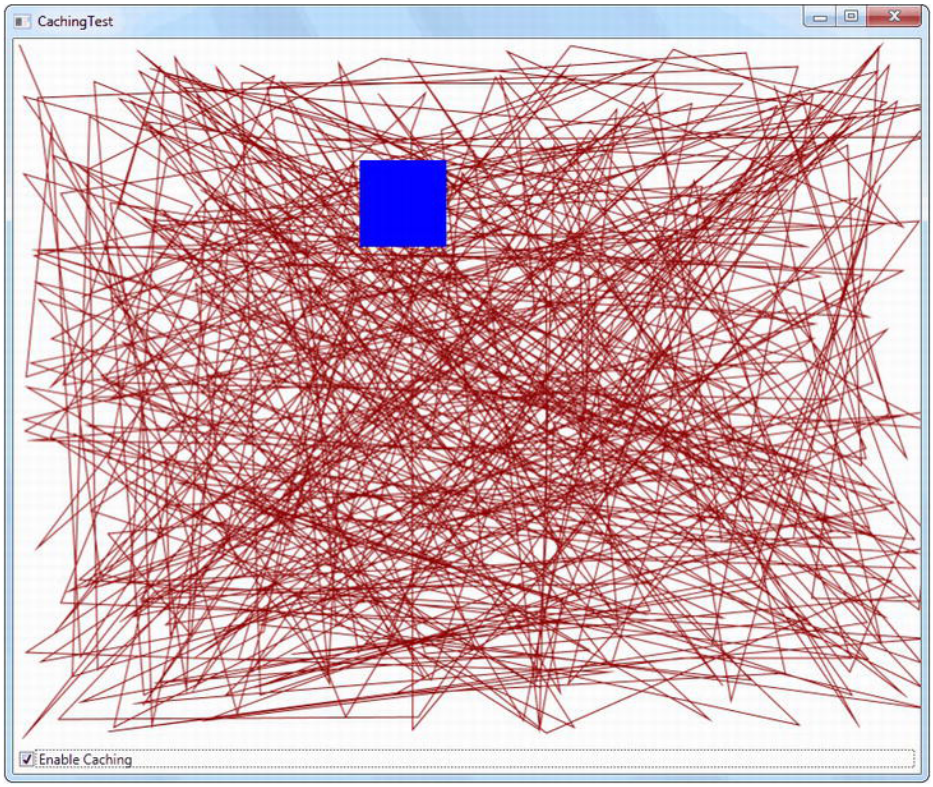

Figure 15-10. *Animating over a complex piece of vector art*

There are several ways to solve this problem. One option is to replace the background with a bitmap, which WPF can manage more efficiently. A more flexible option is to use bitmap caching, which preserves the background as a live, interactive element.

To switch on bitmap caching, you set the CacheMode property of the corresponding element to BitmapCache. Every element provides this property, which means you have a fine-grained ability to choose exactly which elements use this feature:

```
<Path CacheMode="BitmapCache" ...></Path>
```

■ **Note** If you cache an element that contains other elements, such as a layout container, all the elements will be cached in a single bitmap. Thus, you need to be extremely careful about adding caching to something like a Canvas—do it only if the Canvas is small and its content will not change.

With this single, simple change, you'll see an immediate difference. First, the window will take slightly longer to appear. But the animation will run more smoothly, and the CPU load will decrease dramatically. Check it out in Windows Task Manager—it's not unusual to see it drop from close to 100 percent to less than 20 percent.

Ordinarily, when you enable bitmap caching, WPF takes a snapshot of the element at its current size and copies that bitmap to the video card. This can become a problem if you then use a ScaleTransform to make the element bigger. In this situation, you'll be enlarging the cached bitmap, not the actual element, which will cause it to grow fuzzy and pixelated as it grows.

For example, imagine a revised example in which a second simultaneous animation expands the Path to ten times its original size and then shrinks it back. To ensure good quality, you can cache a bitmap of the Path at five times its current size:

```
<Path ...>
  <Path.CacheMode>
    <BitmapCache RenderAtScale="5"></BitmapCache>
  </Path.CacheMode>
</Path>
```

This resolves the pixelation problem. The cached bitmap is still smaller than the maximum animated size of the Path (which reaches ten times its original size), but the video card is able to double the size of the bitmap from five to ten times its original size without any obvious scaling artifacts. More important, this still keeps your application from using an excessive amount of video memory.

The Last Word

In this chapter, you explored WPF's animation support in detail. You learned about the basic animation classes and the concept of linear interpolation. You also saw how to control the playback of one or more animations with a storyboard and how to create more-natural effects with animation easing.

Now that you've mastered the basics, you can spend more time with the art of animation—deciding what properties to animate and how to modify them to get the effect you want. In the next chapter, you'll learn how to create a variety of effects by applying animations to transforms, brushes, and pixel shaders. You'll also learn to create key-frame animations that contain multiple segments and frame-based animations that break free from the standard property-based animation model. Finally, you'll see how to create and manage storyboards with code rather than XAML.