```
<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.

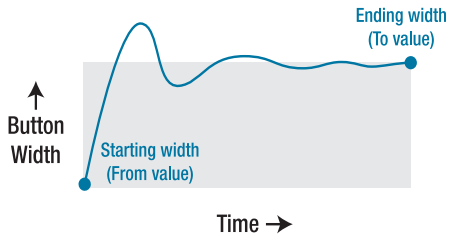

Figure 15-5. *Oscillating to a stop using EaseOut with ElasticEase*

■ **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.

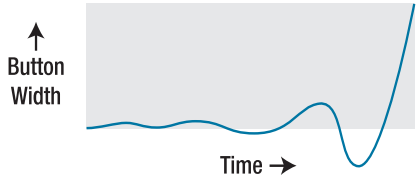

Figure 15-6. *Oscillating to a start using EaseIn with ElasticEase*

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.

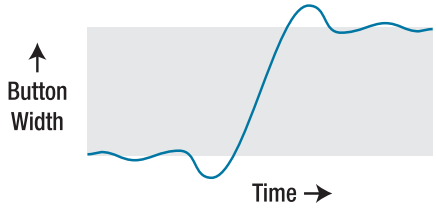

Figure 15-7. *Oscillating to a start and to a stop using EaseInOut with ElasticEase*

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 which 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 using a circular function.	None
CubicEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation 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 using a function based on the square of time. The effect is similar to CubicEase but even more gradual.	None

Name	Description	Properties
QuarticEase	Accelerates (with EaseIn) or decelerates (with EaseOut) the animation 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 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 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 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 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.

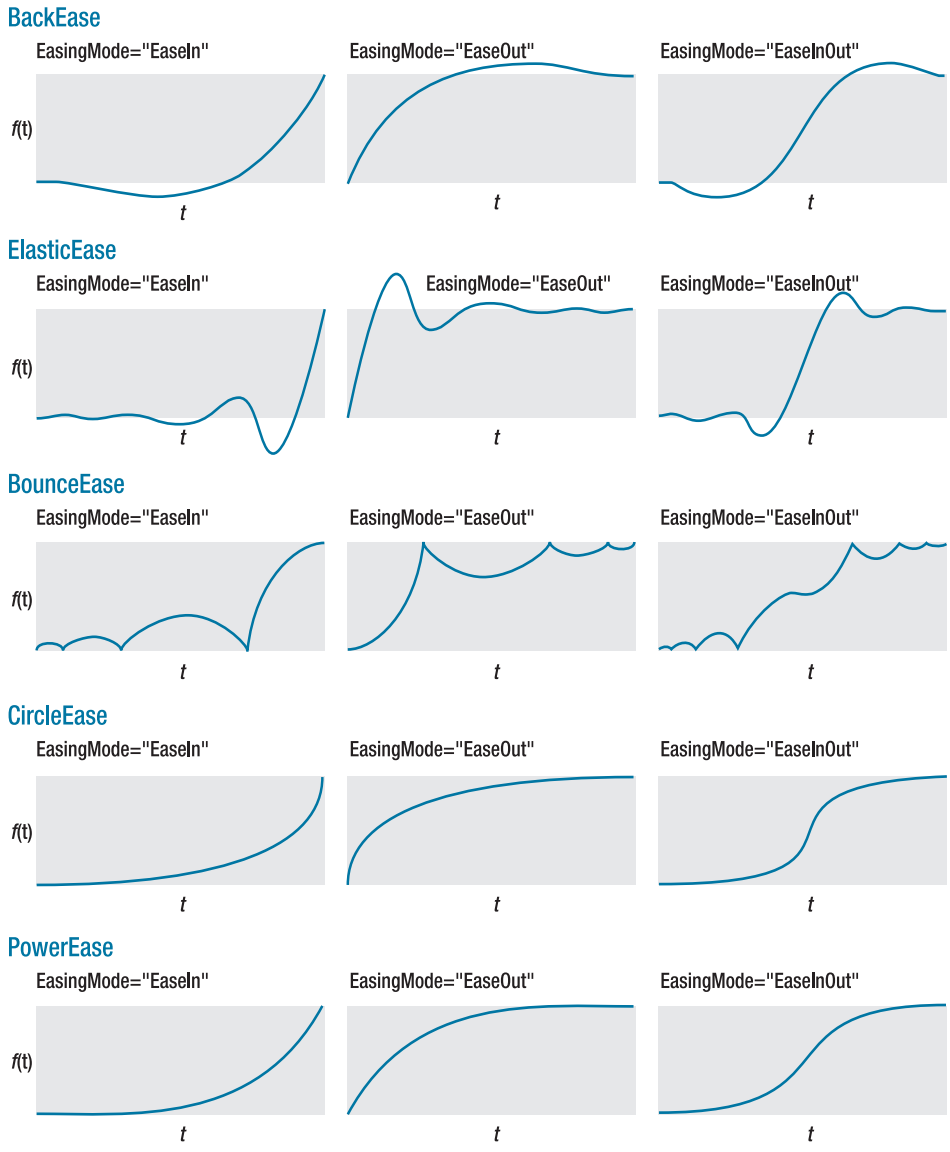

Figure 15-8. *The effect of different easing functions*

Second, Microsoft provides several sample applications that you can use to play with the different 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 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%. (You've already seen this effect with easing functions like 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) 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 that 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/yfqottg`.

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.

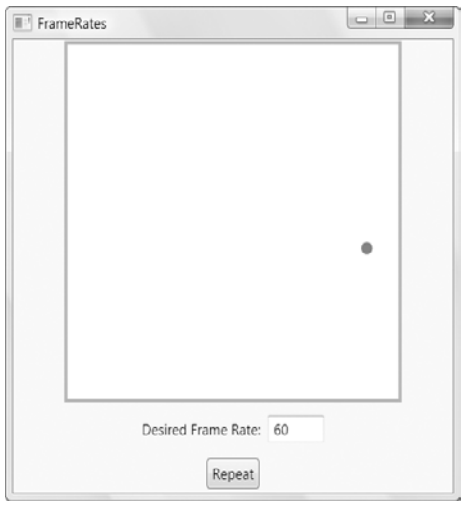

Figure 15-9. *Testing frame rates with a simple animation*

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—once 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 if you're using a BitmapCacheBrush to paint the surface of a shape, while some other animated objects float overtop. 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. Once 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 like Perforator (`http://tinyurl.com/yfqottg`) 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.

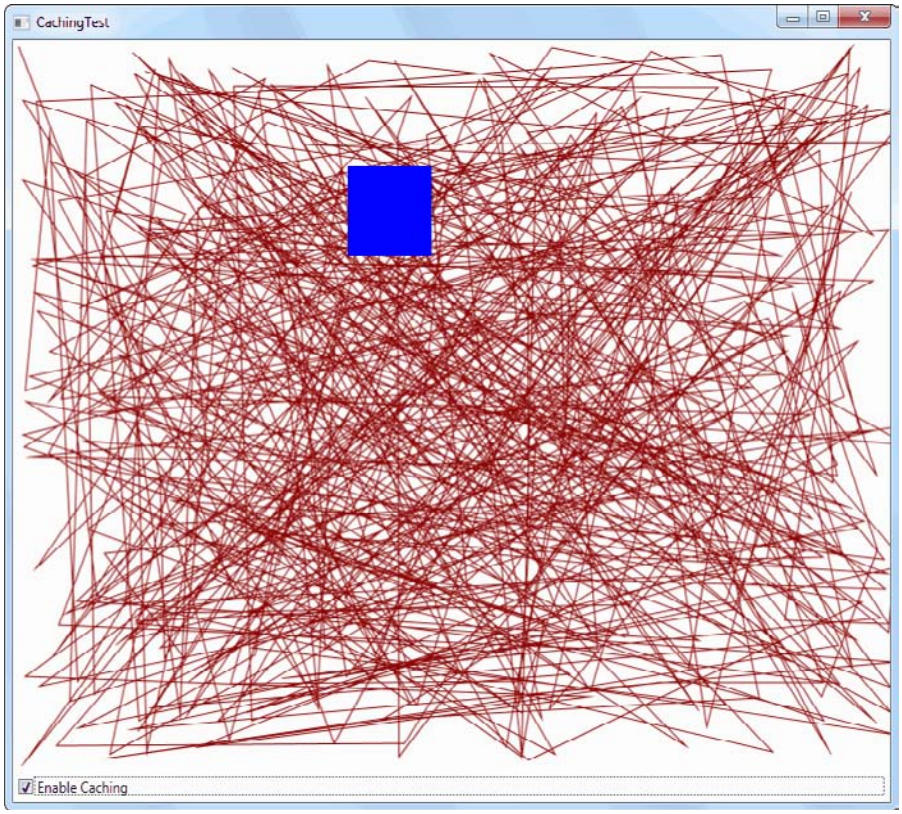

Figure 15-10. *Animating over a complex piece of vector art*

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% to less than 20%.

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 where 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 10 times its original size), but the video card is able to double the size of the bitmap from 5 to 10 times size without any obvious scaling artifacts. More importantly, 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 you can create and manage storyboards with code rather than XAML.

CHAPTER 16

Advanced Animation

You now know the fundamentals of WPF's property animation system—how animations are defined, how they're connected to elements, and how you can control playback with a storyboard. Now is a good time to take a closer look at the practical animation techniques you can use in an application.

In this chapter, you'll begin by considering what you should animate to get the results you want. You'll see examples that animate transforms, brushes, and pixel shaders. Next, you'll learn how key frame and path-based animations allow you to shape the acceleration and deceleration of your animations, in a way that's similar to animation easing but far more flexible. Then, you'll learn how frame-based animation lets you break free of the animation model altogether to create complex effects like realistic collisions. Lastly, you'll examine another example—a bomb-dropping game—that shows how you can integrate animations into the overall flow of an application, by creating and managing them with code.

What's New You won't learn about any new animation features in this chapter. However, you will see some examples that use the new WPF features you've considered in earlier chapters (such as pixel shaders and animation easing).

Animation Types Revisited

The first challenge in creating any animation is choosing the right property to animate. Making the leap between the result you want (for example, an element moving across the window) and the property you need to use (in this case, Canvas.Left and Canvas.Top) isn't always intuitive. Here are a few guidelines:

- If you want to use an animation to make an element appear or disappear, don't use the Visibility property (which allows you to switch only between completely visible or completely invisible). Instead, use the Opacity property to fade it in or out.
- If you want to animate the position of an element, consider using a Canvas. It provides the most direct properties (Canvas.Left and Canvas.Top) and requires the least overhead. Alternatively, you can get similar effects in other layout containers by animating properties such as Margin and Padding using the ThicknessAnimation class. You can also animate the MinWidth or MinHeight or a column or row in a Grid.

■ **Tip** Many animation effects are designed to progressively "reveal" an element. Common options include making an element fade into visibility, slide into view, or expand from a tiny point. However, there are many alternatives. For example, you could blur out an element using the BlurEffect described in Chapter 14 and animate the Radius property to reduce the blur and allow the element to come gradually into focus.

- The most common properties to animate are render transforms. You can use them to move or flip an element (TranslateTransform), rotate it (RotateTransform), resize or stretch it (ScaleTransform), and more. Used carefully, they can sometimes allow you to avoid hard-coding sizes and positions in your animation. They also bypass the WPF layout system, making them faster than other approaches that act directly on element size or position.
- One good way to change the surface of an element through an animation is to modify the properties of the brush. You can use a ColorAnimation to change the color or another animation object to transform a property of a more complex brush, like the offset in a gradient.

The following examples demonstrate how to animate transforms and brushes and how to use a few more animation types. You'll also learn how to create multisegmented animations with key frames, path-based animations, and frame-based animations.

Animating Transforms

Transforms offer one of the most powerful ways to customize an element. When you use transforms, you don't simply change the bounds of an element. Instead, the entire visual appearance of the element is moved, flipped, skewed, stretched, enlarged, shrunk, or rotated. For example, if you animate the size of a button using a ScaleTransform, the entire button is resized, including its border and its inner content. The effect is much more impressive than if you animate its Width and Height or the FontSize property that affects its text.

As you learned in Chapter 12, every element has the ability to use transforms in two different ways: the RenderTransform property and the LayoutTransform property. RenderTransform is more efficient, because it's applied after the layout pass and used to transform the final rendered output. LayoutTransform is applied before the layout pass, and as a result, other controls are rearranged to fit. Changing the LayoutTransform property triggers a new layout operation (unless you're using your element in a Canvas, in which case RenderTransform and LayoutTransform are equivalent).

To use a transform in animation, the first step is to define the transform. (An animation can change an existing transform but not create a new one.) For example, imagine you want to allow a button to rotate. This requires the RotateTransform:

```
<Button Content="A Button">
  <RenderTransform>
    <RotateTransform></RotateTransform>
  </RenderTransform>
</Button>
```

Now here's an event trigger that makes the button rotate when the mouse moves over it. It uses the target property RenderTransform.Angle—in other words, it reads the button's RenderTransform property and modifies the Angle property of the RotateTransform object that's defined there. The fact that the RenderTransform property can hold a variety of different transform objects, each with different properties, doesn't cause a problem. As long as you're using a transform that has an angle property, this trigger will work.

```
<EventTrigger RoutedEvent="Button.MouseEnter">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="RenderTransform.Angle"
         To="360" Duration="0:0:0.8" RepeatBehavior="Forever"></DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

The button rotates one revolution every 0.8 seconds and continues rotating perpetually. While the mouse is rotating, it's still completely usable—for example, you can click it and handle the Click event.

To make sure the button rotates around its center point (not the top-left corner), you need to set the RenderTransformOrigin property as shown here:

```
<Button RenderTransformOrigin="0.5,0.5">
```

Remember, the RenderTransformOrigin property uses relative units from 0 to 1, so 0.5 represents a midpoint.

To stop the rotation, you can use a second trigger that responds to the MouseLeave event. At this point, you could remove the storyboard that performs the rotation, but this causes the button to jump back to its original orientation in one step. A better approach is to start a second animation that replaces the first. This animation leaves out the To and From properties, which means it seamlessly rotates the button back to its original orientation in a snappy 0.2 seconds:

```
<EventTrigger RoutedEvent="Button.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.Angle"
         Duration="0:0:0.2"></DoubleAnimation>
       </Storyboard>
     </BeginStoryboard>
   </EventTrigger.Actions>
 </EventTrigger>
```

To create your rotating button, you'll need to add both these triggers to the Button.Triggers collection. Or, you could pull them (and the transform) into a style and apply that style to as many buttons as you want. For example, here's the markup for the window full of "rotatable" buttons shown in Figure 16-1:

```
<Window x:Class="Animation.RotateButton" ... >
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="HorizontalAlignment" Value="Center"></Setter>
      <Setter Property="RenderTransformOrigin" Value="0.5,0.5"></Setter>
      <Setter Property="Padding" Value="20,15"></Setter>
```

```
      <Setter Property="Margin" Value="2"></Setter>
      <Setter Property="LayoutTransform">
        <Setter.Value>
          <RotateTransform></RotateTransform>
        </Setter.Value>
      </Setter>
      <Style.Triggers>
        <EventTrigger RoutedEvent="Button.MouseEnter">
          ...
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.MouseLeave">
          ...
        </EventTrigger>
      </Style.Triggers>
    </Style>

  </Window.Resources>
  <StackPanel Margin="5" Button.Click="cmd_Clicked">
    <Button>One</Button>
    <Button>Two</Button>
    <Button>Three</Button>
    <Button>Four</Button>
    <TextBlock Name="lbl" Margin="5"></TextBlock>
  </StackPanel>
</Window>
```

When any button is clicked, a message is displayed in the TextBlock.

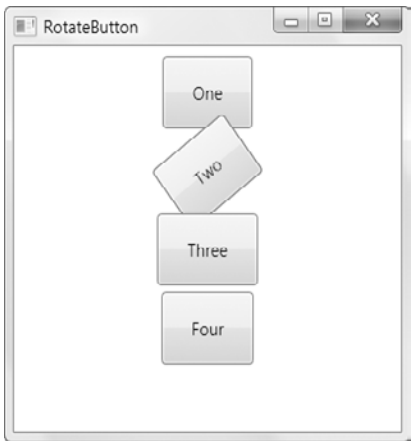

Figure 16-1. *Using a render transform*

This example also gives you a great chance to consider the difference between the RenderTransform and the LayoutTransform. If you modify the code to use a LayoutTransform, you'll see that the other buttons are pushed out of the way as a button spins (see Figure 16-2). For example, if the topmost button turns, the buttons underneath bounce up and down to avoid it.

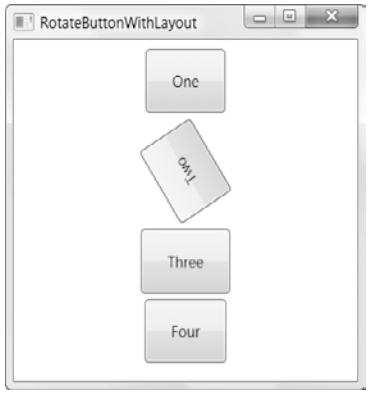

Figure 16-2. *Using a layout transform*

Of course, to get a sense of how the buttons "feel," it's worth trying this example with the downloadable code.

Animating Multiple Transforms

You can easily use transforms in combination. In fact, it's easy—you simply need to use the TransformGroup to set the LayoutTransform or RenderTransform property. You can nest as many transforms as you need inside the TransformGroup.

Figure 16-3 shows an interesting effect that was created using two transforms. A document window begins as a small thumbnail in the top-left corner of the window. When the window appears, this content rotates, expands, and fades into view rapidly. This is conceptually similar to the effect that Windows uses when you maximize a window. In WPF, you can use this trick with any element using transforms.

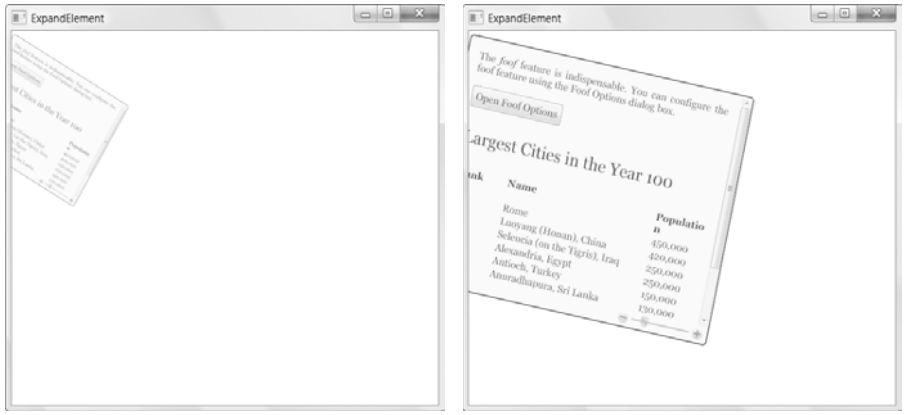

Figure 16-3. *Content that "jumps" into view*

To create this effect, two transforms are defined in a TransformGroup and used to set the RenderTransform property of a Border object that contains all the content.

```
<Border.RenderTransform>
  <TransformGroup>
    <ScaleTransform></ScaleTransform>
      <RotateTransform></RotateTransform>
    </TransformGroup>
</Border.RenderTransform>
```

Your animation can interact with both of these transform objects by specifying a numeric offset (0 for the ScaleTransform that appears first and 1 for the RotateTransform that's next). For example, here's the animation that enlarges the content:

```
<DoubleAnimation Storyboard.TargetName="element"
  Storyboard.TargetProperty="RenderTransform.Children[0].ScaleX"
  From="0" To="1" Duration="0:0:2" AccelerationRatio="1">
</DoubleAnimation>
<DoubleAnimation Storyboard.TargetName="element"
  Storyboard.TargetProperty="RenderTransform.Children[0].ScaleY"
  From="0" To="1" Duration="0:0:2" AccelerationRatio="1">
</DoubleAnimation>
```

and here's the animation in the same storyboard that rotates it:

```
<DoubleAnimation Storyboard.TargetName="element"
  Storyboard.TargetProperty="RenderTransform.Children[1].Angle"
  From="70" To="0" Duration="0:0:2" >
</DoubleAnimation>
```

The animation is slightly more involved than shown here. For example, there's an animation that increases the Opacity property at the same time, and when the Border reaches full size, it briefly "bounces" back, creating a more natural feel. Creating the timeline for this animation and tweaking the various animation object properties takes time—ideally, you'll perform tasks like this using a design tool such as Expression Blend rather than code them by hand. An even better scenario would be if a third-party developer grouped this logic into a single custom animation that you could then reuse and apply to your objects as needed. (As it currently stands, you could reuse this animation by storing the Storyboard as an application-level resource.)

This effect is surprisingly practical. For example, you could use it to draw attention to new content—such as a file that the user has just opened. The possible variations are endless. For example, a retail company could create a product catalog that slides a panel with product details or rolls a product image into view when you hover over the corresponding product name.

Animating Brushes

Animating brushes is another common technique in WPF animations, and it's just as easy as animating transforms. Once again, the technique is to dig into the particular subproperty you want to change, using the appropriate animation type.

Figure 16-4 shows an example that tweaks a RadialGradientBrush. As the animation runs, the center point of the radial gradient drifts along the ellipse, giving it a three-dimensional effect. At the same time, the outer color of the gradient changes from blue to black.

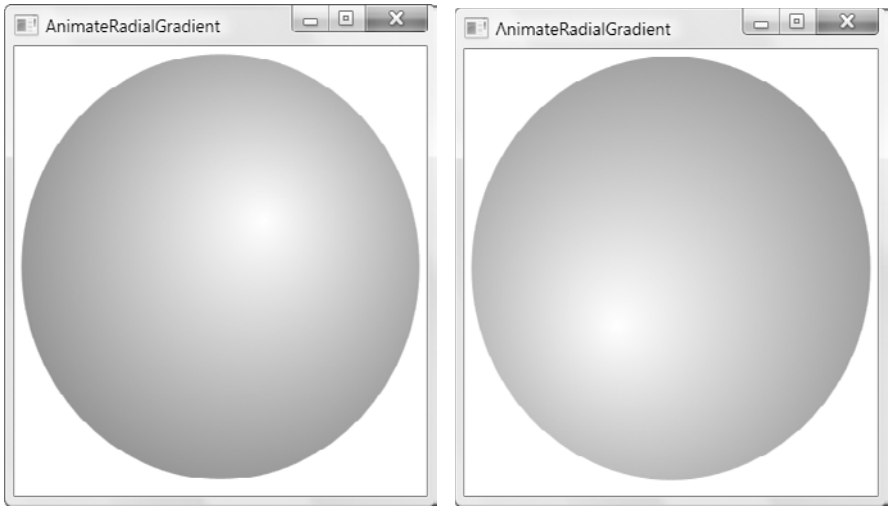

Figure 16-4. *Altering a radial gradient*

To perform this animation, you need to use two animation types that you haven't considered yet. ColorAnimation blends gradually between two colors, creating a subtle color-shift effect. PointAnimation allows you to move a point from one location to another. (It's essentially the same as if you modified both the X coordinate and the Y coordinate using a separate DoubleAnimation, with linear interpolation.) You can use a PointAnimation to deform a figure that you've constructed out of points or to change the location of the radial gradient's center point, as in this example.

Here's the markup that defines the ellipse and its brush:

```
<Ellipse Name="ellipse" Margin="5" Grid.Row="1" Stretch="Uniform">
  <Ellipse.Fill>
    <RadialGradientBrush
     RadiusX="1" RadiusY="1" GradientOrigin="0.7,0.3">
       <GradientStop Color="White" Offset="0"></GradientStop>
       <GradientStop Color="Blue" Offset="1"></GradientStop>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

and here are the two animations that move the center point and change the second color:

```
<PointAnimation Storyboard.TargetName="ellipse"
 Storyboard.TargetProperty="Fill.GradientOrigin"
 From="0.7,0.3" To="0.3,0.7" Duration="0:0:10" AutoReverse="True"
 RepeatBehavior="Forever">
</PointAnimation>
<ColorAnimation Storyboard.TargetName="ellipse"
 Storyboard.TargetProperty="Fill.GradientStops[1].Color"
 To="Black" Duration="0:0:10" AutoReverse="True"
 RepeatBehavior="Forever">
</ColorAnimation>
```