

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 such as realistic collisions. Finally, 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.

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 the element 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 using the `ThicknessAnimation` class to animate properties such as `Margin` and `Padding`. 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 by 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 by 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 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 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>
</Window.Resources>
</Window>
```

```

        </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.

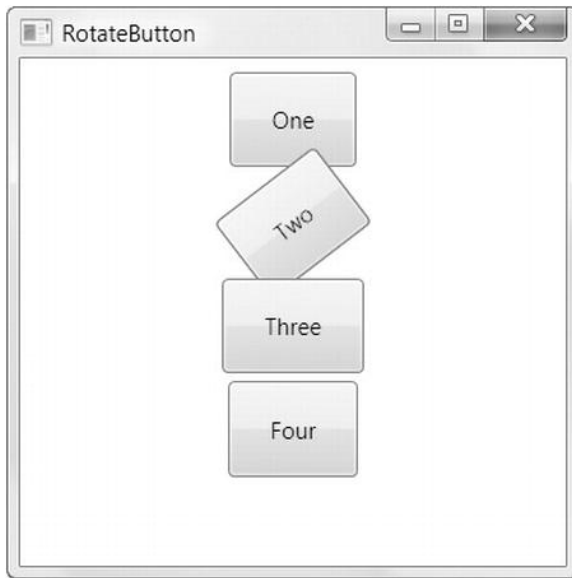


Figure 16-1. Using a render transform

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.

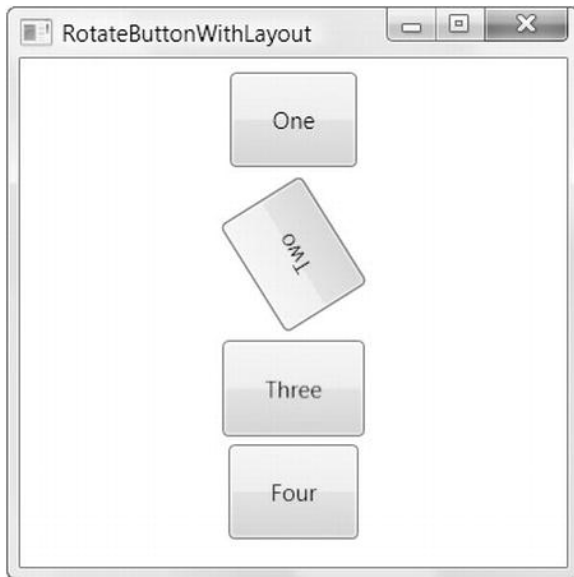


Figure 16-2. Using a layout transform

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 main window. When the document 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.

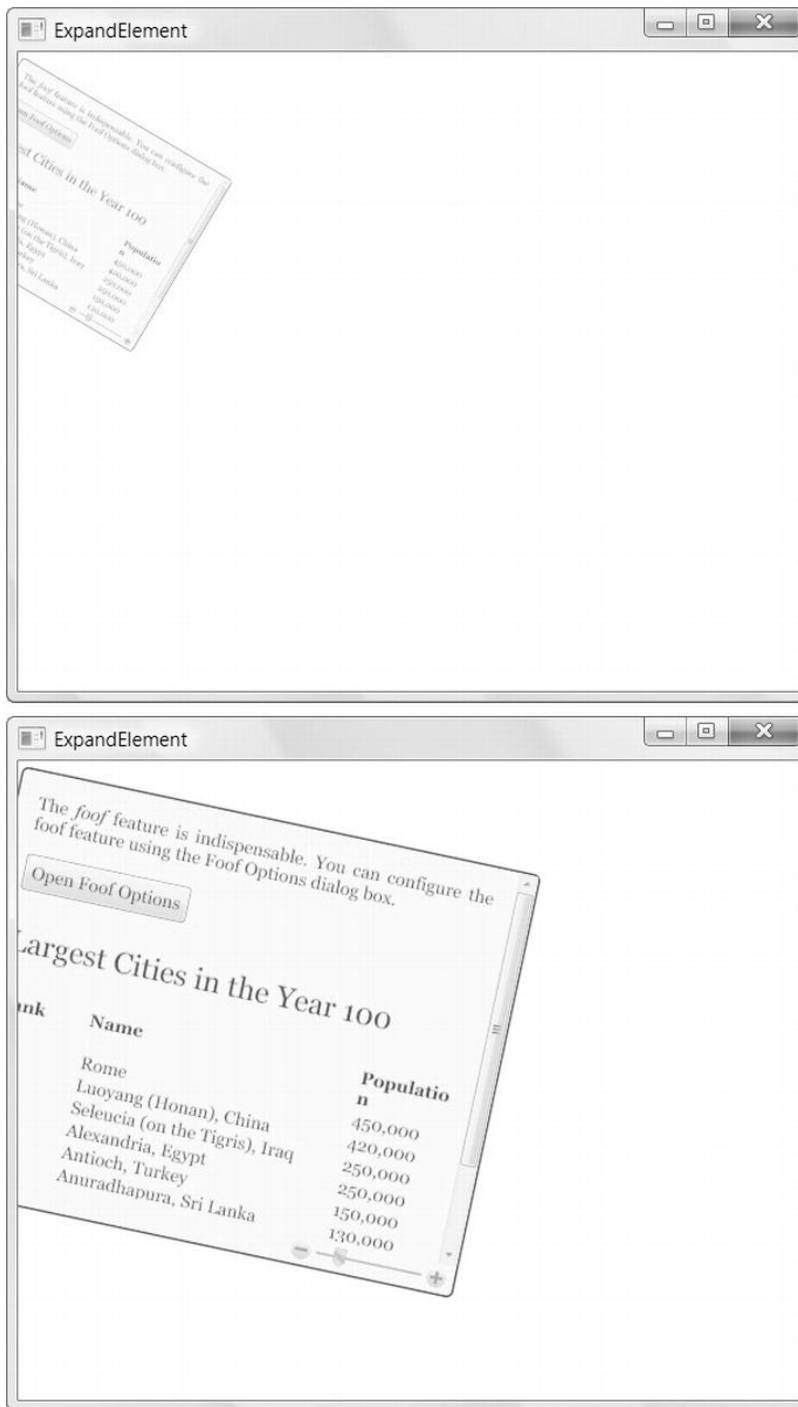


Figure 16-3. Content that “jumps” into view

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 having a third-party developer group 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.

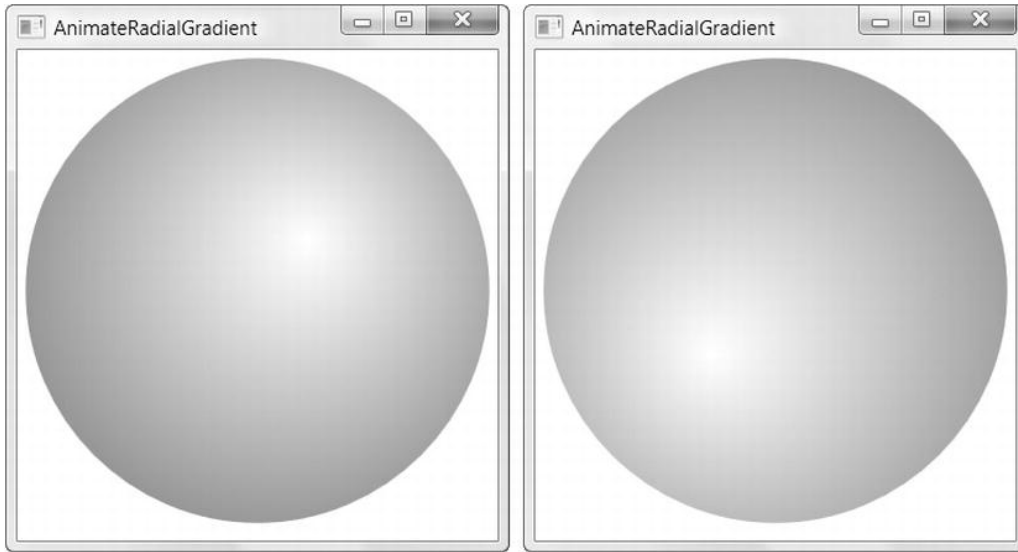


Figure 16-4. Altering a radial gradient

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 by 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>
```


You can create a huge range of hypnotic effects by varying the colors and offsets in `LinearGradientBrush` and `RadialGradientBrush`. And if that's not enough, gradient brushes also have their own `RelativeTransform` property that you can use to rotate, scale, stretch, and skew them. The WPF team has a fun tool called `Gradient Obsession` for building gradient-based animations. You can find it (and the source code) at <http://tinyurl.com/yc5fjpm>. For some additional ideas, check out the animation examples Charles Petzold provides at <http://tinyurl.com/y92mf8a>, which change the geometry of different `DrawingBrush` objects, creating tiled patterns that morph into different shapes.

VisualBrush

As you learned in Chapter 12, a `VisualBrush` allows you to take the appearance of any element and use it to fill another surface. That other surface can be anything from an ordinary rectangle to letters of text.

Figure 16-5 shows a basic example. On top sits a real, live button. Underneath, a `VisualBrush` is used to fill a rectangle with a picture of the button that stretches and rotates under the effect of various transforms.

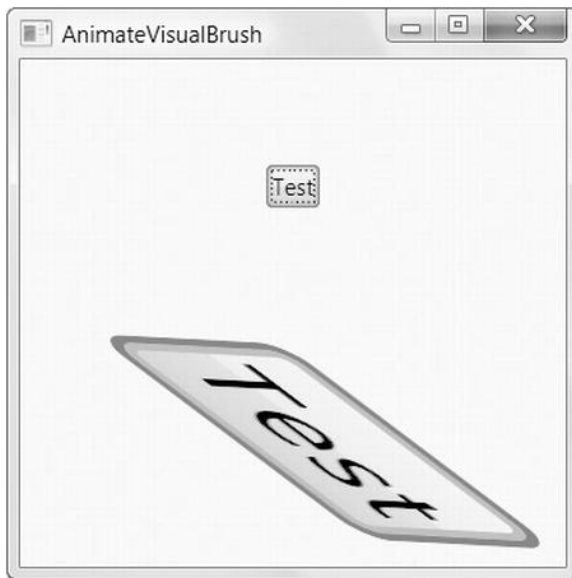


Figure 16-5. Animating an element that's filled with a `VisualBrush`

The `VisualBrush` also opens up some interesting possibilities for animation. For example, instead of animating the live, real element, you can animate a simple rectangle that has the same fill.

To understand how this works, consider the example shown earlier in Figure 16-3, which pops an element into view. While this animation is underway, the animated element is treated the same as any other WPF element, which means it's possible to click the button inside or scroll through the content with the keyboard (if you're fast enough). In some situations, this could cause confusion. In other situations, it might result in worse performance because of the extra overhead required to transform input (for example, mouse clicks) and pass it along to the original element.

Replacing this effect with a `VisualBrush` is easy. First you need to create another element that fills itself by using a `VisualBrush`. That `VisualBrush` must draw its visual from the element you want to animate (which, in this example, is the border named element).

```

<Rectangle Name="rectangle">
  <Rectangle.Fill>
    <VisualBrush Visual="{Binding ElementName=element}">
    </VisualBrush>
  </Rectangle.Fill>
  <Rectangle.RenderTransform>
    <TransformGroup>
      <ScaleTransform></ScaleTransform>
      <RotateTransform></RotateTransform>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>

```

To place the rectangle into the same position as the original element, you can place them both into the same cell of a Grid. The cell is sized to fit the original element (the border), and the rectangle is stretched to match. Another option is to overlay a Canvas on top of your real application layout container. (You could then bind your animation properties to the `ActualWidth` and `ActualHeight` properties of the real element underneath to make sure it lines up.)

After you've added the rectangle, you simply need to adjust your animations to animate its transforms. The final step is to hide the rectangle when the animations are complete:

```

private void storyboardCompleted(object sender, EventArgs e)
{
    rectangle.Visibility = Visibility.Collapsed;
}

```

Animating Pixel Shaders

In Chapter 14, you learned about pixel shaders—low-level routines that can apply bitmap-style effects such as blurs, glows, and warps to any element. On their own, pixel shaders are an interesting but only occasionally useful tool. But combined with animation, they become much more versatile. You can use them to design eye-catching transitions (for example, by blurring out one control, hiding it, and then removing the blur in another one). Or you can use them to create impressive user-interactivity effects (for example, by increasing the glow on a button when the user moves the mouse over it). Best of all, you can animate the properties of a pixel shader just as easily as you animate anything else.

Figure 16-6 shows a page that's based on the rotating button example shown earlier. It contains a sequence of buttons, and when the user moves the mouse over one of the buttons, an animation is attached and started. The difference is that the animation in this example doesn't rotate the button—instead, it reduces the blur radius to 0. The result is that as you move the mouse, the nearest control slides sharply and briskly into focus.

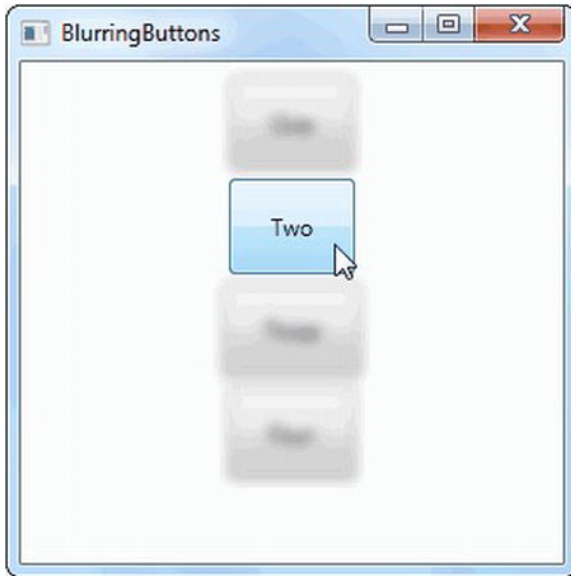


Figure 16-6. Animating a pixel shader

The code is the same as in the rotating button example. You need to give each button a `BlurEffect` instead of a `RotateTransform`:

```
<Button Content="A Button">
  <Button.Effect>
    <BlurEffect Radius="10"></BlurEffect>
  </Button.Effect>
</Button>
```

You also need to change the animation accordingly:

```
<EventTrigger RoutedEvent="Button.MouseEnter">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="Effect.Radius"
          To="0" Duration="0:0:0.4"></DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>

<EventTrigger RoutedEvent="Button.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="Effect.Radius" To="10"
          Duration="0:0:0.2"></DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

```

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

```

You could use the same approach in reverse to highlight a button. For example, you could use a pixel shader that applies a glow effect to highlight the moused-over button. And if you're interested in using pixel shaders to animate page transitions, check out the WPF Shader Effects Library at <http://codeplex.com/wpffx>. It includes a range of eye-popping pixel shaders and a set of helper classes for performing transitions with them.

Key-Frame Animation

All the animations you've seen so far have used linear interpolation to move from a starting point to an ending point. But what if you need to create an animation that has multiple segments and moves less regularly? For example, you might want to create an animation that slides an element into view quickly and then slowly moves it the rest of the way into place. You could achieve this effect by creating a sequence of two animations and using the `BeginTime` property to start the second animation after the first one. However, there's an easier approach—you can use a key-frame animation.

A *key-frame animation* is an animation that's made up of many short segments. Each segment represents an initial, final, or intermediary value in the animation. When you run the animation, it moves smoothly from one value to another.

For example, consider the `Point` animation that allowed you to move the center point of a `RadialGradientBrush` from one spot to another:

```

<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>

```

You can replace this `PointAnimation` object with an equivalent `PointAnimationUsingKeyFrames` object, as shown here:

```

<PointAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
  Storyboard.TargetProperty="Fill.GradientOrigin"
  AutoReverse="True" RepeatBehavior="Forever" >
  <LinearPointKeyFrame Value="0.7,0.3" KeyTime="0:0:0"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.3,0.7" KeyTime="0:0:10"></LinearPointKeyFrame>
</PointAnimationUsingKeyFrames>

```

This animation includes two key frames. The first sets the `Point` value when the animation first starts. (If you want to use the current value that's set in the `RadialGradientBrush`, you can leave out this key frame.) The second key frame defines the end value, which is reached after 10 seconds. The `PointAnimationUsingKeyFrames` object performs linear interpolation to move smoothly from the first key-frame value to the second, just as the `PointAnimation` does with the `From` and `To` values.

■ **Note** Every key-frame animation uses its own key-frame animation object (for example, `LinearPointKeyFrame`). For the most part, these classes are the same—they include a `Value` property that stores the target value and a `KeyTime` property that indicates when the frame reaches the target value. The only difference is the data type of the `Value` property. In a `LinearPointKeyFrame` it's a `Point`, in a `DoubleKeyFrame` it's a `double`, and so on.

You can create a more interesting example by using a series of key frames. The following animation walks the center point through a series of positions that are reached at different times. The speed that the center point moves will change depending on the length of time between key frames and the amount of distance that needs to be covered.

```
<PointAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
Storyboard.TargetProperty="Fill.GradientOrigin"
RepeatBehavior="Forever" >
  <LinearPointKeyFrame Value="0.7,0.3" KeyTime="0:0:0"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.3,0.7" KeyTime="0:0:5"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.5,0.9" KeyTime="0:0:8"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.9,0.6" KeyTime="0:0:10"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.8,0.2" KeyTime="0:0:12"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.7,0.3" KeyTime="0:0:14"></LinearPointKeyFrame>
</PointAnimationUsingKeyFrames>
```

This animation isn't reversible, but it does repeat. To make sure there's no jump between the final value of one iteration and the starting value of the next iteration, the animation ends at the same center point that it began.

Chapter 27 shows another key-frame example. It uses a `Point3DAnimationUsingKeyFrames` animation to move the camera through a 3-D scene and a `Vector3DAnimationUsingKeyFrames` to rotate the camera at the same time.

■ **Note** Using key-frame animation isn't quite as powerful as using a sequence of multiple animations. The most important difference is that you can't apply different `AccelerationRatio` and `DecelerationRatio` values to each key frame. Instead, you can apply only a single value to the entire animation.

Discrete Key-Frame Animations

The key-frame animation you saw in the previous example uses *linear* key frames. As a result, it transitions smoothly between the key-frame values. Another option is to use *discrete* key frames. In this case, no interpolation is performed. When the key time is reached, the property changes abruptly to the new value.

Linear key-frame classes are named in the form `LinearDataTypeKeyFrame`. Discrete key-frame classes are named in the form `DiscreteDataTypeKeyFrame`. Here's a revised version of the `RadialGradientBrush` example that uses discrete key frames:

```
<PointAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
Storyboard.TargetProperty="Fill.GradientOrigin"
RepeatBehavior="Forever" >
  <DiscretePointKeyFrame Value="0.7,0.3" KeyTime="0:0:0"></DiscretePointKeyFrame>
  <DiscretePointKeyFrame Value="0.3,0.7" KeyTime="0:0:5"></DiscretePointKeyFrame>
  <DiscretePointKeyFrame Value="0.5,0.9" KeyTime="0:0:8"></DiscretePointKeyFrame>
  <DiscretePointKeyFrame Value="0.9,0.6" KeyTime="0:0:10"></DiscretePointKeyFrame>
  <DiscretePointKeyFrame Value="0.8,0.2" KeyTime="0:0:12"></DiscretePointKeyFrame>
  <DiscretePointKeyFrame Value="0.7,0.3" KeyTime="0:0:14"></DiscretePointKeyFrame>
</PointAnimationUsingKeyFrames>
```

When you run this animation, the center point will jump from one position to the next at the appropriate time. It's a dramatic (but jerky) effect.

All key-frame animation classes support discrete key frames, but only some support linear key frames. It all depends on the data type. The data types that support linear key frames are the same ones that support linear interpolation and provide a *DataTypeAnimation* class. Examples include *Point*, *Color*, and *double*. Data types that don't support linear interpolation include *string* and *object*. You'll see an example in Chapter 26 that uses the *StringAnimationUsingKeyFrames* class to display different pieces of text as an animation progresses.

■ **Tip** You can combine both types of key frame—linear and discrete—in the same key-frame animation.

Easing Key Frames

In the previous chapter, you saw how easing functions can improve ordinary animations. Even though key-frame animations are split into multiple segments, each of these segments uses ordinary, boring linear interpolation.

If this isn't what you want, you can use animation easing to add acceleration or deceleration to individual key frames. However, the ordinary linear key frame and discrete key-frame classes don't support this feature. Instead, you need to use an *easing* key frame, such as *EasingDoubleKeyFrame*, *EasingColorKeyFrame*, or *EasingPointKeyFrame*. Each one works the same way as its linear counterpart but exposes an additional *EasingFunction* property.

Here's an example that uses animation easing to apply an accelerating effect to the first 5 seconds of the key-frame animation:

```
<PointAnimationUsingKeyFrames Storyboard.TargetName="ellipseBrush"
Storyboard.TargetProperty="GradientOrigin"
RepeatBehavior="Forever" >
  <LinearPointKeyFrame Value="0.7,0.3" KeyTime="0:0:0"></LinearPointKeyFrame>
  <EasingPointKeyFrame Value="0.3,0.7" KeyTime="0:0:5">
    <EasingPointKeyFrame.EasingFunction>
      <CircleEase></CircleEase>
    </EasingPointKeyFrame.EasingFunction>
  </EasingPointKeyFrame>
  <LinearPointKeyFrame Value="0.5,0.9" KeyTime="0:0:8"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.9,0.6" KeyTime="0:0:10"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.8,0.2" KeyTime="0:0:12"></LinearPointKeyFrame>
  <LinearPointKeyFrame Value="0.7,0.3" KeyTime="0:0:14"></LinearPointKeyFrame>
</PointAnimationUsingKeyFrames>
```

The combination of key frames and animation easing is a convenient way to model complex animations, but it still may not give you the control you need. Instead of using animation easing, you can create a mathematical formula that dictates the progression of your animation. This is the technique you'll learn in the next section.

Spline Key-Frame Animations

There's one more type of key frame: a *spline* key frame. Every class that supports linear key frames also supports spline key frames, and they're named in the form *SplineDataTypeKeyFrame*.

Like linear key frames, spline key frames use interpolation to move smoothly from one key value to another. The difference is that every spline key frame sports a *KeySpline* property. Using the *KeySpline*

property, you define a cubic Bézier curve that influences the way interpolation is performed. Although it's tricky to get the effect you want (at least without an advanced design tool to help you), this technique enables you to create more-seamless acceleration and deceleration and more-lifelike motion.

As you may remember from Chapter 13, a Bézier curve is defined by a start point, an end point, and two control points. In the case of a key spline, the start point is always (0,0), and the end point is always (1,1). You simply supply the two control points. The curve that you create describes the relationship between time (in the X axis) and the animated value (in the Y axis).

Here's an example that demonstrates a key spline animation by comparing the motion of two ellipses across a Canvas. The first ellipse uses a `DoubleAnimation` to move slowly and evenly across the window. The second ellipse uses a `DoubleAnimationUsingKeyFrames` with two `SplineDoubleKeyFrame` objects. It reaches the destination at the same time as the first ellipse (after 10 seconds), but it accelerates and decelerates during its travel, pulling ahead and dropping behind the other ellipse.

```
<DoubleAnimation Storyboard.TargetName="ellipse1"
  Storyboard.TargetProperty="(Canvas.Left)"
  To="500" Duration="0:0:10">
</DoubleAnimation>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="ellipse2"
  Storyboard.TargetProperty="(Canvas.Left)" >
  <SplineDoubleKeyFrame KeyTime="0:0:5" Value="250"
    KeySpline="0.25,0 0.5,0.7"></SplineDoubleKeyFrame>
  <SplineDoubleKeyFrame KeyTime="0:0:10" Value="500"
    KeySpline="0.25,0.8 0.2,0.4"></SplineDoubleKeyFrame>
</DoubleAnimationUsingKeyFrames>
```

The fastest acceleration occurs shortly after the 5-second mark, when the second `SplineDoubleKeyFrame` kicks in. Its first control point matches a relatively large Y axis value, which represents the animation progress (0.8) against a correspondingly smaller X axis value, which represents the time. As a result, the ellipse increases its speed over a small distance, before slowing down again.

Figure 16-7 shows a graphical depiction of the two curves that control the movement of the ellipse. To interpret these curves, remember that they chart the progress of the animation from top to bottom. Looking at the first curve, you can see that it follows a fairly even progress downward, with a short pause at the beginning and a gradual leveling off at the end. However, the second curve plummets downward quite quickly, achieving the bulk of its progress, and then levels off for the remainder of the animation.



Figure 16-7. Charting the progress of a key spline animation

Path-Based Animation

A path-based animation uses a PathGeometry object to set a property. Although a path-based animation can, in principle, be used to modify any property that has the right data type, it's most useful when animating position-related properties. In fact, the path-based animation classes are primarily intended to help you move a visual object along a path.

As you learned in Chapter 13, a PathGeometry object describes a figure that can include lines, arcs, and curves. Figure 16-8 shows an example with a PathGeometry object that consists of two arcs and a straight line segment that joins the last-defined point to the starting point. This creates a closed route over which a small vector image travels at a constant rate.

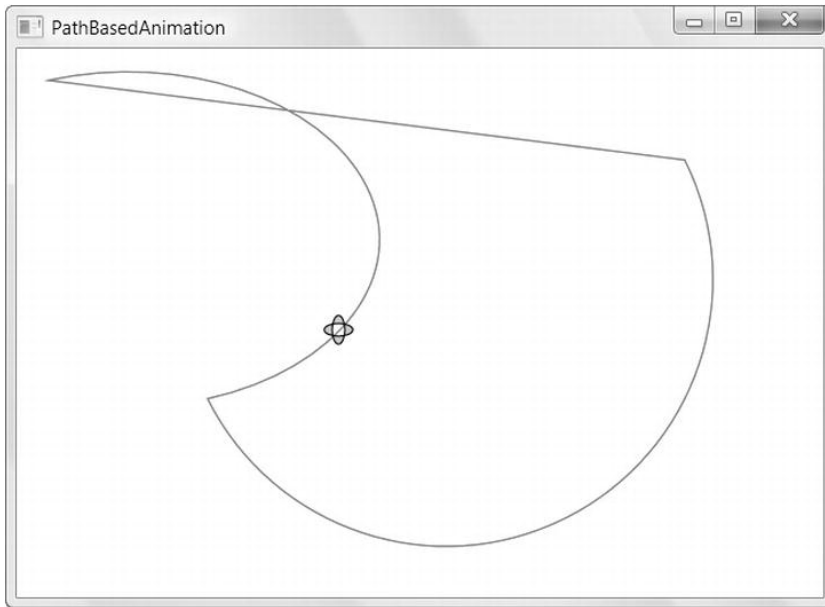


Figure 16-8. *Moving an image along a path*

Creating this example is easy. The first step is to build the path you want to use. In this example, the path is defined as a resource:

```
<Window.Resources>
  <PathGeometry x:Key="path">
    <PathFigure IsClosed="True">
      <ArcSegment Point="100,200" Size="15,10"
        SweepDirection="Clockwise"></ArcSegment>
      <ArcSegment Point="400,50" Size="5,5" ></ArcSegment>
    </PathFigure>
  </PathGeometry>
</Window.Resources>
```

Although it's not necessary, this example displays the path. That way, you can clearly see that the image follows the route you've defined. To show the path, you simply need to add a Path element that uses the geometry you've defined:


```
<Path Stroke="Red" StrokeThickness="1" Data="{StaticResource path}"
  Canvas.Top="10" Canvas.Left="10">
</Path>
```

The Path element is placed in a Canvas, along with the Image element that you want to move around the path:

```
<Image Name="image">
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <GeometryDrawing Brush="LightSteelBlue">
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <EllipseGeometry Center="10,10" RadiusX="9" RadiusY="4" />
              <EllipseGeometry Center="10,10" RadiusX="4" RadiusY="9" />
            </GeometryGroup>
          </GeometryDrawing.Geometry>
          <GeometryDrawing.Pen>
            <Pen Thickness="1" Brush="Black" />
          </GeometryDrawing.Pen>
        </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

The final step is to create the animations that move the image. To move the image, you need to adjust the Canvas.Left and Canvas.Top properties. The DoubleAnimationUsingPath does the trick, but you'll need two—one to work on the Canvas.Left property and one to deal with the Canvas.Top property. Here's the complete storyboard:

```
<Storyboard>
  <DoubleAnimationUsingPath Storyboard.TargetName="image"
    Storyboard.TargetProperty="(Canvas.Left)"
    PathGeometry="{StaticResource path}"
    Duration="0:0:5" RepeatBehavior="Forever" Source="X" />
  <DoubleAnimationUsingPath Storyboard.TargetName="image"
    Storyboard.TargetProperty="(Canvas.Top)"
    PathGeometry="{StaticResource path}"
    Duration="0:0:5" RepeatBehavior="Forever" Source="Y" />
</Storyboard>
```

As you can see, when creating a path-based animation, you don't supply starting and ending values. Instead, you indicate the PathGeometry that you want to use with the PathGeometry property. Some path-based animation classes, such as PointAnimationUsingPath, apply both the X and Y components to the destination property. The DoubleAnimationUsingPath class doesn't have this ability, because it sets just one double value. As a result, you also need to set the Source property to X or Y to indicate whether you're using the X coordinate or the Y coordinate from the path.

Although a path-based animation can use a path that includes a Bézier curve, it's quite a bit different from the key spline animations you learned about in the previous section. In a key spline animation, the Bézier curve describes the relationship between animation progress and time, allowing you to create an

animation that changes speed. But in a path-based animation, the collection of lines and curves that constitutes the path determines the *values* that will be used for the animated property.

■ **Note** A path-based animation always runs at a continuous speed. WPF considers the total length of the path and the duration you've specified to determine that speed.

Frame-Based Animation

Along with the property-based animation system, WPF provides a way to create frame-based animation using nothing but code. All you need to do is respond to the static `CompositionTarget.Rendering` event, which is fired to get the content for each frame. This is a far lower-level approach, which you won't want to tackle unless you're sure the standard property-based animation model won't work for your scenario (for example, if you're building a simple side-scrolling game, creating physics-based animations, or modeling particle effects such as fire, snow, and bubbles).

The basic technique for building a frame-based animation is easy. You simply need to attach an event handler to the static `CompositionTarget.Rendering` event. After you do, WPF will begin calling this event handler continuously. (As long as your rendering code executes quickly enough, WPF will call it 60 times each second.) In the rendering event handler, it's up to you to create or adjust the elements in the window accordingly. In other words, you need to manage all the work yourself. When the animation has ended, detach the event handler.

Figure 16-9 shows a straightforward example. Here, a random number of circles fall from the top of a Canvas to the bottom. They fall at different speeds (based on a random starting velocity), but they accelerate downward at the same rate. The animation ends when all the circles reach the bottom.

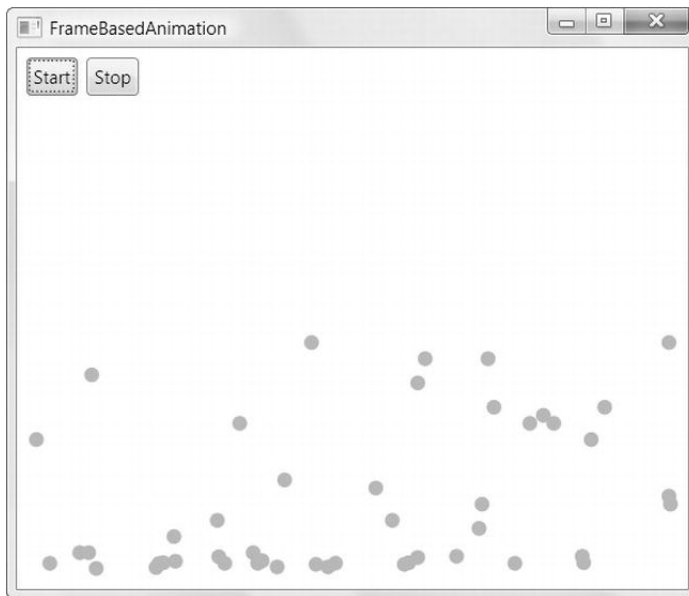


Figure 16-9. A frame-based animation of falling circles

In this example, each falling circle is represented by an `Ellipse` element. A custom class named `EllipseInfo` keeps a reference to the ellipse and tracks the details that are important for the physics model. In this case, there's only one piece of information—the velocity at which the ellipse is moving along the X axis. (You could easily extend this class to include a velocity along the Y axis, additional acceleration information, and so on.)

```
public class EllipseInfo
{
    public Ellipse Ellipse
    {
        get; set;
    }

    public double VelocityY
    {
        get; set;
    }

    public EllipseInfo(Ellipse ellipse, double velocityY)
    {
        VelocityY = velocityY;
        Ellipse = ellipse;
    }
}
```

The application keeps track of the `EllipseInfo` object for each ellipse by using a collection. There are several more window-level fields, which record various details that are used when calculating the fall of the ellipse. You could easily make these details configurable.

```
private List<EllipseInfo> ellipses = new List<EllipseInfo>();

private double accelerationY = 0.1;
private int minStartingSpeed = 1;
private int maxStartingSpeed = 50;
private double speedRatio = 0.1;
private int minEllipses = 20;
private int maxEllipses = 100;
private int ellipseRadius = 10;
```

When a button is clicked, the collection is cleared, and the event handler is attached to the `CompositionTarget.Rendering` event:

```
private bool rendering = false;

private void cmdStart_Clicked(object sender, RoutedEventArgs e)
{
    if (!rendering)
    {
        ellipses.Clear();
        canvas.Children.Clear();

        CompositionTarget.Rendering += RenderFrame;
        rendering = true;
    }
}
```

```
    }
}
```

If the ellipses don't exist, the rendering code creates them automatically. It creates a random number of ellipses (currently, between 20 and 100) and gives each of them the same size and color. The ellipses are placed at the top of the Canvas, but they're offset randomly along the X axis.

```
private void RenderFrame(object sender, EventArgs e)
{
    if (ellipses.Count == 0)
    {
        // Animation just started. Create the ellipses.
        int halfCanvasWidth = (int)canvas.ActualWidth / 2;

        Random rand = new Random();
        int ellipseCount = rand.Next(minEllipses, maxEllipses+1);
        for (int i = 0; i < ellipseCount; i++)
        {
            // Create the ellipse.
            Ellipse ellipse = new Ellipse();
            ellipse.Fill = Brushes.LimeGreen;
            ellipse.Width = ellipseRadius;
            ellipse.Height = ellipseRadius;

            // Place the ellipse.
            Canvas.SetLeft(ellipse, halfCanvasWidth +
                rand.Next(-halfCanvasWidth, halfCanvasWidth));
            Canvas.SetTop(ellipse, 0);
            canvas.Children.Add(ellipse);

            // Track the ellipse.
            EllipseInfo info = new EllipseInfo(ellipse,
                speedRatio * rand.Next(minStartingSpeed, maxStartingSpeed));
            ellipses.Add(info);
        }
    }
    ...
}
```

If the ellipses already exist, the code tackles the more interesting job of animating them. Each ellipse is moved slightly by using the `Canvas.SetTop()` method. The amount of movement depends on the assigned velocity.

```
...
else
{
    for (int i = ellipses.Count-1; i >= 0; i--)
    {
        EllipseInfo info = ellipses[i];
        double top = Canvas.GetTop(info.Ellipse);
        Canvas.SetTop(info.Ellipse, top + 1 * info.VelocityY);
        ...
    }
}
```

To improve performance, the ellipses are removed from the tracking collection as soon as they've reached the bottom of the Canvas. That way, you don't need to process them again. To allow this to work without causing you to lose your place while stepping through the collection, you need to iterate backward, from the end of the collection to the beginning.

If the ellipse hasn't yet reached the bottom of the Canvas, the code increases the velocity. (Alternatively, you could set the velocity based on how close the ellipse is to the bottom of the Canvas for a magnet-like effect.)

```
...
if (top >= (canvas.ActualHeight - ellipseRadius*2))
{
    // This circle has reached the bottom.
    // Stop animating it.
    ellipses.Remove(info);
}
else
{
    // Increase the velocity.
    info.VelocityY += accelerationY;
}
...
```

Finally, if all the ellipses have been removed from the collection, the event handler is removed, allowing the animation to end:

```
...
if (ellipses.Count == 0)
{
    // End the animation.
    // There's no reason to keep calling this method
    // if it has no work to do.
    CompositionTarget.Rendering -= RenderFrame;
    rendering = false;
}
}
}
```

Obviously, you could extend this animation to make the circles bounce, scatter, and so on. The technique is the same—you simply need to use more-complex formulas to arrive at the velocity.

There's one caveat to consider when building frame-based animations: they aren't time-dependent. In other words, your animation may run faster on fast computers, because the frame rate will increase and your `CompositionTarget.Rendering` event will be called more frequently. To compensate for this effect, you need to write code that takes the current time into account.

The best way to get started with frame-based animations is to check out the surprisingly detailed per-frame animation sample included with the WPF SDK (and also provided with the sample code for this chapter). It demonstrates several particle effects and uses a custom `TimeTracker` class to implement time-dependent frame-based animations.

Storyboards in Code

In the previous chapter, you saw how to create simple one-off animations in code and how to build more-sophisticated storyboards—complete with multiple animations and playback controls—with XAML markup. But sometimes, it makes sense to take the more sophisticated storyboard route but do all the hard work in code. In fact, this scenario is fairly common. It occurs anytime you have multiple animations to deal with and you don't know in advance how many animations there will be or how they should be configured. (This is the case with the simple bomb-dropping game you'll see in this section.) It also occurs if you want to use the same animation in different windows or you simply want the flexibility to separate all the animation-related details from your markup for easier reuse.

It isn't difficult to create, configure, and launch a storyboard programmatically. You just need to create the animation and storyboard objects, add the animations to the storyboard, and start the storyboard. You can perform any cleanup work after your animation ends by reacting to the `Storyboard.Completed` event.

In the following example, you'll see how to create the game shown in Figure 16-10. Here, a series of bombs are dropped at ever-increasing speeds. The player must click each bomb to defuse it. When a set limit is reached—by default, five dropped bombs—the game ends.

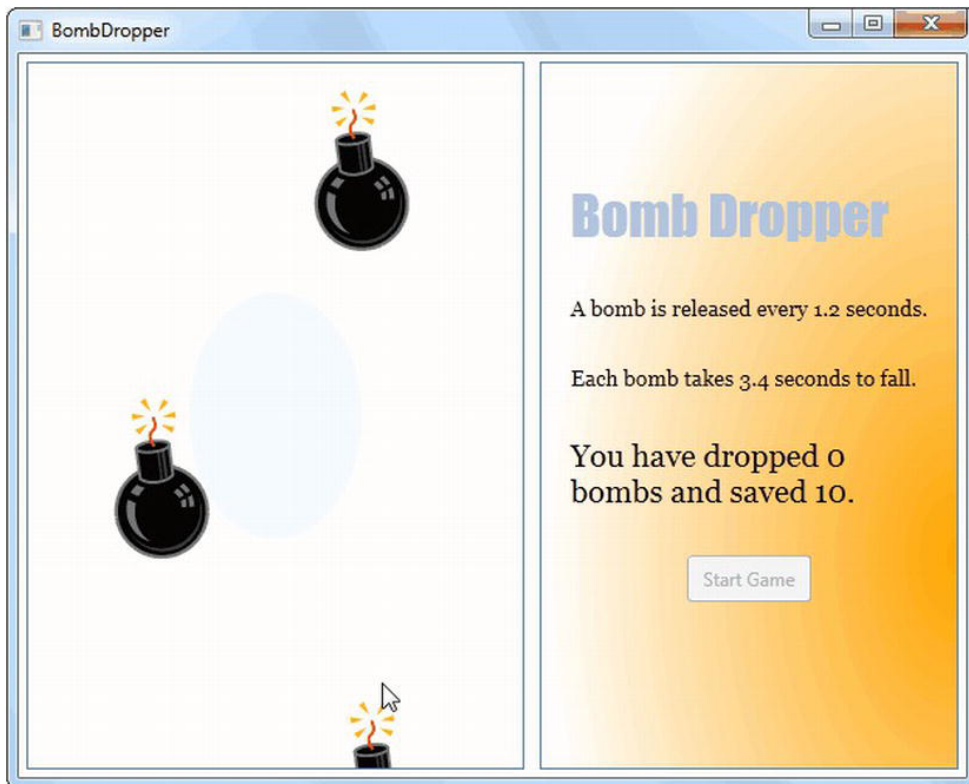


Figure 16-10. Catching bombs

In this example, every dropped bomb has its own storyboard with two animations. The first animation drops the bomb (by animating the `Canvas.Top` property), and the second animation rotates the bomb slightly back and forth, giving it a realistic wiggle effect. If the user clicks a dropping bomb, these

animations are halted and two more take place that send the bomb careening harmlessly off the side of the Canvas. Finally, every time an animation ends, the application checks to see whether it represents a bomb that fell down or one that was saved, and it updates the count accordingly.

In the following sections, you'll see how to create each part of this example.

Creating the Main Window

The main window in the BombDropper example is straightforward. It contains a two-column Grid. On the left side is a Border element, which contains the Canvas that represents the game surface:

```
<Border Grid.Column="0" BorderBrush="SteelBlue" BorderThickness="1" Margin="5">
  <Grid>
    <Canvas x:Name="canvasBackground" SizeChanged="canvasBackground_SizeChanged"
      MinWidth="50">
      <Canvas.Background>
        <RadialGradientBrush>
          <GradientStop Color="AliceBlue" Offset="0"></GradientStop>
          <GradientStop Color="White" Offset="0.7"></GradientStop>
        </RadialGradientBrush>
      </Canvas.Background>
    </Canvas>
  </Grid>
</Border>
```

When the Canvas is sized for the first time or resized (when the user changes the size of the window), the following code runs and sets the clipping region:

```
private void canvasBackground_SizeChanged(object sender, SizeChangedEventArgs e)
{
    // Set the clipping region to match the current display region of the Canvas.
    RectangleGeometry rect = new RectangleGeometry();
    rect.Rect = new Rect(0, 0,
        canvasBackground.ActualWidth, canvasBackground.ActualHeight);
    canvasBackground.Clip = rect;
}
```

This is required because otherwise the Canvas draws its children even if they lie outside its display area. In the bomb-dropping game, this would cause the bombs to fly out of the box that delineates the Canvas.

■ **Note** Because the user control is defined without explicit sizes, it's free to resize itself to match the window. The game logic uses the current window dimensions without attempting to compensate for them in any way. Thus, if you have a very wide window, bombs are spread across a wide area, making the game more difficult. Similarly, if you have a very tall window, bombs fall faster so they can complete their trajectory in the same interval of time. You could get around this issue by using a fixed-size region, which you could then center in the middle of your user control. However, a resizable window makes the example more adaptable and more interesting.

On the right side of the main window is a panel that shows the game statistics, the current bomb-dropped and bomb-saved count, and a button for starting the game:

```

<Border Grid.Column="1" BorderBrush="SteelBlue" BorderThickness="1" Margin="5">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="1,0.7" Center="1,0.7"
      RadiusX="1" RadiusY="1">
      <GradientStop Color="Orange" Offset="0"></GradientStop>
      <GradientStop Color="White" Offset="1"></GradientStop>
    </RadialGradientBrush>
  </Border.Background>

  <StackPanel Margin="15" VerticalAlignment="Center" HorizontalAlignment="Center">
    <TextBlock FontFamily="Impact" FontSize="35" Foreground="LightSteelBlue">
      Bomb Dropper</TextBlock>
    <TextBlock x:Name="lblRate" Margin="0,30,0,0" TextWrapping="Wrap"
      FontFamily="Georgia" FontSize="14"></TextBlock>
    <TextBlock x:Name="lblSpeed" Margin="0,30" TextWrapping="Wrap"
      FontFamily="Georgia" FontSize="14"></TextBlock>
    <TextBlock x:Name="lblStatus" TextWrapping="Wrap"
      FontFamily="Georgia" FontSize="20">No bombs have dropped.</TextBlock>
    <Button x:Name="cmdStart" Padding="5" Margin="0,30" Width="80"
      Content="Start Game" Click="cmdStart_Click"></Button>
  </StackPanel>
</Border>

```

Creating the Bomb User Control

The next step is to create the graphical image of the bomb. Although you can use a static image (as long as it has a transparent background), it's always better to deal with more-flexible WPF shapes. By using shapes, you gain the ability to resize the bomb without introducing distortion, and you can animate or alter individual parts of the drawing. The bomb shown in this example is drawn straight from Microsoft Word's online clip-art collection. The bomb was converted to XAML by inserting it into a Word document and then saving that document as an XPS file, a process described in Chapter 12. The full XAML, which uses a combination of Path elements, isn't shown here. But you can see it by downloading the BombDropper game along with the samples for this chapter.

The XAML for the Bomb class was then simplified slightly (by removing the unnecessary extra Canvas elements around it and the transforms for scaling it). The XAML was then inserted into a new user control named Bomb. This way, the main page can show a bomb by creating the Bomb user control and adding it to a layout container (such as a Canvas).

Placing the graphic in a separate user control makes it easy to instantiate multiple copies of that graphic in your user interface. It also lets you encapsulate related functionality by adding to the user control's code. In the bomb-dropping example, only one detail is added to the code—a Boolean property that tracks whether the bomb is currently falling:

```

public partial class Bomb: UserControl
{
    public Bomb()
    {
        InitializeComponent();
    }

    public bool IsFalling

```



```

    {
        get;
        set;
    }
}

```

The markup for the bomb includes a `RotateTransform`, which the animation code can use to give the bomb a wiggling effect as it falls. Although you could create and add this `RotateTransform` programmatically, it makes more sense to define it in the XAML file for the bomb:

```

<UserControl x:Class="BombDropper.Bomb"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <UserControl.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="20" CenterX="50" CenterY="50"></RotateTransform>
            <ScaleTransform ScaleX="0.5" ScaleY="0.5"></ScaleTransform>
        </TransformGroup>
    </UserControl.RenderTransform>

    <Canvas>
        <!-- The Path elements that draw the bomb graphic are defined here. -->
    </Canvas>
</UserControl>

```

With this code in place, you could insert a bomb into your window by using a `<bomb:Bomb>` element, much as the main window inserts the `Title` user control (as described in the previous section). However, in this case it makes far more sense to create the bombs programmatically.

Dropping the Bombs

To drop the bombs, the application uses `DispatcherTimer`, a timer that plays nicely with the WPF user interface because it triggers events on the user-interface thread (saving you the multithreaded programming challenges that are described in Chapter 31). You choose a time interval, and then the `DispatcherTimer` fires a periodic `Tick` event at that interval.

```

private DispatcherTimer bombTimer = new DispatcherTimer();

public MainWindow()
{
    InitializeComponent();
    bombTimer.Tick += bombTimer_Tick;
}

```

In the `BombDropper` game, the timer initially fires every 1.3 seconds. When the user clicks the button to start the game, the timer is started:

```

// Keep track of how many bombs are dropped and stopped.
private int droppedCount = 0;
private int savedCount = 0;

// Initially, bombs fall every 1.3 seconds, and hit the ground after 3.5 seconds.

```

```

private double initialSecondsBetweenBombs = 1.3;
private double initialSecondsToFall = 3.5;
private double secondsBetweenBombs;
private double secondsToFall;

private void cmdStart_Click(object sender, RoutedEventArgs e)
{
    cmdStart.IsEnabled = false;

    // Reset the game.
    droppedCount = 0;
    savedCount = 0;
    secondsBetweenBombs = initialSecondsBetweenBombs;
    secondsToFall = initialSecondsToFall;

    // Start the bomb-dropping timer.
    bombTimer.Interval = TimeSpan.FromSeconds(secondsBetweenBombs);
    bombTimer.Start();
}

```

Every time the timer fires, the code creates a new Bomb object and sets its position on the Canvas. The bomb is placed just above the top edge of the Canvas so it can fall seamlessly into view. It's given a random horizontal position that falls somewhere between the left and right sides:

```

private void bombTimer_Tick(object sender, EventArgs e)
{
    // Create the bomb.
    Bomb bomb = new Bomb();
    bomb.IsFalling = true;

    // Position the bomb.
    Random random = new Random();
    bomb.SetValue(Canvas.LeftProperty,
        (double)(random.Next(0, (int)(canvasBackground.ActualWidth - 50))));
    bomb.SetValue(Canvas.TopProperty, -100.0);

    // Add the bomb to the Canvas.
    canvasBackground.Children.Add(bomb);
    ...
}

```

The code then dynamically creates a storyboard to animate the bomb. Two animations are used: one that drops the bomb by changing the attached Canvas.Top property and one that wiggles the bomb by changing the angle of its rotate transform. Because Storyboard.TargetElement and Storyboard.TargetProperty are attached properties, you must set them using the Storyboard.SetTargetElement() and Storyboard.SetTargetProperty() methods:

```

...
// Attach mouse click event (for defusing the bomb).
bomb.MouseLeftButtonDown += bomb_MouseLeftButtonDown;

// Create the animation for the falling bomb.
Storyboard storyboard = new Storyboard();

```

```

DoubleAnimation fallAnimation = new DoubleAnimation();
fallAnimation.To = canvasBackground.ActualHeight;
fallAnimation.Duration = TimeSpan.FromSeconds(secondsToFall);

Storyboard.SetTarget(fallAnimation, bomb);
Storyboard.SetTargetProperty(fallAnimation, new PropertyPath("(Canvas.Top)"));
storyboard.Children.Add(fallAnimation);

// Create the animation for the bomb "wiggle."
DoubleAnimation wiggleAnimation = new DoubleAnimation();
wiggleAnimation.To = 30;
wiggleAnimation.Duration = TimeSpan.FromSeconds(0.2);
wiggleAnimation.RepeatBehavior = RepeatBehavior.Forever;
wiggleAnimation.AutoReverse = true;

Storyboard.SetTarget(wiggleAnimation,
    ((TransformGroup)bomb.RenderTransform).Children[0]);
Storyboard.SetTargetProperty(wiggleAnimation, new PropertyPath("Angle"));
storyboard.Children.Add(wiggleAnimation);
...

```

Both of these animations could use animation easing for more-realistic behavior, but this example keeps the code simple by using basic linear animations.

The newly created storyboard is stored in a dictionary collection so it can be retrieved easily in other event handlers. The collection is stored as a field in the main window class:

```

// Make it possible to look up a storyboard based on a bomb.
private Dictionary<Storyboard, Bomb> bombs = new Dictionary<Storyboard, Bomb>();
Here's the code that adds the storyboard to the tracking collection:
...
storyboards.Add(bomb, storyboard);
...

```

Next, you attach an event handler that reacts when the storyboard finishes the fallAnimation, which occurs when the bomb hits the ground. Finally, the storyboard is started, and the animations are put in motion:

```

...
storyboard.Duration = fallAnimation.Duration;
storyboard.Completed += storyboard_Completed;
storyboard.Begin();
...

```

The bomb-dropping code needs one last detail. As the game progresses, it becomes more difficult. The timer begins to fire more frequently, the bombs begin to appear more closely together, and the fall time is reduced. To implement these changes, the timer code makes adjustments whenever a set interval of time has passed. By default, BombDropper makes an adjustment every 15 seconds. Here are the fields that control the adjustments:

```

// Perform an adjustment every 15 seconds.
private double secondsBetweenAdjustments = 15;
private DateTime lastAdjustmentTime = DateTime.MinValue;

```

```
// After every adjustment, shave 0.1 seconds off both.
private double secondsBetweenBombsReduction = 0.1;
private double secondsToFallReduction = 0.1;
```

And here's the code at the end of the `DispatcherTimer.Tick` event handler, which checks whether an adjustment is needed and makes the appropriate changes:

```
...
// Perform an "adjustment" when needed.
if ((DateTime.Now.Subtract(lastAdjustmentTime).TotalSeconds >
    secondsBetweenAdjustments))
{
    lastAdjustmentTime = DateTime.Now;

    secondsBetweenBombs -= secondsBetweenBombsReduction;
    secondsToFall -= secondsToFallReduction;

    // (Technically, you should check for 0 or negative values.
    // However, in practice these won't occur because the game will
    // always end first.)

    // Set the timer to drop the next bomb at the appropriate time.
    bombTimer.Interval = TimeSpan.FromSeconds(secondsBetweenBombs);

    // Update the status message.
    lblRate.Text = String.Format("A bomb is released every {0} seconds.",
        secondsBetweenBombs);
    lblSpeed.Text = String.Format("Each bomb takes {0} seconds to fall.",
        secondsToFall);
}
}
```

With this code in place, there's enough functionality to drop bombs at an ever-increasing rate. However, the game still lacks the code that responds to dropped and saved bombs.

Intercepting a Bomb

The user saves a bomb by clicking it before it reaches the bottom of the Canvas. Because each bomb is a separate instance of the `Bomb` user control, intercepting mouse clicks is easy—all you need to do is handle the `MouseLeftButtonDown` event, which fires when any part of the bomb is clicked (but doesn't fire if you click somewhere in the background, such as around the edges of the bomb circle).

When a bomb is clicked, the first step is to get the appropriate bomb object and set its `IsFalling` property to indicate that it's no longer falling. (The `IsFalling` property is used by the event handler that deals with completed animations.)

```
private void bomb_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    // Get the bomb.
    Bomb bomb = (Bomb)sender;
    bomb.IsFalling = false;

    // Record the bomb's current (animated) position.
```

```
double currentTop = Canvas.GetTop(bomb);
...
```

The next step is to find the storyboard that controls the animation for this bomb so it can be stopped. To find the storyboard, you need to look it up in the collection that this game uses for tracking. Currently, WPF doesn't include any standardized way to find the animations that are acting on a given element.

```
...
// Stop the bomb from falling.
Storyboard storyboard = storyboards[bomb];
storyboard.Stop();
...
```

After a button is clicked, another set of animations moves the bomb off the screen, throwing it up and left or right (depending on which side is closest). Although you could create an entirely new storyboard to implement this effect, the BombDropper game clears the current storyboard that's being used for the bomb and adds new animations to it. When this process is completed, the new storyboard is started:

```
...
// Reuse the existing storyboard, but with new animations.
// Send the bomb on a new trajectory by animating Canvas.Top
// and Canvas.Left.
storyboard.Children.Clear();

DoubleAnimation riseAnimation = new DoubleAnimation();
riseAnimation.From = currentTop;
riseAnimation.To = 0;
riseAnimation.Duration = TimeSpan.FromSeconds(2);

Storyboard.SetTarget(riseAnimation, bomb);
Storyboard.SetTargetProperty(riseAnimation, new PropertyPath("(Canvas.Top)"));
storyboard.Children.Add(riseAnimation);

DoubleAnimation slideAnimation = new DoubleAnimation();
double currentLeft = Canvas.GetLeft(bomb);

// Throw the bomb off the closest side.
if (currentLeft < canvasBackground.ActualWidth / 2)
{
    slideAnimation.To = -100;
}
else
{
    slideAnimation.To = canvasBackground.ActualWidth + 100;
}
slideAnimation.Duration = TimeSpan.FromSeconds(1);
Storyboard.SetTarget(slideAnimation, bomb);
Storyboard.SetTargetProperty(slideAnimation, new PropertyPath("(Canvas.Left)"));
storyboard.Children.Add(slideAnimation);

// Start the new animation.
storyboard.Duration = slideAnimation.Duration;
```

```

        storyboard.Begin();
    }

```

Now the game has enough code to drop bombs and bounce them off the screen when the user saves them. However, to keep track of which bombs are saved and which ones are dropped, you need to react to the `Storyboard.Completed` event that fires at the end of an animation.

Counting Bombs and Cleaning Up

As you've seen, the `BombDropper` uses storyboards in two ways: to animate a falling bomb and to animate a defused bomb. You could handle the completion of these storyboards with different event handlers, but to keep things simple, the `BombDropper` uses just one. It tells the difference between an exploded bomb and a rescued bomb by examining the `Bomb.IsFalling` property.

```

// End the game when 5 bombs have fallen.
private int maxDropped = 5;

private void storyboard_Completed(object sender, EventArgs e)
{
    ClockGroup clockGroup = (ClockGroup)sender;

    // Get the first animation in the storyboard, and use it to find the
    // bomb that's being animated.
    DoubleAnimation completedAnimation =
        (DoubleAnimation)clockGroup.Children[0].Timeline;
    Bomb completedBomb = (Bomb)Storyboard.GetTarget(completedAnimation);

    // Determine if a bomb fell or flew off the Canvas after being clicked.
    if (completedBomb.IsFalling)
    {
        droppedCount++;
    }
    else
    {
        savedCount++;
    }
    ...
}

```

Either way, the code then updates the display test to indicate the number of bombs that have been dropped and saved:

```

...
// Update the display.
lblStatus.Text = String.Format("You have dropped {0} bombs and saved {1}.",
    droppedCount, savedCount);
...

```

At this point, the code checks to see whether the maximum number of dropped bombs has been reached. If it has, the game ends, the timer is stopped, and all the bombs and storyboards are removed:

```

...
// Check if it's game over.
if (droppedCount >= maxDropped)

```

```

{
    bombTimer.Stop();
    lblStatus.Text += "\r\n\r\nGame over.";

    // Find all the storyboards that are underway.
    foreach (KeyValuePair<Bomb, Storyboard> item in storyboards)
    {
        Storyboard storyboard = item.Value;
        Bomb bomb = item.Key;

        storyboard.Stop();
        canvasBackground.Children.Remove(bomb);
    }

    // Empty the tracking collection.
    storyboards.Clear();

    // Allow the user to start a new game.
    cmdStart.IsEnabled = true;
}
else
{
    // Clean up just this bomb, and let the game continue.
    Storyboard storyboard = (Storyboard)clockGroup.Timeline;
    storyboard.Stop();

    storyboards.Remove(completedBomb);
    canvasBackground.Children.Remove(completedBomb);
}
}

```

This completes the code for BombDropper game. However, you can make plenty of refinements. Some examples include the following:

Animate a bomb explosion effect: This effect can make the flames around the bomb twinkle or send small pieces of shrapnel flying across the Canvas.

Animate the background: This change is easy, and it adds pizzazz. For example, you can create a linear gradient that shifts up, creating an impression of movement, or one that transitions between two colors.

Add depth: It's easier than you think. The basic technique is to give the bombs different sizes. Bombs that are bigger should have a higher ZIndex, ensuring that they overlap smaller bombs, and should be given a shorter animation time, ensuring that they fall faster. You can also make the bombs partially transparent, so as one falls, the others behind it are visible.

Add sound effects: In Chapter 26, you'll learn to use sound and other media in WPF. You can use well-timed sound effects to punctuate bomb explosions or rescued bombs.

Use animation easing: If you want bombs to accelerate as they fall, bounce off the screen, or wiggle more naturally, you can add easing functions to the animations used here. And, as you'd expect, easing functions can be constructed in code just as easily as in XAML.

Fine-tune the parameters: You can provide more dials to tweak behavior (for example, variables that set how the bomb times, trajectories, and frequencies that are altered as the game processes). You can also inject more randomness (for example, allowing saved bombs to bounce off the Canvas in slightly different ways).

The Last Word

In this chapter, you learned the techniques needed to make practical animations and integrate them into your applications. The only missing ingredient is the eye candy—in other words, making sure the animated effects are as polished as your code.

As you've seen over the past two chapters, the animation model in WPF is surprisingly full-featured. However, getting the result you want isn't always easy. If you want to animate separate portions of your interface as part of a single animated “scene,” you're often forced to write a fair bit of markup with interdependent details that aren't always clear. In more-complex animations, you may be forced to hard-code details and fall back to code to perform calculations for the ending value of animation. And if you need fine-grained control over an animation, such as when modeling a physical particle system, you'll need to control every step of the way by using frame-based animation.

The future of WPF animation promises higher-level classes that are built on the basic plumbing you've learned about in this chapter. Ideally, you'll be able to plug animations into your application simply by using prebuilt animation classes, wrapping your elements in specialized containers, and setting a few attached properties. The actual implementation that generates the effect you want—whether it's a smooth dissolve between two images or a series of animated fly-ins that builds a window—will be provided for you.