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.

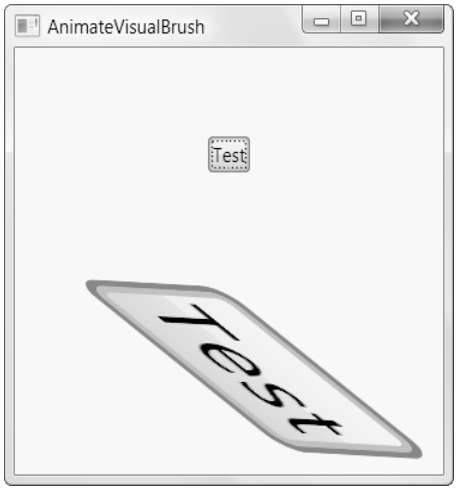

Figure 16-5. *Animating an element that's filled with a VisualBrush*

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 (like 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 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.)

Once 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 one control out, hiding it, and then blurring another one in). 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.

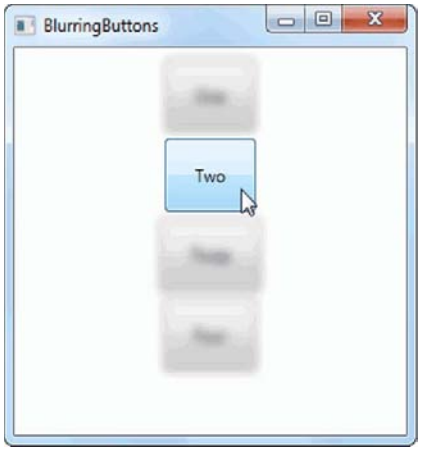

Figure 16-6. *Animating a pixel shader*

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>
```

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 ten 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 (like LinearPointKeyFrame). For the most point, 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 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 how long the duration is between key frames and how much distance 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 3D scene and a Vector3DAnimationUsingKeyFrames to rotate the camera at the same time.

■ **Note** Using a 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 Linear*DataType*KeyFrame. Discrete key frame classes are named in the form Discrete*DataType*KeyFrame. 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 *DataType*Animation 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 five 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 Spline*DataType*KeyFrame.

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 gives the ability 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 times (after ten 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 five-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.

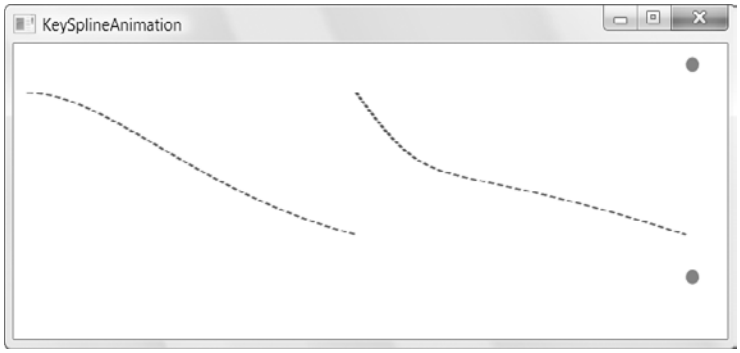

Figure 16-7. *Charting the progress of a key spline animation*

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.

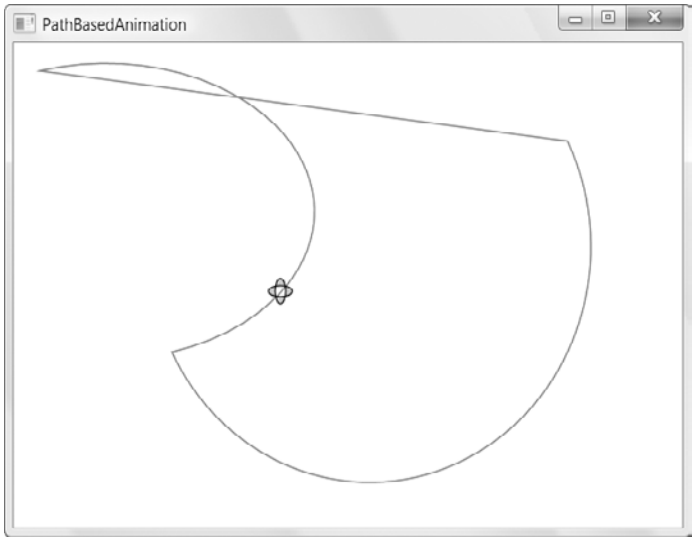

Figure 16-8. *Moving an image along a path*

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. Once 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.

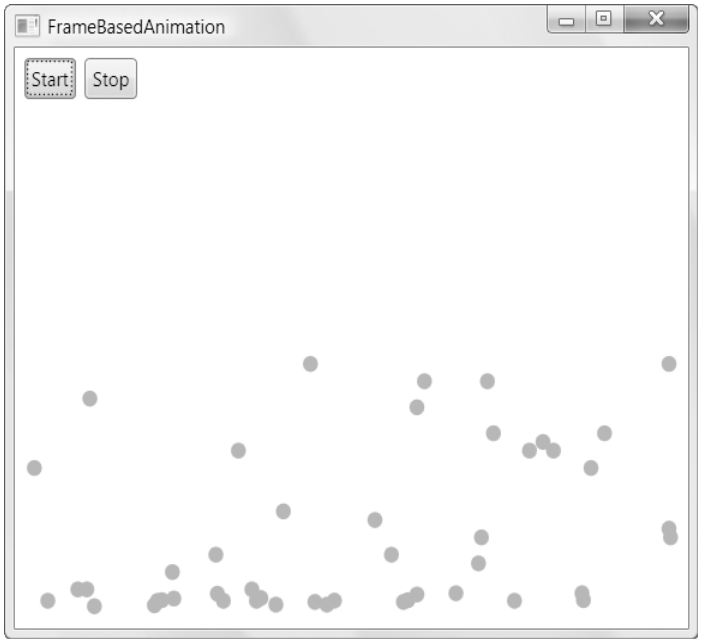

Figure 16-9. *A frame-based animation of falling circles*

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 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 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 any time 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.

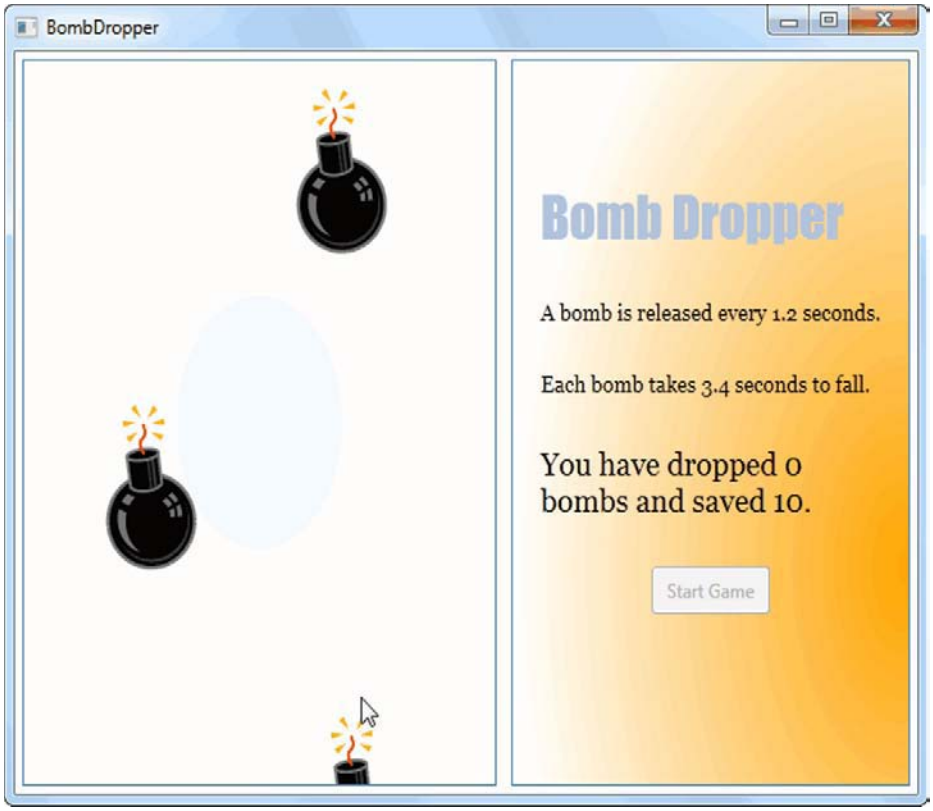

Figure 16-10. *Catching bombs*

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.

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>
```

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 (like 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 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 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;
```