CHAPTER 26

Sound and Video

In this chapter, you'll tackle two more areas of WPF functionality: audio and video. The support WPF provides for audio is a significant step up from the first versions of .NET, but it's far from groundbreaking. WPF gives you the ability to play a wide variety of sound formats, including MP3 files and anything else supported by Windows Media Player. However, WPF's sound capabilities still fall far short of DirectSound (the advanced audio API in DirectX), which allows you to apply dynamic effects and place sounds in a simulated 3-D space. WPF also lacks a way to retrieve spectrum data that tells you the highs and lows of sound, which is useful for creating some types of synchronized effects and sound-driven animations.

WPF's video support is more impressive. Although the ability to play video (such as MPEG and WMV files) isn't earth-shattering, the way it integrates into the rest of the WPF model is dramatic. For example, you can use video to fill thousands of elements at once and combine it with effects, animation, transparency, and even 3-D objects.

In this chapter, you'll see how to integrate video and audio content into your applications. You'll even take a quick look at WPF's support for speech synthesis and speech recognition. But before you get to the more exotic examples, you'll begin by considering the basic code required to play humble WAV audio.

Playing WAV Audio

The simplest way to play audio files in .NET is with the underwhelming SoundPlayer class, which you can find in the underpopulated System.Media namespace. The SoundPlayer is severely limited: it can play only WAV audio files, it doesn't support playing more than one sound at once, and it doesn't provide the ability to control any aspect of the audio playback (for example, details such as volume and balance).

If you can live with the SoundPlayer's significant limitations, it still presents the easiest, most lightweight way to add audio to an application. The SoundPlayer class is also wrapped by the SoundPlayerAction class, which allows you to play sounds through a declarative trigger (rather than writing a few lines of C# code in an event handler). In the following sections, you'll take a quick look at both classes, before you move on to WPF's much more powerful MediaPlayer and MediaElement classes.

The SoundPlayer

To play a sound with the SoundPlayer class, you follow several steps:

1. Create a SoundPlayer instance.

2. Specify the sound content by setting either the SoundLocation property or the Stream property. If you have a file path that points to a WAV file, use the SoundLocation property. If you have a Stream-based object that contains WAV audio content, use the Stream property.

■ **Note** If your audio content is stored in a binary resource and embedded in your application, you'll need to access it as a stream (see Chapter 7) and use the SoundPlayer.Stream property. That's because the SoundPlayer doesn't support WPF's pack URI syntax.

---

3. After you've set the Stream or SoundLocation property, you can tell SoundPlayer to actually load the audio data by calling the Load() or LoadAsync() method. The Load() method is the simplest—it stalls your code until all the audio is loaded into memory. LoadAsync() quietly carries its work out on another thread and fires the LoadCompleted event when it's finished.

■ **Note** Technically, you don't need to use Load() or LoadAsync(). The SoundPlayer will load the audio data if needed when you call Play() or PlaySync(). However, it's a good idea to explicitly load the audio—not only does that save you the overhead if you need to play it multiple times, but it also makes it easy to handle exceptions related to file problems separately from exceptions related to audio playback problems.

---

4. Now you can call PlaySync() to pause your code while the audio plays, or you can use Play() to play the audio on another thread, ensuring that your application's interface remains responsive. Your only other option is PlayLooping(), which plays the audio asynchronously in an unending loop (perfect for those annoying soundtracks). To halt the current playback at any time, just call Stop().

The following code snippet shows the simplest approach to load and play a sound asynchronously:

```
SoundPlayer player = new SoundPlayer();
player.SoundLocation = "test.wav";
try
{
    player.Load();
    player.Play();
}
catch (System.IO.FileNotFoundException err)
{
    // An error will occur here if the file can't be found.
}
catch (FormatException err)
{
    // A FormatException will occur here if the file doesn't
    // contain valid WAV audio.
}
```

So far, the code has assumed that the audio is present in the same directory as the compiled application. However, you don't need to load the SoundPlayer audio from a file. If you've created small sounds that are played at several points in your application, it may make more sense to embed the sound files into your compiled assembly as a binary resource (not to be confused with declarative resources, which are the resources you define in XAML markup). This technique, which was discussed in Chapter 11, works just as well with sound files as it does with images. For example, if you add the ding.wav audio file with the resource name Ding (just browse to the Properties ➤ Resources node in the Solution Explorer and use the designer support), you could use this code to play it:

```
SoundPlayer player = new SoundPlayer();
player.Stream = Properties.Resources.Ding;
player.Play();
```

■ **Note** The SoundPlayer class doesn't deal well with large audio files, because it needs to load the entire file into memory at once. You might think that you can resolve this problem by submitting a large audio file in smaller chunks, but the SoundPlayer wasn't designed with this technique in mind. There's no easy way to synchronize the SoundPlayer so that it plays multiple audio snippets one after the other, because it doesn't provide any sort of queuing feature. Each time you call PlaySync() or Play(), the current audio playback stops. Workarounds are possible, but you'll be far better off using the MediaElement class discussed later in this chapter.

The SoundPlayerAction

The SoundPlayerAction makes it more convenient to use the SoundPlayer class. The SoundPlayerAction class derives from TriggerAction (Chapter 11), which allows you to use it in response to any event.

Here's a button that uses a SoundPlayerAction to connect the Click event to a sound. The trigger is wrapped in a style that you could apply to multiple buttons (if you pulled it out of the button and placed it in a Resources collection).

```
<Button>
  <Button.Content>Play Sound</Button.Content>
  <Button.Style>
    <Style>
      <Style.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
          <EventTrigger.Actions>
            <SoundPlayerAction Source="test.wav"></SoundPlayerAction>
          </EventTrigger.Actions>
        </EventTrigger>
      </Style.Triggers>
    </Style>
  </Button.Style>
</Button>
```

When using the SoundPlayerAction, the sound is always played asynchronously.

System Sounds

One of the shameless frills of the Windows operating system is its ability to map audio files to specific system events. Along with SoundPlayer, WPF also includes a System.Media.SystemSounds class that allows you to access the most common of these sounds and use them in your own applications. This technique works best if all you want is a simple chime to indicate the end of a long-running operation or an alert sound to indicate a warning condition.

Unfortunately, the SystemSounds class is based on the MessageBeep Win32 API, and as a result, it provides access only to the following generic system sounds:

- Asterisk
- Beep
- Exclamation
- Hand
- Question

The SystemSounds class provides a property for each of these sounds, which returns a SystemSound object you can use to play the sound through its Play() method. For example, to sound a beep in your code, you simply need to execute this line of code:

```
SystemSounds.Beep.Play();
```

To configure what WAV files are used for each sound, head to the Control Panel and double-click the Sound icon.

The MediaPlayer

The SoundPlayer, SoundPlayerAction, and SystemSounds classes are easy to use but relatively underpowered. In today's world, it's much more common to use compressed MP3 audio for everything except the simplest of sounds, instead of the original WAV format. But if you want to play MP3 audio or MPEG video, you need to turn to two classes: MediaPlayer and MediaElement. Both classes depend on key pieces of technology that are provided through Windows Media Player.

The MediaPlayer class (found in the WPF-specific System.Windows.Media namespace) is the WPF equivalent to the SoundPlayer class. Although it's clearly not as lightweight, it works in a similar way—namely, you create a MediaPlayer object, call the Open() method to load your audio file, and call Play() to begin playing it asynchronously. (There's no option for synchronous playback.) Here's a bare-bones example:

```
private MediaPlayer player = new MediaPlayer();

private void cmdPlayWithMediaPlayer_Click(object sender, RoutedEventArgs e)
{
    player.Open(new Uri("test.mp3", UriKind.Relative));
    player.Play();
}
```

There are a few important details to notice in this example:

- The MediaPlayer is created outside the event handler, so it lives for the lifetime of the window. That's because the MediaPlayer.Close() method is called when the MediaPlayer object is disposed from memory. If you create a MediaPlayer object in the event handler, it will be released from memory almost immediately and probably garbage collected shortly after, at which point the Close() method will be called and playback will be halted.

■ **Tip** You should create a Window.Unloaded event handler to call Close() to stop any currently playing audio when the window is closed.

- You supply the location of your file as a URI. Unfortunately, this URI doesn't use the application pack syntax that you learned about in Chapter 7, so it's not possible to embed an audio file and play it by using the MediaPlayer class. This limitation exists because the MediaPlayer class is built on functionality that's not native to WPF—instead, it's provided by a distinct, unmanaged component of the Windows Media Player.
- There's no exception-handling code. Irritatingly, the Open() and Play() methods don't throw exceptions (the asynchronous load and playback process is partly to blame). Instead, it's up to you to handle the MediaOpened and MediaFailed events if you want to determine whether your audio is being played.

The MediaPlayer is fairly straightforward but still more capable than SoundPlayer. It provides a small set of useful methods, properties, and events. Table 26-1 has the full list.

Table 26-1. *Key MediaPlayer Members*

Member	Description
Balance	Sets the balance between the left and right speaker as a number from –1 (left speaker only) to 1 (right speaker only).
Volume	Sets the volume as a number from 0 (completely muted) to 1 (full volume). The default value is 0.5.
SpeedRatio	Sets a speed multiplier to play audio (or video) at faster than normal speed. The default value of 1 is normal speed, while 2 is two-times normal speed, 10 is ten-times speed, 0.5 is half-times speed, and so on. You can use any positive double value.
HasAudio and HasVideo	Indicates whether the currently loaded media file includes audio or video, respectively. To show video, you need to use the MediaElement class described in the next section.
NaturalDuration, NaturalVideoHeight, and NaturalVideoWidth	Indicates the play duration at normal speed and the size of the video window. (As you'll discover later, you can scale or stretch a video to fit different window sizes.)
Position	A TimeSpan indicating the current location in the media file. You can set this property to skip to a specific time position.
DownloadProgress and BufferingProgress	Indicates the percentage of a file that has been downloaded (useful if the Source is a URL pointing to a web or remote computer) or buffered (if the media file you're using is encoded in a streaming format so it can be played before it's entirely downloaded). The percentage is represented as a number from 0 to 1.

Member	Description
Clock	Gets or sets the MediaClock that's associated with this player. The MediaClock is used only when you're synchronizing audio to a timeline (in much the same way that you learned to synchronize an animation to a timeline in Chapter 15). If you're using the methods of the MediaPlayer to perform manual playback, this property is null.
Open()	Loads a new media file.
Play()	Begins playback. Has no effect if the file is already being played.
Pause()	Pauses playback but doesn't change the position. If you call Play() again, playback will begin at the current position. Has no effect if the audio is not playing.
Stop()	Stops playback and resets the position to the beginning of the file. If you call Play() again, playback will begin at the beginning of the file. Has no effect if the audio has already been stopped.

Using these members, you could build a basic but full-featured media player. However, WPF programmers usually use another quite similar element, which is defined in the next section: the MediaElement class.

The MediaElement

The MediaElement is a WPF element that wraps all the functionality of the MediaPlayer class. Like all elements, the MediaElement is placed directly in your user interface. If you're using the MediaElement to play audio, this fact isn't important, but if you're using the MediaElement for video, you place it where the video window should appear.

A simple MediaElement tag is all you need to play a sound. For example, if you add this markup to your user interface:

```
<MediaElement Source="test.mp3"></MediaElement>
```

the test.mp3 audio will be played as soon as it's loaded (which is more or less as soon as the window is loaded).

Playing Audio Programmatically

Usually, you'll want the ability to control playback more precisely. For example, you might want it to be triggered at a specific time, repeated indefinitely, and so on. One way to achieve this result is to use the methods of the MediaElement class at the appropriate time.

The startup behavior of the MediaElement is determined by its LoadedBehavior property, which is one of the few properties that the MediaElement class adds, which isn't found in the MediaPlayer class. The LoadedBehavior takes any value from the MediaState enumeration. The default value is Play, but you can also use Manual, in which case the audio file is loaded, and your code takes responsibility for starting the playback at the right time. Another option is Pause, which also suspends playback but doesn't allow you to use the playback methods. (Instead, you'll need to start playback by using triggers and a storyboard, as described in the next section.)

■ **Note** The MediaElement class also provides an UnloadedBehavior property, which determines what should happen when the element is unloaded. In this case, Close is really the only sensible choice, because it closes the file and releases all system resources.

So to play audio programmatically, you must begin by changing the LoadedBehavior, as shown here:

```
<MediaElement Source="test.mp3" LoadedBehavior="Manual" Name="media"></MediaElement>
```

You must also choose a name so that you can interact with the media element in code. Generally, interaction consists of the straightforward Play(), Pause(), and Stop() methods. You can also set Position to move through the audio. Here's a simple event handler that seeks to the beginning and starts playback:

```
private void cmdPlay_Click(object sender, RoutedEventArgs e)
{
    media.Position = TimeSpan.Zero;
    media.Play();
}
```

If this code runs while playback is already underway, the first line will reset the position to the beginning, and playback will continue from that point. The second line will have no effect, because the media file is already being played. If you try to use this code on a MediaElement that doesn't have the LoadedBehavior property set to Manual, you'll receive an exception.

■ **Note** In a typical media player, you can trigger basic commands such as play, pause, and stop in more than one way. Obviously, this is a great place to use the WPF command model. In fact, there's a command class that already includes some handy infrastructure, the System.Windows.Input.MediaCommands class. However, the MediaElement does not have any default command bindings that support the MediaCommands class. In other words, it's up to you to write the event-handling logic that implements each command and calls the appropriate MediaElement method. The savings to you is that multiple user interface elements can be hooked up to the same command, reducing code duplication. Chapter 9 has more about commands.

Handling Errors

The MediaElement doesn't throw an exception if it can't find or load a file. Instead, it's up to you to handle the MediaFailed event. Fortunately, this task is easy. Just tweak your MediaElement tag:

```
<MediaElement ... MediaFailed="media_MediaFailed"></MediaElement>
```

And, in the event handler, use the ExceptionRoutedEventArgs.ErrorException property to get an exception object that describes the problem:

```
private void media_MediaFailed(object sender, ExceptionRoutedEventArgs e)
{
    lblErrorText.Content = e.ErrorException.Message;
}
```

Playing Audio with Triggers

So far, you haven't received any advantage by switching from the MediaPlayer to the MediaElement class (other than support for video, which is discussed later in this chapter). However, by using a MediaElement, you also gain the ability to control audio declaratively, through XAML markup rather than code. You do this by using triggers and storyboards, which you first saw when you considered animation in Chapter 15. The only new ingredient is the MediaTimeline, which controls the timing of your audio or video file and works with MediaElement to coordinate its playback. MediaTimeline derives from Timeline and adds a Source property that identifies the audio file you want to play.

The following markup demonstrates a simple example. It uses the BeginStoryboard action to begin playing a sound when the mouse clicks a button. (Obviously, you could respond equally well to other mouse and keyboard events.)

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Size="Auto"></RowDefinition>
   <RowDefinition Size="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <MediaElement x:Name="media"></MediaElement>

  <Button>
    <Button.Content>Click me to hear a sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <MediaTimeline Source="soundA.wav"
             Storyboard.TargetName="media"></MediaTimeline>
          </Storyboard>
        </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
</Grid>
```

Because this example plays audio, the positioning of the MediaElement isn't important. In this example, it's placed inside a Grid, behind a Button. (The ordering isn't important, because the MediaElement won't have any visual appearance at runtime.) When the button is clicked, a Storyboard is created with a MediaTimeline. Notice that the source isn't specified in the MediaElement.Source property. Instead, the source is passed along through the MediaTimeline.Source property.

■ **Note** When you use MediaElement as the target of a MediaTimeline, it no longer matters what you set the LoadedBehavior and UnloadedBehavior to. Once you use a MediaTime, your audio or video is driven by a WPF animation clock (technically, an instance of the MediaClock class, which is exposed through the MediaElement.Clock property).

You can use a single Storyboard to control the playback of a single MediaElement—in other words, not only stopping it but also pausing, resuming, and stopping it at will. For example, consider the extremely simple four-button media player shown in Figure 26-1.

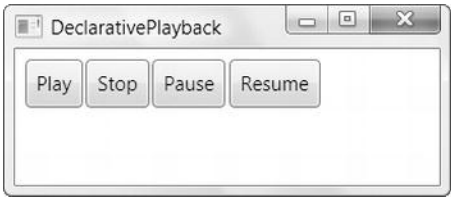


Figure 26-1. *A window for controlling playback*

This window uses a single MediaElement, MediaTimeline, and Storyboard. The Storyboard and MediaTimeline are declared in the Window.Resources collection:

```
<Window.Resources>
  <Storyboard x:Key="MediaStoryboardResource">
    <MediaTimeline Storyboard.TargetName="media" Source="test.mp3"></MediaTimeline>
    </Storyboard>
</Window.Resources>
```

The only challenge is that you must remember to define all the triggers for managing the storyboard in one collection. You can then attach them to the appropriate controls by using the EventTrigger. SourceName property.

In this example, the triggers are all declared inside the StackPanel that holds the buttons. Here are the triggers and the buttons that use them to manage the audio:

```
<StackPanel Orientation="Horizontal">
  <StackPanel.Triggers>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdPlay">
      <EventTrigger.Actions>
        <BeginStoryboard Name="MediaStoryboard"
         Storyboard="{StaticResource MediaStoryboardResource}"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdStop">
      <EventTrigger.Actions>
        <StopStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdPause">
      <EventTrigger.Actions>
        <PauseStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdResume">
      <EventTrigger.Actions>
        <ResumeStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
      </EventTrigger>
    </StackPanel.Triggers>

    <MediaElement  Name="media"></MediaElement>
    <Button Name="cmdPlay">Play</Button>
    <Button Name="cmdStop">Stop</Button>
    <Button Name="cmdPause">Pause</Button>
    <Button Name="cmdResume">Resume</Button>
  </StackPanel>
```

Notice that even though the implementation of MediaElement and MediaPlayer allows you to resume playback after pausing by calling Play(), the Storyboard doesn't work in the same way. Instead, a separate ResumeStoryboard action is required. If this isn't the behavior you want, you can consider adding some code for your play button instead of using the declarative approach.

■ **Note** The downloadable code samples for this chapter include a declarative media player window and a more flexible code-driven media player window.

Playing Multiple Sounds

Although the previous example showed you how to control the playback of a single media file, there's no reason you can't extend it to play multiple audio files. The following example includes two buttons, each of which plays its own sound. When a button is clicked, a new Storyboard is created, with a new MediaTimeline, which is used to play a different audio file through the same MediaElement.

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Size="Auto"></RowDefinition>
   <RowDefinition Size="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <MediaElement x:Name="media"></MediaElement>

  <Button>
    <Button.Content>Click me to hear a sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <MediaTimeline Source="soundA.wav"
             Storyboard.TargetName="media"></MediaTimeline>
          </Storyboard>
        </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
  <Button Grid.Row="1">
    <Button.Content >Click me to hear a different sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <MediaTimeline Source="soundB.wav"
               Storyboard.TargetName="media"></MediaTimeline>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
</Grid>
```

In this example, if you click both buttons in quick succession, you'll see that the second sound interrupts the playback of the first. This is a consequence of using the same MediaElement for both timelines. A slicker (but more resource-heavy) approach is to use a separate MediaElement for each button and point the MediaTimeline to the corresponding MediaElement. (In this case, you can specify the Source directly in the MediaElement tag, because it doesn't change.) Now, if you click both buttons in quick succession, both sounds will play at the same time.

The same applies to the MediaPlayer class—if you want to play multiple audio files, you need multiple MediaPlayer objects. If you decide to use the MediaPlayer or MediaElement with code, you have the opportunity to use more-intelligent optimization that allows exactly two simultaneous sounds, but no more. The basic technique is to define two MediaPlayer objects and flip between them each time you play a new sound. (You can keep track of which object you used last by using a Boolean variable.) To make this technique really effortless, you can store the audio file names in the Tag property of the appropriate element, so all your event-handling code needs to do is find the right MediaPlayer to use, set its Source property, and call its Play() method.

Changing Volume, Balance, Speed, and Position

The MediaElement exposes the same properties as the MediaPlayer (detailed in Table 26-1) for controlling the volume, the balance, the speed, and the current position in the media file. Figure 26-2 shows a simple window that extends the sound player example from Figure 26-1 with additional controls for adjusting these details.

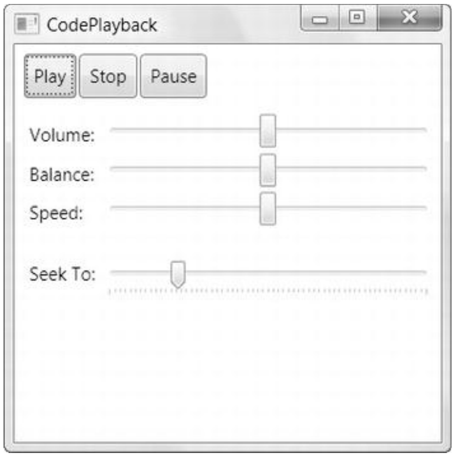


Figure 26-2. Controlling more playback details

The Volume and Balance sliders are the easiest to wire up. Because Volume and Balance are dependency properties, you can connect the slider to the MediaElement with a two-way binding expression. Here's what you need:

```
<Slider Grid.Row="1" Minimum="0" Maximum="1"
 Value="{Binding ElementName=media, Path=Volume, Mode=TwoWay}"></Slider>
<Slider Grid.Row="2" Minimum="-1" Maximum="1"
 Value="{Binding ElementName=media, Path=Balance, Mode=TwoWay}"></Slider>
```

Although two-way data-binding expressions incur slightly more overhead, they ensure that if the MediaElement properties are changed some other way, the slider controls remain synchronized.

The SpeedRatio property can be connected in the same way:

```
<Slider Grid.Row="3" Minimum="0" Maximum="2"
  Value="{Binding ElementName=media, Path=SpeedRatio}"></Slider>
```

However, this has a few quirks. First, SpeedRatio isn't used in a clock-driven audio (one that uses a MediaTimeline). To use it, you need to set the LoadedBehavior property of SpeedRatio to Manual and take control of its playback manually through the playback methods.

■ **Tip** If you're using a MediaTimeline, you can get the same effect from the SetStoryboardSpeedRatio action as you get from setting the MediaElement.SpeedRatio property. You learned about these details in Chapter 15.

Second, SpeedRatio isn't a dependency property, and WPF doesn't receive change notifications when it's modified. That means if you include code that modifies the SpeedRatio property, the slider won't be updated accordingly. (One workaround is to modify the slider in your code, rather than modify the MediaElement directly.)

■ **Note** Changing the playback speed of audio can distort the audio and cause sound artifacts, such as echoes.

The last detail is the current position, which is provided by the Position property. Once again, the MediaElement needs to be in Manual mode before you can set the Position property, which means you can't use the MediaTimeline. (If you're using a MediaTimeline, consider using the BeginStoryboard action with an Offset to the position you want, as described in Chapter 15.)

To make this work, you don't use any data binding in the slider:

```
<Slider Minimum="0" Name="sliderPosition"
  ValueChanged="sliderPosition_ValueChanged"></Slider>
```

You use code like this to set up the position slider when you open a media file:

```
private void media_MediaOpened(object sender, RoutedEventArgs e)

{
    sliderPosition.Maximum = media.NaturalDuration.TimeSpan.TotalSeconds;
}
```

You can then jump to a specific position when the slider tab is moved:

```
private void sliderPosition_ValueChanged(object sender, RoutedEventArgs e)
{
    // Pausing the player before moving it reduces audio "glitches"
    // when the value changes several times in quick succession.
    media.Pause();
    media.Position = TimeSpan.FromSeconds(sliderPosition.Value);
    media.Play();
}
```

The drawback here is that the slider isn't updated as the media advances. If you want this feature, you need to cook up a suitable workaround (for example, a DispatcherTimer that triggers a periodic check while playback is taking place and updates the slider then). The same is true if you're using the MediaTimeline. For various reasons, you can't bind directly to the MediaElement.Clock information. Instead, you'll need to handle the Storyboard.CurrentTimeInvalidated event, as demonstrated in the AnimationPlayer example in Chapter 15.

Synchronizing an Animation with Audio

In some cases, you may want to synchronize another animation to a specific point in a media file (audio or video). For example, if you have a lengthy audio file that features a person describing a series of steps, you might want to fade in different images after each pause.

Depending on your needs, this design may be overly complex, and you may be able to achieve better performance and simpler design by segmenting the audio into separate files. That way, you can load the new audio and perform the correlated action all at once, simply by responding to the MediaEnded event. In other situations, you need to synchronize something with continuous, unbroken playback of a media file.

One technique that allows you to pair playback with other actions is key-frame animation (which was introduced in Chapter 16). You can then wrap this key-frame animation and your MediaTimeline into a single storyboard. That way, you can supply specific time offsets for your animation, which will then correspond to precise times in the audio file. In fact, you can even use a third-party program that allows you to annotate audio and export a list of important times. You can then use this information to set up the time for each key frame.

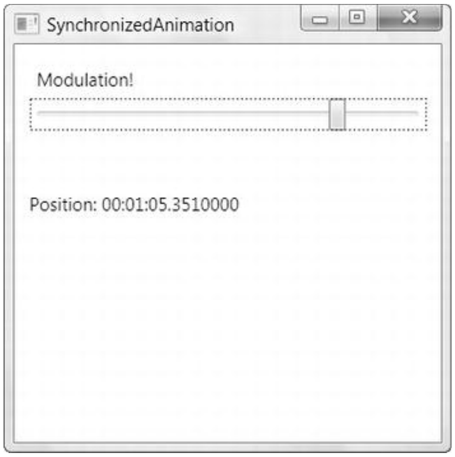
When using key-frame animation, it's important to set the Storyboard.SlipBehavior property to Slip. This specifies that your key-frame animation should not creep ahead of the MediaTimeline, if the media file is delayed. This is important because the MediaTimeline could be delayed by buffering (if it's being streamed from a server) or, more commonly, by load time.

The following markup demonstrates a basic example of an audio file with two synchronized animations. The first varies the text in a label as specific parts of the audio file are reached. The second shows a small circle halfway through the audio and pulses it in time to the beat by varying the value of the Opacity property.

```
<Window.Resources>
  <Storyboard x:Key="Board" SlipBehavior="Slip">
    <MediaTimeline Source="sq3gm1.mid"
     Storyboard.TargetName="media"/>

      <StringAnimationUsingKeyFrames
       Storyboard.TargetName="lblAnimated"
       Storyboard.TargetProperty="(Label.Content)" FillBehavior="HoldEnd">
        <DiscreteStringKeyFrame Value="First note..." KeyTime="0:0:3.4" />
        <DiscreteStringKeyFrame Value="Introducing the main theme..."
         KeyTime="0:0:5.8" />
        <DiscreteStringKeyFrame Value="Irritating bass begins..."
         KeyTime="0:0:28.7" />
        <DiscreteStringKeyFrame Value="Modulation!" KeyTime="0:0:53.2" />
        <DiscreteStringKeyFrame Value="Back to the original theme."
         KeyTime="0:1:8" />
      </StringAnimationUsingKeyFrames>

      <DoubleAnimationUsingKeyFrames
        Storyboard.TargetName="ellipse"
        Storyboard.TargetProperty="Opacity" BeginTime="0:0:29.36"
        RepeatBehavior="30x">
      <LinearDoubleKeyFrame Value="1" KeyTime="0:0:0" />
      <LinearDoubleKeyFrame Value="0" KeyTime="0:0:0.64" />
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</Window.Resources>

<Window.Triggers>
  <EventTrigger RoutedEvent="MediaElement.Loaded">
    <EventTrigger.Actions>
      <BeginStoryboard Name="mediaStoryboard" Storyboard="{StaticResource Board}">
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
</Window.Triggers>
```

To make this example even more interesting, it also includes a slider that allows you to change your position. You'll see that even if you change the position by using the slider, the three animations are adjusted automatically to the appropriate point by the MediaTimeline. (The slider is kept synchronized by using the Storyboard.CurrentTimeInvalidated event, and the ValueChanged event is handled to seek to a new position after the user drags the slider thumb. You saw both of these techniques in Chapter 15, with the AnimationPlayer example.)

Figure 26-3 shows the program in action.



Modulation!

Position: 00:01:05.3510000

Figure 26-3. *Synchronized animations*

Playing Video

Everything you've learned about using the MediaElement class applies equally well when you use a video file instead of an audio file. As you'd expect, the MediaElement class supports all the video formats that are supported by Windows Media Player. Although support depends on the codecs you've installed, you can't count on basic support for WMV, MPEG, and AVI files.

The key difference with video files is that the visual and layout-related properties of the MediaElement are suddenly important. Most important, the Stretch and StretchDirection properties determine how the video window is scaled to fit its container (and work in the same way as the Stretch and StretchDirection properties that you learned about on all Shape-derived classes). When setting the Stretch value, you can use None to keep the native size, Uniform to stretch it to fit its container without changing its aspect ratio, Uniform to stretch it to fit its container in both dimensions (even if that means stretching the picture), and UniformToFill to resize the picture to fit the largest dimension of its container while preserving its aspect ratio (which guarantees that part of the video window will be clipped out if the container doesn't have the same aspect ratio as the video).

■ **Tip** The MediaElement's preferred size is based on the native video dimensions. For example, if you create a MediaElement with a Stretch value of Uniform (the default) and place it inside a Grid row with a Height value of Auto, the row will be sized just large enough to keep the video at its standard size, so no scaling is required.

Video Effects

Because the MediaElement works like any other WPF element, you have the ability to manipulate it in some surprising ways. Here are some examples:

- You can use a MediaElement as the content inside a content control, such as a button.

- You can set the content for thousands of content controls at once with multiple MediaElement objects—although your CPU probably won't bear up very well under the strain.
- You can also combine video with transformations through the LayoutTransform or RenderTransform property. This allows you to move your video window, stretch it, skew it, or rotate it.

■ **Tip** Generally, RenderTransform is preferred over LayoutTransform for the MediaElement, because it's lighter weight. It also takes the value of the handy RenderTransformOrigin property into account, allowing you to use relative coordinates for certain transforms (such as rotation).

- You can set the Clipping property of the MediaElement to cut down the video window to a specific shape or path and show only a portion of the full window.
- You can set the Opacity property to allow content behind your video window to show through. In fact, you can even stack multiple semitransparent video windows on top of each other (with dire consequences for performance).
- You can use animation to change a property of the MediaElement (or one of its transforms) dynamically.
- You can copy the current content of the video window to another place in your user interface by using a VisualBrush, which allows you to create specific effects such as reflection.
- You can place a video window on a three-dimensional surface and use animation to move the video window as the video is being played (as described in Chapter 27).

For example, the following markup creates the reflection effect shown in Figure 26-4. It does so by creating a Grid with two rows. The top row holds a MediaElement that plays a video file. The bottom row holds a Rectangle that's painted with a VisualBrush. The trick is that the VisualBrush takes its content from the video window above it, using a binding expression. The video content is then flipped over by using the RelativeTransform property and then faded out gradually toward the bottom by using an OpacityMask gradient.

```
<Grid Margin="15" HorizontalAlignment="Center">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Border BorderBrush="DarkGray" BorderThickness="1" CornerRadius="2">
    <MediaElement x:Name="video" Source="test.mpg" LoadedBehavior="Manual"
     Stretch="Fill"></MediaElement>
  </Border>

  <Border Grid.Row="1" BorderBrush="DarkGray" BorderThickness="1" CornerRadius="2">
```

```
    <Rectangle VerticalAlignment="Stretch" Stretch="Uniform">
    <Rectangle.Fill>
      <VisualBrush Visual="{Binding ElementName=video}">
        <VisualBrush.RelativeTransform>
          <ScaleTransform ScaleY="-1" CenterY="0.5"></ScaleTransform>
        </VisualBrush.RelativeTransform>
      </VisualBrush>
    </Rectangle.Fill>

    <Rectangle.OpacityMask>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="Black" Offset="0"></GradientStop>
        <GradientStop Color="Transparent" Offset="0.6"></GradientStop>
      </LinearGradientBrush>
    </Rectangle.OpacityMask>
    </Rectangle>
  </Border>
</Grid>
```

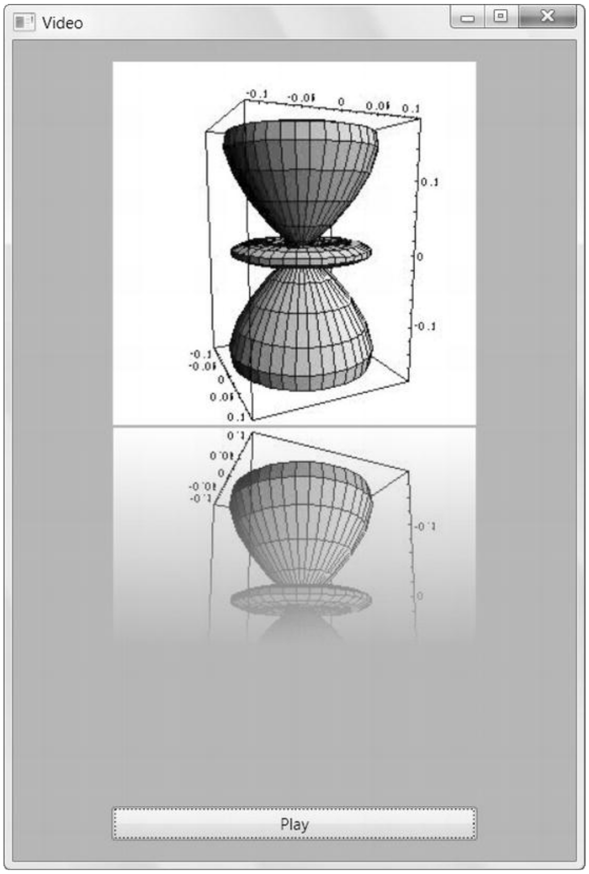


Figure 26-4. *Reflected video*

This example performs fairly well. The reflection effect has a similar rendering overhead to two video windows, because each frame must be copied to the lower rectangle. In addition, each frame needs to be flipped and faded to create the reflection effect. (WPF uses an intermediary rendering surface to perform these transformations.) But on a modern computer, the extra overhead is barely noticeable.

This isn't the case with other video effects. In fact, video is one of the few areas in WPF where it's extremely easy to overtask the CPU and create interfaces that perform poorly. Average computers can't handle more than a few simultaneous video windows (depending, obviously, on the size of your video file—higher resolutions and higher frame rates obviously mean more data, which is more time-consuming to process).

THE VIDEODRAWING CLASS

WPF includes a VideoDrawing class that derives from the Drawing class you learned about in Chapter 13. The VideoDrawing can be used to create a DrawingBrush, which can then be used to fill the surface of an element, creating much the same effect as demonstrated in the previous example with the VisualBrush.

However, there's a difference that may make the VideoDrawing approach more efficient. That's because VideoDrawing uses the MediaPlayer class, while the VisualBrush approach requires the use of the MediaElement class. The MediaPlayer class doesn't need to manage layout, focus, or any other element details, so it's more lightweight than the MediaElement class. In some situations, using the VideoDrawing and DrawingBrush instead of the MediaElement and VisualBrush can avoid the need for an intermediary rendering surface and thus improve performance (although in my testing, I didn't notice much of a difference between the two approaches).

Using the VideoDrawing takes a fair bit more work, because the MediaPlayer needs to be started in code (by calling its Play() method). Usually, you'll create all three objects—the MediaPlayer, VideoDrawing, and DrawingBrush—in code. Here's a basic example that paints the video on the background of the current window:

```
// Create the MediaPlayer.
MediaPlayer player = new MediaPlayer();
player.Open(new Uri("test.mpg", UriKind.Relative));

// Create the VideoDrawing.
VideoDrawing videoDrawing = new VideoDrawing();
videoDrawing.Rect = new Rect(150, 0, 100, 100);
videoDrawing.Player = player;

// Assign the DrawingBrush.
DrawingBrush brush = new DrawingBrush(videoDrawing);
this.Background = brush;

// Start playback.
player.Play();
```

The downloadable examples for this chapter include a demonstration of video effects: an animation that rotates a video window as it plays. The need to wipe out each video frame and redraw a new one at a slightly different angle runs relatively well on modern video cards but causes a noticeable flicker on lower-tier cards. If in doubt, you should profile your user interface plans on a lesser-powered computer to see whether they stand up and should provide a way to opt out of the more complex effects your application provides or gracefully disable them on lower-tier cards.

Speech

Audio and video support is a core pillar of the WPF platform. However, WPF also includes libraries that wrap two less commonly used multimedia features: speech synthesis and speech recognition.

Both of these features are supported through classes in the System.Speech.dll assembly. By default, Visual Studio doesn't add a reference to this assembly in a new WPF project, so it's up to you to add one to your project.

■ **Note** Speech is a peripheral part of WPF. Although the speech support is technically considered to be part of WPF and was released with WPF in the .NET Framework 3.0, the speech namespaces start with System.Speech, not System.Windows.

Speech Synthesis

Speech synthesis is a feature that generates spoken audio based on text you supply. Speech synthesis isn't built into WPF—instead, it's a Windows accessibility feature. System utilities such as Narrator, a lightweight screen reader included with Windows, use speech synthesis to help blind users navigate basic dialog boxes. More generally, speech synthesis can be used to create audio tutorials and spoken instructions, although prerecorded audio provides better quality.

■ **Note** Speech synthesis makes sense when you need to create audio for dynamic text—in other words, when you don't know at compile time what words need to be spoken at runtime. But if the audio is fixed, prerecorded audio is easier to use, is more efficient, and sounds better. The only other reason you might consider speech synthesis is if you need to narrate a huge amount of text and prerecording it all would be impractical.

Modern versions of Windows have speech synthesis built in. They use a relatively natural female voice named Anna, although you can download and install additional voices.

Playing speech is deceptively simple. All you need to do is create an instance of the SpeechSynthesizer class from the System.Speech.Synthesis namespace and call its Speak() method with a string of text. Here's an example:

```
SpeechSynthesizer synthesizer = new SpeechSynthesizer();
synthesizer.Speak("Hello, world");
```

When using this approach—passing plain text to the SpeechSynthesizer—you give up a fair bit of control. You may run into words that aren't pronounced properly, emphasized appropriately, or spoken at the correct speed. To get more control over spoken text, you need to use the PromptBuilder class to construct a definition of the speech. Here's how you could replace the earlier example with completely equivalent code that uses the PromptBuilder:

```
PromptBuilder prompt = new PromptBuilder();
prompt.AppendText("Hello, world");

SpeechSynthesizer synthesizer = new SpeechSynthesizer();
synthesizer.Speak(prompt);
```

This code doesn't provide any advantage. However, the PromptBuilder class has a number of other methods that you can use to customize the way text is spoken. For example, you can emphasize a specific word (or several words) by using an overloaded version of the AppendText() method that takes a value from the PromptEmphasis enumeration. Although the precise effect of emphasizing a word depends on the voice you're using, the following code stresses the *are* in the sentence "How are you?"

```
PromptBuilder prompt = new PromptBuilder();
prompt.AppendText("How ");
prompt.AppendText("are ", PromptEmphasis.Strong);
prompt.AppendText("you");
```

The AppendText() method has two other overloads—one that takes a PromptRate value that lets you increase or decrease speed and one that takes a PromptVolume value that lets you increase or decrease the volume.

If you want to change more than one of these details at the same time, you need to use a PromptStyle object. The PromptStyle wraps PromptEmphasis, PromptRate, and PromptVolume values. You can supply values for all three details or just the one or two you want to use.

To use a PromptStyle object, you call PromptBuilder.BeginStyle(). The PromptStyle you've created is then applied to all the spoken text until you can EndStyle(). Here's a revised example that uses emphasis and a change in speed to put the stress on the word *are*:

```
PromptBuilder prompt = new PromptBuilder();
prompt.AppendText("How ");
PromptStyle style = new PromptStyle();
style.Rate = PromptRate.ExtraSlow;
style.Emphasis = PromptEmphasis.Strong;
prompt.StartStyle(style);
prompt.AppendText("are ");
prompt.EndStyle();
prompt.AppendText("you");
```

■ **Note** If you call BeginStyle(), you must call EndStyle() later in your code. If you fail to do so, you'll receive a runtime error.

The PromptEmphasis, PromptRate, and PromptVolume enumerations provide relatively crude ways to influence a voice. There's no way to get finer-grained control or introduce nuances or subtler specific speech patterns into spoken text. However, the PromptBuilder includes an AppendTextWithHint() method that allows you to deal with telephone numbers, dates, times, and words that need to be spelled out. You supply your choice by using the SayAs enumeration. Here's an example:

```
prompt.AppendText("The word laser is spelled ");
prompt.AppendTextWithHint("laser", SayAs.SpellOut);
```

This produces the narration "The word laser is spelled l-a-s-e-r."

Along with the AppendText() and AppendTextWithHint() methods, the PromptBuilder also includes a small collection of additional methods for adding ordinary audio to the stream (AppendAudio()), creating pauses of a specified duration (AppendBreak()), switching voices (StartVoice() and EndVoice()), and speaking text according to a specified phonetic pronunciation (AppendTextWithPronounciation()).

The PromptBuilder is really a wrapper for the Speech Synthesis Markup Language (SSML) standard, which is described at `www.w3.org/TR/speech-synthesis`. As such, it shares the limitations of that standard.

As you call the PromptBuilder methods, the corresponding SSML markup is generated behind the scenes. You can see the final SSML representation of your code by calling PromptBuilder.ToXml() at the end of your work, and you can call PromptBuilder.AppendSsml() to take existing SSML markup and read it into your prompt.

Speech Recognition

Speech recognition is a feature that translates user-spoken audio into text. As with speech synthesis, speech recognition is a built-in feature of the Windows operating system.

■ **Note** If speech recognition isn't currently running, the speech recognition toolbar will appear when you instantiate the SpeechRecognizer class. If you attempt to instantiate the SpeechRecognizer class and you haven't configured speech recognition for your voice, Windows will automatically start a wizard that leads you through the process.

Speech recognition is also a Windows accessibility feature. For example, it allows users with disabilities to interact with common controls by voice. Speech recognition also allows hands-free computer use, which is useful in certain environments.

The most straightforward way to use speech recognition is to create an instance of the SpeechRecognizer class from the System.Speech.Recognition namespace. You can then attach an event handler to the SpeechRecognized event, which is fired whenever spoken words are successfully converted to text:

```
SpeechRecognizer recognizer = new SpeechRecognizer();
recognizer.SpeechRecognized += recognizer_SpeechRecognized;
```

You can then retrieve the text in the event handler from the SpeechRecognizedEventArgs.Result property:

```
private void recognizer_SpeechRecognized(object sender, SpeechRecognizedEventArgs e)
{
    MessageBox.Show("You said:" + e.Result.Text);
}
```

The SpeechRecognizer wraps a COM object. To avoid unseemly glitches, you should declare it as a member variable in your window class (so the object remains alive as long as the window exists) and you should call its `Dispose()` method when the window is closed (to remove your speech-recognition hooks).

■ **Note** The SpeechRecognizer class raises a sequence of events when audio is detected. First, SpeechDetected is raised if the audio appears to be speech. SpeechHypothesized then fires one or more times, as the words are tentatively recognized. Finally, the SpeechRecognizer raises a SpeechRecognized if it can successfully process the text or a SpeechRecognitionRejected event if it cannot. The SpeechRecognitionRejected event includes information about what the SpeechRecognizer believes the spoken input might have been, but its confidence level is not high enough to accept the input.

It's generally not recommended that you use speech recognition in this fashion. That's because WPF has its own UI Automation feature that works seamlessly with the speech-recognition engine. When configured, it allows users to enter text in text controls and trigger button controls by speaking their automation names. However, you could use the SpeechRecognition class to add support for more-specialized commands to support specific scenarios. You do this by specifying a *grammar* based on the Speech Recognition Grammar Specification (SRGS).

The SRGS grammar identifies what commands are valid for your application. For example, it may specify that commands can use only one of a small set of words (*in* or *off*) and that these words can be used only in specific combinations (*blue on*, *red on*, *blue off*, and so on).

You can construct an SRGS grammar in two ways. You can load it from an SRGS document, which specifies the grammar rules by using an XML-based syntax. To do this, you need to use the SrgsDocument from the System.Speech.Recognition.SrgsGrammar namespace:

```
SrgsDocument doc = new SrgsDocument("app_grammar.xml");
Grammar grammar = new Grammar(doc);
recognizer.LoadGrammar(grammar);
```

Alternatively, you can construct your grammar declaratively by using the GrammarBuilder. The GrammarBuilder plays an analogous role to that of the PromptBuilder you considered in the previous section—it allows you to append grammar rules bit by bit to create a complete grammar. For example, here's a declaratively constructed grammar that accepts two-word input, where the first words has five possibilities and the second word has just two:

```
GrammarBuilder grammar = new GrammarBuilder();
grammar.Append(new Choices("red", "blue", "green", "black", "white"));
grammar.Append(new Choices("on", "off"));

recognizer.LoadGrammar(new Grammar(grammar));
```

This markup allows commands such as *red on* and *green off*. Alternate input such as *yellow on* or *on red* won't be recognized.

The Choices object represents the SRGS *one-of* rule, which allows the user to speak one word out of a range of choices. It's the most versatile ingredient when building a grammar. Several more overloads to the GrammarBuilder.Append() method accept different input. You can pass an ordinary string, in which case the grammar will require the user to speak exactly that word. You can pass a string followed by a value from the SubsetMatchingMode enumeration to require the user to speak some part of a word or phrase. Finally, you can pass a string followed by a number of minimum and maximum repetitions. This allows the grammar to ignore the same word if it's repeated multiple times, and it also allows you to make a word optional (by giving it a minimum repetition of 0).

Grammars that use all these features can become quite complex. For more information about the SRGS standard and its grammar rules, refer to `www.w3.org/TR/speech-grammar`.

The Last Word

In this example, you explored how to integrate sound and video into a WPF application. You learned about two ways to control the playback of media files—either programmatically by using the methods of the MediaPlayer or MediaTimeline classes or declaratively by using a storyboard.

As always, the best approach depends on your requirements. The code-based approach gives you more control and flexibility, but it also forces you to manage more details and introduces additional complexity. As a general rule, the code-based approach is best if you need fine-grained control over audio playback. However, if you need to combine media playback with animations, the declarative approach is far easier.