```
    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 (like 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 a 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 a 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.

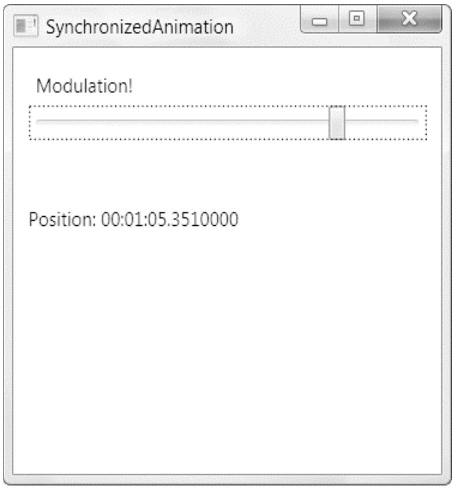
```
<Window.Resources>
  <Storyboard x:Key="Board" SlipBehavior="Slip">
    <MediaTimeline Source="sq3gm1.mid"
     Storyboard.TargetName="media"/>

      <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 using the slider, the three animations are adjusted automatically to the appropriate point by the MediaTimeline. (The slider is kept synchronized 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.



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 other content to show through behind your video window. In fact, you can even stack multiple semitransparent video windows on top of each other (with dire consequences for performance).
- You can use an 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 using a VisualBrush, which allows you to create specific effects like reflection.
- You can place a video window on a three-dimensional surface and use an animation to move it 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 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>
```

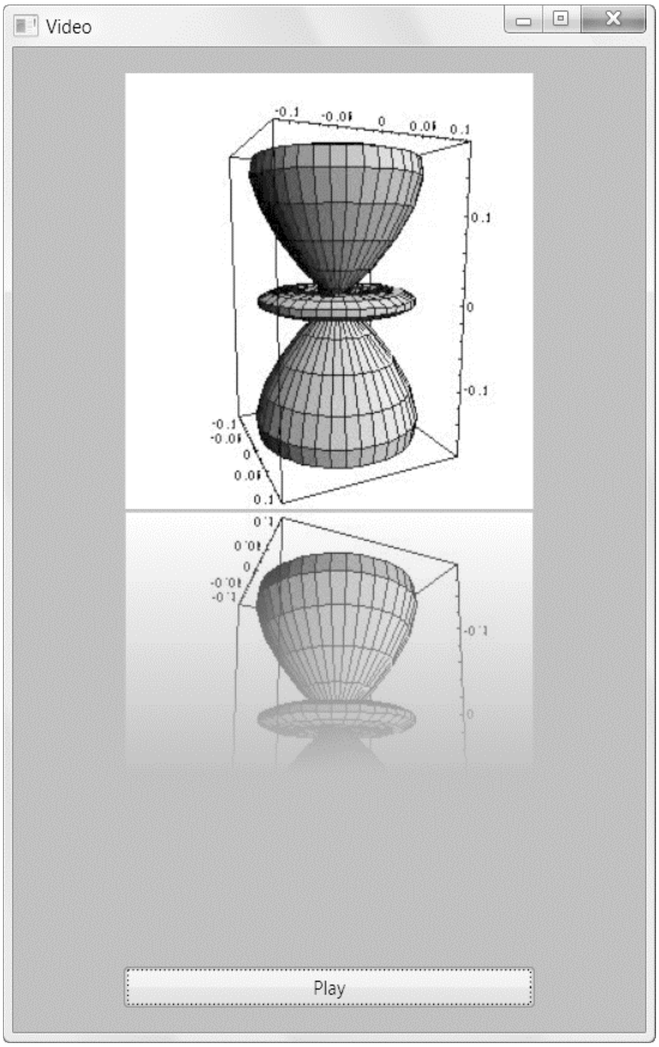


Figure 26-4. *Reflected video*

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 another example that demonstrates 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 it 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 to 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.

Although all modern versions of Windows have speech synthesis built in, the computerized voice they use is different. Windows XP uses the robotic-sounding Sam voice, while Windows Vista and Windows 7 include a more natural female voice named Anna. You can download and install additional voices on either operating system.

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 a AppendTextWithHint() method that allows you to deal with telephone numbers, dates, times, and words that need to spelled out. You supply your choice using the SayAs enumeration. Here's an example:

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

This produces the narration "The word laser is spelt 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 Synthesis Markup Language (SSML) standard, which is described at `http://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 feature of the Windows operating system. Speech recognition is built into Windows Vista and Windows 7, but not Windows XP. Instead, it's available to Windows XP users through Office XP or later, the Windows XP Plus! Pack, or the free Microsoft Speech Software Development Kit.

■ **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 actually 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 SpeechRecognitionRejected event if it cannot. The SpeechRecognitionRejected event includes information about what the SpeechRecognizer believes the spoken input might have been, but its confident 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 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 using the GrammarBuilder. The GrammarBuilder plays an analogous role 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 like *red on* and *green off*. Alternate input like *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 `http://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 different ways to control the playback of media files—either programmatically using the methods of the MediaPlayer or MediaTimeline classes or declaratively 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.

CHAPTER 27

3-D Drawing

Developers have used DirectX and OpenGL to build three-dimensional interfaces for many years. However, the difficult programming model and the substantial video card requirements have kept 3-D programming out of most mainstream consumer applications and business software.

WPF introduces a new expansive 3-D model that promises to change all that. Using WPF, you can build complex 3-D scenes out of straightforward markup. Helper classes provide hit-testing, mouse-based rotation, and other fundamental building blocks. And virtually any computer can display the 3-D content, thanks to WPF's ability to fall back on software rendering when video card support is lacking.

The most remarkable part of WPF's libraries for 3-D programming is that they are designed to be a clear, consistent extension of the WPF model you've already learned about. For example, you use the same set of brush classes to paint 3-D surfaces as you use to paint 2-D shapes. You use a similar transform model to rotate, skew, and move 3-D objects, and a similar geometry model to define their contours. More dramatically, you can use the same styling, data binding, and animation features on 3-D objects as you use with 2-D content. It's this support of high-level WPF features that makes WPF's 3-D graphics suitable for everything from eye-catching effects in simple games to charting and data visualization in a business application. (The one situation where WPF's 3-D model *isn't* sufficient is high-powered real-time games. If you're planning to build the next Halo, you're much better off with the raw power of DirectX.)

Even though WPF's model for 3-D drawing is surprisingly clear and consistent, creating rich 3-D interfaces is still difficult. In order to code 3-D animations by hand (or just understand the underlying concepts), you need to master more than a little math. And modeling anything but a trivial 3-D scene with handwritten XAML is a huge, error-prone chore—it's far more involved than the 2-D equivalent of creating a XAML vector image by hand. For that reason, you're much more likely to rely on a third-party tool to create 3-D objects, export them to XAML, and then add them to your WPF applications.

Entire books have been written about all these issues—3-D programming math, 3-D design tools, and the 3-D libraries in WPF. In this chapter, you'll learn enough to understand the WPF model for 3-D drawing, create basic 3-D shapes, design more advanced 3-D scenes with a 3-D modeling tool, and use some of the invaluable code released by the WPF team and other third-party developers.

3-D Drawing Basics

A 3-D drawing in WPF involves four ingredients:

- A viewport, which hosts your 3-D content
- A 3-D object
- A light source that illuminates part or all of your 3-D scene
- A camera, which provides the vantage point from which you view the 3-D scene

Of course, more complex 3-D scenes will feature multiple objects and may include multiple light sources. (It's also possible to create a 3-D object that doesn't require a light source, if the 3-D object itself gives off light.) However, these basic ingredients provide a good starting point.

Compared to 2-D graphics, it's the second and third points that really make a difference. Programmers who are new to 3-D programming sometimes assume that 3-D libraries are just a simpler way to create an object that has a 3-D appearance, such as a glowing cube or a spinning sphere. But if that's all you need, you're probably better off creating a 3-D drawing using the 2-D drawing classes you've already learned about. After all, there's no reason that you can't use the shapes, transforms, and geometries you learned about in Chapter 12 and Chapter 13 to construct a shape that appears to be 3-D—in fact, it's usually easier than working with the 3-D libraries.

So what's the advantage of using the 3-D support in WPF? The first advantage is that you can create effects that would be extremely complex to calculate using a simulated 3-D model. One good example is light effects such as reflection, which become very involved when working with multiple light sources and different materials with different reflective properties. The other advantage to using a 3-D drawing model is that it allows you to interact with your drawing as a set of 3-D objects. This greatly extends what you can do programmatically. For example, once you build the 3-D scene you want, it becomes almost trivially easy to rotate your object or rotate the camera around your object. Doing the same work with 2-D programming would require an avalanche of code (and math).

Now that you know what you need, it's time to build an example that has all these pieces. This is the task you'll tackle in the following sections.

The Viewport

If you want to work with 3-D content, you need a container that can host it. This container is the Viewport3D class, which is found in the System.Windows.Controls namespace. Viewport3D derives from FrameworkElement, and so it can be placed anywhere you'd place a normal element. For example, you can use it as the content of a window or a page, or you can place it inside a more complex layout.

The Viewport3D class only hints at the complexity of 3-D programming. It adds just two properties—Camera, which defines your lookout onto the 3-D scene, and Children, which holds all the 3-D objects you want to place in the scene. Interestingly enough, the light source that illuminates your 3-D scene is itself an object in the viewport.

■ **Note** Among the inherited properties in the Viewport3D class, one is particularly significant: ClipToBounds. If set to true (the default), content that stretches beyond the bounds of the viewport is trimmed out. If set to false, this content appears overtop of any adjacent elements. This is the same behavior you get from the ClipToBounds property of the Canvas. However, there's an important difference when using the Viewport3D: performance. Setting Videport3D.ClipToBounds to false can dramatically improve performance when rendering a complex, frequently refreshed 3-D scene.

3-D Objects

The viewport can host any 3-D object that derives from Visual3D (from the System.Windows.Media.Media3D namespace, where the vast majority of the 3-D classes live). However, you'll need to perform a bit more work than you might expect to create a 3-D visual. In version 1.0, the WPF library lacks a collection of 3-D shape primitives. If you want a cube, a cylinder, a torus, and so on, you'll need to build it yourself.

One of the nicest design decisions that the WPF team made when building the 3-D drawing classes was to structure them in a similar way as the 2-D drawing classes. That means you'll immediately be able to understand the purpose of a number of core 3-D classes (even if you don't yet know how to use them). Table 27-1 spells out the relationships.

Table 27-1. *2-D Classes and 3-D Classes Compared*

2-D Class	3-D Class	Notes
Visual	Visual3D	Visual3D is the base class for all 3-D objects (objects that are rendered in a Viewport3D container). Like the Visual class, you could use the Visual3D class to derive lightweight 3-D shapes or to create more complex 3-D controls that provide a richer set of events and framework services. However, you won't get much help. You're more likely to use one of the classes that derive from Visual3D, such as ModelVisual3D or ModelUIElement3D.
Geometry	Geometry3D	The Geometry class is an abstract way to define a 2-D figure. Often geometries are used to define complex figures that are composed out of arcs, lines, and polygons. The Geometry3D class is the 3-D analogue—it represents a 3-D surface. However, while there are several 2-D geometries, WPF includes just a single concrete class that derives from Geometry3D: MeshGeometry3D. The MeshGeometry3D has a central importance in 3-D drawing because you'll use it to define all your 3-D objects.
GeometryDrawing	GeometryModel3D	There are several ways to use a 2-D Geometry object. You can wrap it in a GeometryDrawing and use that to paint the surface of an element or the content of a Visual. The GeometryModel3D class serves the same purpose—it takes a Geometry3D, which can then be used to fill your Visual3D.
Transform	Transform3D	You already know that 2-D transforms are incredibly useful tools for manipulating elements and shapes in all kinds of ways, including moving, skewing, and rotating them. Transforms are also indispensable when performing animations. Classes that derive from Transform3D perform the same magic with 3-D objects. In fact, you'll find surprisingly similar transform classes such as RotateTransform3D, ScaleTransform3D, TranslateTransform3D, Transform3DGroup, and MatrixTransform3D. Of course, the options provided by an extra dimension are considerable, and 3-D transforms are able to warp and distort visuals in ways that look quite different.

At first, you may find it a bit difficult to untangle the relationships between these classes. Essentially, the Viewport3D holds Visual3D objects. To actually give a Visual3D some content, you'll need to define a Geometry3D that describes the shape and wrap it in a GeometryModel3D. You can then use that as the content for your Visual3D. Figure 27-1 shows this relationship.

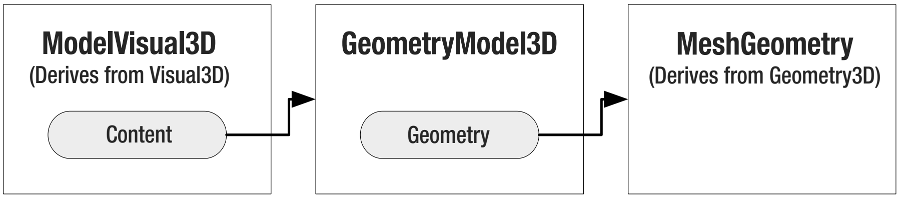


Figure 27-1. *How a 3-D object is defined*

This two-step process—defining the shapes you want to use in abstract and then fusing them with a visual—is an optional approach for 2-D drawing. However, it's mandatory for 3-D drawing because there are no prebuilt 3-D classes in the library. (The members of the WPF team and others have released some sample code online that starts to fill this gap, but it's still evolving.)

The two-step process is also important because 3-D models are a bit more complex than 2-D models. For example, when you create a Geometry3D object, you not only specify the vertexes of your shape, you also specify the *material* out of which it's composed. Different materials have different properties for reflecting and absorbing light.

Geometry

To build a 3-D object, you need to start by building the geometry. As you've already learned, there's just one class that fills this purpose: MeshGeometry3D.

Unsurprisingly, a MeshGeometry3D object represents a *mesh*. If you've ever dealt with 3-D drawing before (or if you've read a bit about the technology that underlies modern video cards), you may already know that computers prefer to build 3-D drawings out of triangles. That's because a triangle is the simplest, most granular way to define a surface. Triangles are simple because every triangle is defined by just three points (the vertexes at the corner). Arcs and curved surfaces are obviously more complex. Triangles are granular because other straight-edged shapes (squares, rectangles, and more complex polygons) can be broken down into a collection of triangles. For better or worse, modern day graphics hardware and graphics programming is built on this core abstraction.

Obviously, most of the 3-D objects you want won't look like simple, flat triangles. Instead you'll need to combine triangles—sometimes just a few, but often hundreds or thousands that line up with one another at varying angles. A mesh is this combination of triangles. With enough triangles, you can ultimately create the illusion of anything, including a complex surface. (Of course, there are performance considerations involved, and 3-D scenes often map some sort of bitmap or 2-D content onto a triangle in a mesh to create the illusion of a complex surface with less overhead. WPF supports this technique.)

Understanding how a mesh is defined is one of the first keys to 3-D programming. If you look at the MeshGeometry3D class, you'll find that it adds the four properties listed in Table 27-2.

Table 27-2. *Properties of the MeshGeometry3D Class*

Name	Description
Positions	Contains a collection of all the points that define the mesh. Each point is a vertex in a triangle. For example, if your mesh has 10 completely separate triangles, you'll have 30 points in this collection. More commonly, some of your triangles will join at their edges, which means one point will become the vertex of several triangles. For example, a cube requires 12 triangles (two for each side), but only 8 distinct points. Making matters even more complicated, you may choose to define the same shared vertex multiple times, so that you can better control how separate triangles are shaded with the Normals property.
TriangleIndices	Defines the triangles. Each entry in this collection represents a single triangle by referring to three points from the Positions collection.
Normals	Provides a vector for each vertex (each point in the Positions collection). This vector indicates how the point is angled for lighting calculations. When WPF shades the face of a triangle, it measures the light at each of the three vertexes using the normal vector. Then, it interpolates between these three points to fill the surface of the triangle. Getting the right normal vectors makes a substantial difference to how a 3-D object is shaded—for example, it can make the divisions between triangles blend together or appear as sharp lines.
TextureCoordinates	Defines how a 2-D texture is mapped onto your 3-D object when you use a VisualBrush to paint it. The TextureCoordinates collection provides a 2-D point for each 3-D point in the Positions collection.

You'll consider shading with normals and texture mapping later in this chapter. But first, you'll learn how to build a basic mesh.

The following example shows the simplest possible mesh, which consists of a single triangle. The units you use aren't important because you can move the camera closer or farther away, and you can change the size or placement of individual 3-D objects using transforms. What *is* important is the coordinate system, which is shown in Figure 27-2. As you can see, the X and Y axes have the same orientation as in 2-D drawing. What's new is the Z axis. As the Z axis value decreases, the point moves farther away. As it increases, the point moves closer.

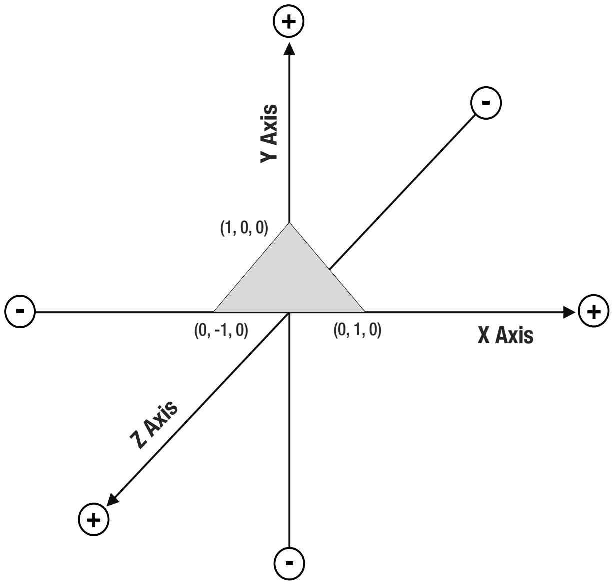


Figure 27-2. *A triangle in 3-D space*

Here's the MeshGeometry element that you can use to define this shape inside a 3-D visual. The MeshGeometry3D object in this example doesn't use the Normals property or the TextureCoordinates property because the shape is so simple and will be painted with a SolidColorBrush:

```
<MeshGeometry3D Positions="-1,0,0  0,1,0  1,0,0" TriangleIndices="0,2,1" />
```

Here, there are obviously just three points, which are listed one after the other in the Positions property. The order you use in the Positions property isn't important because the TriangleIndices property clearly defines the triangle. Essentially, the TriangleIndices property states that there is a single triangle made of point #0, #2, and #1. In other words, the TriangleIndices property tells WPF to draw the triangle by drawing a line from (-1, 0 ,0) to (1, 0, 0) and then to (0, 1, 0).

3-D programming has several subtle, easily violated rules. When defining a shape, you'll face the first one—namely, you must list the points in a counterclockwise order around the Z axis. This example follows that rule. However, you could easily violate it if you changed the TriangleIndices to 0, 1, 2. In this case, you'd still define the same triangle, but that triangle would be backward—in other words, if you look at it down the Z axis (as in Figure 27-2), you'll actually be looking at the *back* of the triangle.

■ **Note** The difference between the back of a 3-D shape and the front is not a trivial one. In some cases, you may paint both with a different brush. Or you may choose not to paint the back at all in order to avoid using any resources for a part of the scene that you'll never see. If you inadvertently define the points in a clockwise order, and you haven't defined the material for the back of your shape, it will disappear from your 3-D scene.

Geometry Model and Surfaces

Once you have the properly configured MeshGeometry3D that you want, you need to wrap it in a GeometryModel3D.

The GeometryModel3D class has just three properties: Geometry, Material, and Back-Material. The Geometry property takes the MeshGeometry3D that defines the shape of your 3-D object. In addition, you can use the Material and BackMaterial properties to define the surface out of which your shape is composed.

The surface is important for two reasons. First, it defines the color of the object (although you can use more complex brushes that paint textures rather than solid colors). Second, it defines how that material responds to light.

WPF includes four material classes, all of which derive from the abstract Material class in the System.Windows.Media.Media3D namespace). They're listed in Table 27-3. In this example, we'll stick with DiffuseMaterial, which is the most common choice because its behavior is closest to a real-world surface.

Table 27-3. *Material Classes*

Name	Description
DiffuseMaterial	Creates a flat, matte surface. It diffuses light evenly in all directions.
SpecularMaterial	Creates a glossy, highlighted look (think metal or glass). It reflects light back directly, like a mirror.
EmissiveMaterial	Creates a glowing look. It generates its own light (although this light does not reflect off other objects in the scene).
MaterialGroup	Lets you combine more than one material. The materials are then layered overtop of one another in the order they're added to the MaterialGroup.

DiffuseMaterial offers a single Brush property that takes the Brush object you want to use to paint the surface of your 3-D object. (If you use anything other than a SolidColorBrush, you'll need to set the MeshGeometry3D.TextureCoordinates property to define the way it's mapped onto the object, as you'll see later in this chapter.)

Here's how you can configure the triangle to be painted with a yellow matte surface:

```
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="-1,0,0  0,1,0  1,0,0" TriangleIndices="0,2,1" />
  </GeometryModel3D.Geometry>

  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Yellow" />
  </GeometryModel3D.Material>
</GeometryModel3D>
```

In this example, the BackMaterial property is not set, so the triangle will disappear if viewed from behind.

All that remains is to use this GeometryModel3D to set the Content property of a ModelVisual3D and then place that ModelVisual3D in a viewport. But in order to see your object, you'll also need two more details: a light source and a camera.

Light Sources

In order to create realistically shaded 3-D objects, WPF uses a lighting model. The basic idea is that you add one (or several) light sources to your 3-D scene. Your objects are then illuminated based on the type of light you've chosen, its position, direction, and intensity.

Before you delve into WPF lighting, it's important that you realize that the WPF lighting model doesn't behave like light in the real world. Although the WPF lighting system is constructed to emulate the real world, calculating true light reflections is a processor-intensive task. WPF makes use of a number of simplifications that ensure the lighting model is practical, even in animated 3-D scenes with multiple light sources. These simplifications include the following:

- Light effects are calculated for objects *individually*. Light reflected from one object will not reflect off another object. Similarly, an object will not cast a shadow on another object, no matter where it's placed.
- Lighting is calculated at the vertexes of each triangle and then interpolated over the surface of the triangle. (In other words, WPF determines the light strength at each corner and blends that to fill in the triangle.) As a result of this design, objects that have relatively few triangles may not be illuminated correctly. To achieve better lighting, you'll need to divide your shapes into hundreds or thousands of triangles.

Depending on the effect you're trying to achieve, you may need to work around these issues by combining multiple light sources, using different materials, and even adding extra shapes. In fact, getting the precise result you want is part of the art of 3-D scene design.

■ **Note** Even if you don't provide a light source, your object will still be visible. However, without a light source, all you'll see is a solid black silhouette.

WPF provides four light classes, all of which derive from the abstract Light class. Table 27-4 lists them all. In this example, we'll stick with a single DirectionalLight, which is the most common type of lighting.

Table 27-4. *Light Classes*

Name	Description
DirectionalLight	Fills the scene with parallel rays of light traveling in the direction you specify.
AmbientLight	Fills the scene with scattered light.
PointLight	Radiates light in all directions, beginning at a single point in space.
SpotLight	Radiates light outward in a cone, starting from a single point.

Here's how you can define a white DirectionalLight:

```
<DirectionalLight Color="White" Direction="-1,-1,-1" />
```

In this example, the vector that determines the path of the light starts at the origin (0, 0, 0) and goes to (-1, -1, -1). That means that each ray of light is a straight line that travels from top-right front toward the bottom-left back. This makes sense in this example because the triangle (shown in Figure 27-2) is angled to face this light.

When calculating the light direction, it's the angle that's important, not the length of your vector. That means a light direction of (-2, -2, -2) is equivalent to the normalized vector (-1, -1, -1) because the angle it describes is the same.

In this example, the direction of the light doesn't line up exactly with the triangle's surface. If that's the effect you want, you'll need a light source that sends its beams straight down the Z axis, using a direction of (0, 0, -1). This distinction is deliberate. Because the beams strike the triangle at an angle, the triangle's surface will be shaded, which creates a more pleasing effect.

Figure 27-3 shows an approximation of the (-1, -1, -1) directional light as it strikes the triangle. Remember, a directional light fills the entire 3-D space.

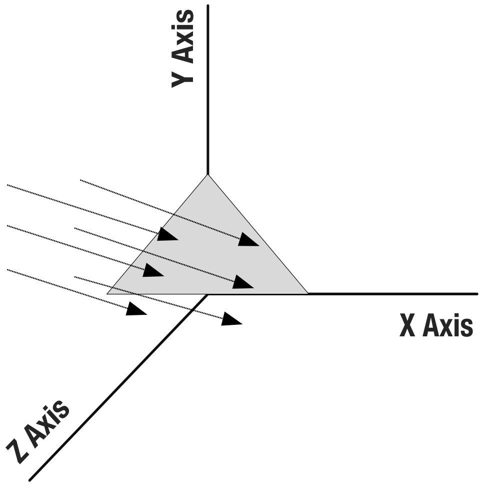


Figure 27-3. *The path of a (-1, -1, -1) directional light*

■ **Note** Directional lights are sometimes compared to sunlight. That's because the light rays received from a faraway light source (such as the sun) become almost parallel.

All light objects derive indirectly from GeometryModel3D. That means that you treat them exactly like 3-D objects by placing them inside a ModelVisual3D and adding them to a viewport. Here's a viewport that includes both the triangle you saw earlier and the light source:

```
<Viewport3D>
  <Viewport3D.Camera>...</Viewport3D.Camera>
```