Objectives

In this chapter you’ll:
■ Manipulate fonts.
■ Draw basic WPF shapes.
■ Use WPF brushes to customize the Fill or Background of an object.
■ Use WPF transforms to reposition or reorient GUI elements.
■ Customize the look of a control while maintaining its functionality.
■ Animate the properties of a GUI element.
■ Use speech synthesis and recognition.
Chapter 24  WPF Graphics and Multimedia

Outline

24.1 Introduction
24.2 Controlling Fonts
24.3 Basic Shapes
24.4 Polygons and Polylines
24.5 Brushes
24.6 Transforms
24.7 WPF Customization: A Television GUI
24.8 Animations
24.9 Speech Synthesis and Speech Recognition
24.10 Wrap-Up

24.1 Introduction

This chapter overviews WPF’s graphics and multimedia capabilities, including two-dimensional and three-dimensional shapes, fonts, transformations, animations, audio and video. The graphics system in WPF is designed to use your computer’s graphics hardware to reduce the load on the CPU.

WPF graphics use resolution-independent units of measurement, making apps more uniform and portable across devices. The size properties of graphic elements in WPF are measured in machine-independent pixels, where one pixel typically represents 1/96 of an inch—however, this depends on the computer’s DPI (dots per inch) setting. The graphics engine determines the correct pixel count so that all users see elements of the same size on all devices.

Graphic elements are rendered on screen using a vector-based system in which calculations determine how to size and scale each element, allowing graphic elements to be preserved across any rendering size. This produces smoother graphics than the so-called raster-based systems, in which the precise pixels are specified for each graphical element. Raster-based graphics tend to degrade in appearance as they’re scaled larger. Vector-based graphics appear smooth at any scale. Graphic elements other than images and video are drawn using WPF’s vector-based system, so they look good at any screen resolution.

The basic 2-D shapes are Lines, Rectangles and Ellipses. WPF also has controls that can be used to create custom shapes or curves. Brushes can be used to fill an element with solid colors, complex patterns, gradients, images or videos, allowing for unique and interesting visual experiences. WPF’s robust animation and transform capabilities allow you to further customize GUIs. Transforms reposition and reorient graphic elements. The chapter ends with an introduction to speech synthesis and recognition.

24.2 Controlling Fonts

This section introduces how to control fonts by modifying the font properties of a TextBlock control in the XAML code. Figure 24.1 shows how to use TextBlocks and how to change the properties to control the appearance of the displayed text. When building this example, we removed the StackPanel’s HorizontalAlignment, Height, VerticalAlignment and Width attributes from the XAML so that the StackPanel would occupy the entire window. Some of the font formatting in this example was performed by editing the XAML markup because some TextDecorations are not available via the Properties window.
24.2 Controlling Fonts

The text that you want to display in the TextBlock is specified either via the TextBlock’s Text property (line 10) or by placing the text between a TextBlock’s start and end tags. The FontFamily property defines the font of the displayed text. This property can be
set to any available font. Lines 11, 16 and 21 define the first three TextBlock fonts to be Arial, Times New Roman and Courier New, respectively. If the font is not specified or is not available, the default font (Segoe UI) is used.

The **FontSize** property defines the text size measured in machine-independent pixels unless the value is qualified by appending in (inches), cm (centimeters) or pt (points). When no FontSize is specified, the property is set to the default value of 12 (this is actually determined by System.MessageFontSize). The font sizes are defined in lines 11 and 21.

TextBlocks have various font-related properties. Lines 11 and 22 set the **FontWeight** property to **Bold** to make the font thicker. This property can be set either to a numeric value (1–999) or to a predefined descriptive value—such as **Light** or **UltraBold** (msdn.microsoft.com/en-us/library/system.windows.fontweights.aspx)—to define the thickness of the text. You can use the **FontStyle** property to make the text either **Italic** (line 22) or **Oblique**—which is simply a more emphasized italic.

You also can define **TextDecorations** for a TextBlock to draw a horizontal line through the text. **Overline** and **Baseline**—shown in the fourth TextBlock of Fig. 24.1—create lines above the text and at the base of the text, respectively (lines 28–29). **Strike-through** and **Underline**—shown in the fifth TextBlock—create lines through the middle of the text and under the text, respectively (lines 37–38). The **Underline** option leaves a small amount of space between the text and the line, unlike the **Baseline**. The **Location** property of the **TextDecoration** class defines which decoration you want to apply.

### 24.3 Basic Shapes

WPF has several built-in shapes. The BasicShapes example (Fig. 24.2) shows you how to display **Lines**, **Rectangles** and **Ellipses** on a WPF **Canvas** object. When building this example, we removed the Canvas’s **HorizontalAlignment**, Height, **VerticalAlignment** and Width attributes from the XAML so that the StackPanel would occupy the entire window. By default, the shape elements are not displayed in the WPF **Toolbox**, so all the shape elements in this example were added via the XAML editor—this will be the case for many other examples in this chapter.

---

**Fig. 24.2** Drawing basic shapes in XAML. (Part 1 of 2.)
24.3 Basic Shapes

The first shape drawn uses the **Rectangle** object to create a filled rectangle in the window. The layout control is a **Canvas**, which allows us to use coordinates to position the shapes. To specify the upper-left corner of the **Rectangle** at lines 9–10, we set the **Canvas.Left** and **Canvas.Top** properties to 90 and 30, respectively. We then set the **Width** and **Height** properties to 150 and 90, respectively, to specify the size. To define the **Rectangle**’s color, we use the **Fill** property (line 10). You can assign any **Color** or **Brush** to this property.

**Rectangles** also have a **Stroke** property, which defines the color of the **outline** of the shape (line 20). If either the **Fill** or the **Stroke** is not specified, that property will be rendered **transparently**. For this reason, the light blue **Rectangle** in the window has no outline, while the second **Rectangle** drawn has only an outline (with a transparent center). Shape objects have a **StrokeThickness** property which defines the **thickness** of the outline. The default value for **StrokeThickness** is 1 pixel.

A **Line** is defined by its two **endpoints**—X1, Y1 and X2, Y2. **Lines** have a **Stroke** property that defines the **color** of the line. In this example, the lines are all set to have **Black** **Strokes** (lines 13–16 and 25–26).

To draw a circle or ellipse, you can use the **Ellipse** control. The placement and size of an **Ellipse** is defined like a **Rectangle**—with the **Canvas.Left** and **Canvas.Top** properties for the **upper-left corner**, and the **Width** and **Height** properties for the size (line 23). Together, the **Canvas.Left**, **Canvas.Top**, **Width** and **Height** of an **Ellipse** define a **bounding rectangle** in which the **Ellipse** touches the center of each side of the rectangle. To draw a circle, provide the same value for the **Width** and **Height** properties. As with...
Rectangles, having an unspecified Fill property for an Ellipse makes the shape’s fill transparent (lines 29–30).

**24.4 Polygons and Polylines**

There are two shape controls for drawing *multisided shapes*—Polyline and Polygon. Polyline draws a series of connected lines defined by a set of points, while Polygon does the same but connects the start and end points to make a closed figure. The app DrawPolygons (Fig. 24.3) allows you to click anywhere on the Canvas to define points for one of three shapes. You select which shape you want to display by selecting one of the RadioButton in the second column. The difference between the Filled Polygon and the Polygon options is that the former has a Fill property specified while the latter does not.

```xml
<!-- Fig. 24.3: MainWindow.xaml -->
<!-- Defining Polylines and Polygons in XAML. -->
<Window x:Class="DrawPolygons.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="DrawPolygons" Height="400" Width="450" Name="mainWindow">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Canvas Name="drawCanvas" Grid.Column="0" Background="White"
      MouseDown="drawCanvas_MouseDown">
      <Polyline Name="polyLine" Stroke="Black" Visibility="Collapsed" />
      <Polygon Name="polygon" Stroke="Black" Visibility="Collapsed" />
      <Polygon Name="filledPolygon" Fill="DarkBlue" Visibility="Collapsed" />
    </Canvas>
  </Grid.ColumnDefinitions>
</Window>

<!-- StackPanel containing the RadioButton options -->
<StackPanel Grid.Column="1" Orientation="Vertical"
      Background="WhiteSmoke">
  <GroupBox Header="Select Type" Margin="10">
    <StackPanel>
      <!-- Polyline option -->
      <RadioButton Name="lineRadio" Content="Polyline"
        Margin="5" Checked="lineRadio_Checked" />
      <!-- unfilled Polygon option -->
      <RadioButton Name="polygonRadio" Content="Polygon"
        Margin="5" Checked="polygonRadio_Checked" />
    </StackPanel>
  </GroupBox>
</StackPanel>
```

**Fig. 24.3** | Defining Polylines and Polygons in XAML. (Part 1 of 2.)
24.4 Polygons and Polylines

The XAML defines a two-column GUI (lines 9–10). The first column contains a Canvas (lines 15–22) that the user interacts with to create the points of the selected shape. Embedded in the Canvas are a Polyline (lines 17–18) and two Polygons—one with a Fill (lines 20–21) and one without (line 19). The Visibility of a control can be set to Visible, Collapsed or Hidden. This property is initially set to Collapsed for all three shapes (lines 18, 19 and 21), because we'll display only the shape that corresponds to the selected RadioButton. The difference between Hidden and Collapsed is that a Hidden object occupies space in the GUI but is not visible, while a Collapsed object has a Width and Height of 0. As you can see, Polyline and Polygon objects have Fill and Stroke properties like the simple shapes we discussed earlier.

The RadioButtons (lines 30–40) allow you to select which shape appears in the Canvas. There is also a Button (lines 45–46) that clears the shape’s points to allow you to start over. The code-behind file for this app is shown in Fig. 24.4.

To allow the user to specify a variable number of points, line 12 in Fig. 24.4 declares a PointCollection, which is a collection that stores Point objects. This keeps track of each mouse-click location. The collection’s Add method adds new points to the end of the collection.
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace DrawPolygons
{
    public partial class MainWindow : Window
    {
        // stores the collection of points for the multisided shapes
        private PointCollection points = new PointCollection();

        // initialize the points of the shapes
        public MainWindow()
        {
            InitializeComponent();
            polyLine.Points = points; // assign Polyline points
            polygon.Points = points; // assign Polygon points
            filledPolygon.Points = points; // assign filled Polygon points
        }

        // adds a new point when the user clicks on the canvas
        private void drawCanvas_MouseDown( object sender, MouseButtonEventArgs e )
        {
            // add point to collection
            points.Add( e.GetPosition( drawCanvas ) );
        }

        // when the clear Button is clicked
        private void clearButton_Click( object sender, RoutedEventArgs e )
        {
            points.Clear(); // clear the points from the collection
        }

        // when the user selects the Polyline
        private void lineRadio_Checked( object sender, RoutedEventArgs e )
        {
            polyLine.Visibility = Visibility.Visible; // Polygon is visible, the other two are not
            polygon.Visibility = Visibility.Collapsed;
            filledPolygon.Visibility = Visibility.Collapsed;
        }

        // when the user selects the Polygon
        private void polygonRadio_Checked( object sender, RoutedEventArgs e )
        {
            polyLine.Visibility = Visibility.Collapsed;
            polygon.Visibility = Visibility.Visible;
        }
    }
}
24.5 Brushes

When the app executes, we set the Points property (lines 19–21) of each shape to reference the PointCollection instance variable created in line 12.

We created a MouseDown event handler to capture mouse clicks on the Canvas (lines 25–30). When the user clicks the mouse on the Canvas, the mouse coordinates are recorded (line 29) and the points collection is updated. Since the Points property of each of the three shapes has a reference to our PointCollection object, the shapes are automatically updated with the new Point. The Polyline and Polygon shapes connect the Points based on the ordering in the collection.

Each RadioButton's Checked event handler sets the corresponding shape's Visibility property to Visible and sets the other two to Collapsed to display the correct shape in the Canvas. For example, the lineRadio_Checked event handler (lines 39–45) makes polyline.Visible (line 42) and makes polygon and filledPolygon.Collapsed (lines 43–44). The other two RadioButton event handlers are defined similarly in lines 48–55 and lines 58–65.

The clearButton_Click event handler erases the stored collection of Points (line 35). The Clear method of the PointCollection points erases its elements.

24.5 Brushes

Brushes change an element's graphic properties, such as the Fill, Stroke or Background. A SolidColorBrush fills the element with the specified color. To customize elements further, you can use ImageBrushes, VisualBrushes and gradient brushes. Run the UsingBrushes app (Fig. 24.5) to see Brushes applied to TextBlocks and Ellipses.
Chapter 24  WPF Graphics and Multimedia

Fig. 24.5  Applying brushes to various XAML elements.  (Part 2 of 4.)
Fig. 24.5  Applying brushes to various XAML elements. (Part 3 of 4.)
ImageBrush

An ImageBrush paints an image into the property it’s assigned to (such as a Background). For instance, the TextBlock with the text “Image” and the Ellipse next to it are both filled with the same flower picture. To fill the text, we can assign the ImageBrush to the Foreground property of the TextBlock. The Foreground property specifies the fill for the text itself while the Background property specifies the fill for the area surrounding the text. Lines 33–37 apply the ImageBrush with its ImageSource set to the file we want to display (the image file must be included in the project). We also can assign the brush to the Fill of the Ellipse (lines 43–44) to display the image inside the shape. The ImageBrush’s Stretch property specifies how to stretch the image. The UniformToFill value indicates that the image should fill the element in which it’s displayed and that the original image’s aspect ratio (that is, the proportion between its width and height) should be maintained. Keeping this ratio at its original value ensures that the video does not look “stretched,” though it might be cropped.

VisualBrush and MediaElement

This example displays a video in a TextBlock’s Foreground and an Ellipse’s Fill. To use audio or video in a WPF app, you use the MediaElement control. Before using a video file in your app, add it to your Visual Studio project by dragging it from Windows Explorer to your project’s folder in the Visual Studio Solution Explorer. Select the newly added video in the Solution Explorer. Then, in the Properties window, change the Copy to Output Directory property to Copy if newer. This tells the project to copy your video to the project’s output directory where it can directly reference the file. You can now set the Source prop-
24.5 Brushes

Property of your MediaElement to the video. In the UsingBrushes app, we used media.mp4 (line 55 and 66), which we downloaded from www.nasa.gov/multimedia/videogallery.

We use the **VisualBrush** element to display a video in the desired controls. Lines 53–57 define the Brush with a MediaElement assigned to its *Visual* property. In this property you can completely customize the look of the brush. By assigning the video to this property, we can apply the brush to the Foreground of the TextBlock (lines 51–58) and the Fill of the Ellipse (lines 63–69) to *play the video inside the controls*. The Fill of the third Row's elements is different in each screen capture in Fig. 24.5, because the video is playing inside the two elements. The VisualBrush's Stretch property specifies how to stretch the video.

**Gradients**

A gradient is a gradual *transition* through two or more colors. Gradients can be applied as the background or fill for various elements. There are two types of gradients in WPF—LinearGradientBrush and RadialGradientBrush. The LinearGradientBrush transitions through colors along a straight path. The RadialGradientBrush transitions through colors radially outward from a specified point. Linear gradients are discussed in the UsingGradients example (Figs. 24.6–24.7), which displays a gradient across the window. This was created by applying a LinearGradientBrush to a Rectangle's Fill. The gradient starts white and transitions linearly to black from left to right. You can set the RGBA values of the start and end colors to change the look of the gradient. The values entered in the TextBoxes must be in the range 0–255 for the app to run properly. If you set either color’s alpha value to less than 255, you’ll see the text “Transparency test” in the background, showing that the Rectangle is semitransparent. The XAML code for this app is shown in Fig. 24.6.

```xml
<Window x:Class="UsingGradients.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UsingGradients" Height="200" Width="450">
    <Grid>
        <!-- TextBlock in the background to show transparency -->
        <TextBlock TextWrapping="Wrap" Text="Transparency Test"
            FontSize="30" HorizontalAlignment="Center"
            VerticalAlignment="Center"/>
        <!-- sample rectangle with linear gradient fill -->
        <Rectangle>
            <Rectangle.Fill>
                <!-- Defining gradients in XAML. (Part 1 of 3.)
                <!-- Fig. 24.6: MainWindow.xaml -->
                <!-- Defining gradients in XAML. -->
                <window x:Class="UsingGradients.MainWindow"
                    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    Title="UsingGradients" Height="200" Width="450">
                    <Grid>
                        <Grid.RowDefinitions>
                            <RowDefinition />
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                        </Grid.RowDefinitions>
                        <!-- TextBlock in the background to show transparency -->
                        <TextBlock TextWrapping="Wrap" Text="Transparency Test"
                            FontSize="30" HorizontalAlignment="Center"
                            VerticalAlignment="Center"/>
                        <!-- sample rectangle with linear gradient fill -->
                        <Rectangle>
                            <Rectangle.Fill>
                                <!-- Defining gradients in XAML. (Part 1 of 3.)
                                <!-- Fig. 24.6: MainWindow.xaml -->
                                <!-- Defining gradients in XAML. -->
                                <window x:Class="UsingGradients.MainWindow"
                                    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                                    Title="UsingGradients" Height="200" Width="450">
                                    <Grid>
                                        <Grid.RowDefinitions>
                                            <RowDefinition />
                                            <RowDefinition Height="Auto" />
                                            <RowDefinition Height="Auto" />
                                            <RowDefinition Height="Auto" />
                                        </Grid.RowDefinitions>
                                        <!-- TextBlock in the background to show transparency -->
                                        <TextBlock TextWrapping="Wrap" Text="Transparency Test"
                                            FontSize="30" HorizontalAlignment="Center"
                                            VerticalAlignment="Center"/>
                                        <!-- sample rectangle with linear gradient fill -->
                                        <Rectangle>
                                            <Rectangle.Fill>
...
<LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
    <!-- gradient stop can define a color at any offset -->
    <GradientStop x:Name="startGradient" Offset="0.0"
        Color="White" />
    <GradientStop x:Name="stopGradient" Offset="1.0"
        Color="Black" />
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>

<!-- shows which TextBox corresponds with which ARGB value-->
<StackPanel Grid.Row="1" Orientation="Horizontal">
    <TextBlock TextWrapping="Wrap" Text="Alpha:
        Width="75" Margin="5">
    <TextBlock TextWrapping="Wrap" Text="Red:
        Width="75" Margin="5">
    <TextBlock TextWrapping="Wrap" Text="Green:
        Width="75" Margin="5">
    <TextBlock TextWrapping="Wrap" Text="Blue:
        Width="75" Margin="5">
</StackPanel>

<!-- GUI to select the color of the first GradientStop -->
<StackPanel Grid.Row="2" Orientation="Horizontal">
    <TextBox Name="fromAlpha" TextWrapping="Wrap" Text="255"
        Width="75" Margin="5"/>
    <TextBox Name="fromRed" TextWrapping="Wrap" Text="255"
        Width="75" Margin="5"/>
    <TextBox Name="fromGreen" TextWrapping="Wrap" Text="255"
        Width="75" Margin="5"/>
    <TextBox Name="fromBlue" TextWrapping="Wrap" Text="255"
        Width="75" Margin="5"/>
    <Button Name="fromButton" Content="Start Color"
        Width="75" Margin="5" Click="fromButton_Click"/>
</StackPanel>

<!-- GUI to select the color of second GradientStop -->
<StackPanel Grid.Row="3" Orientation="Horizontal">
    <TextBox Name="toAlpha" TextWrapping="Wrap" Text="255"
        Width="75" Margin="5"/>
    <TextBox Name="toRed" TextWrapping="Wrap" Text="0"
        Width="75" Margin="5"/>
    <TextBox Name="toGreen" TextWrapping="Wrap" Text="0"
        Width="75" Margin="5"/>
    <TextBox Name="toBlue" TextWrapping="Wrap" Text="0"
        Width="75" Margin="5"/>
    <Button Name="toButton" Content="End Color"
        Width="75" Margin="5" Click="toButton_Click"/>
</StackPanel>
</StackPanel>
</Grid>
</Window>

---

**Fig. 24.6** | Defining gradients in XAML. (Part 2 of 3.)
24.5 Brushes

The GUI for this app contains a single Rectangle with a LinearGradientBrush applied to its Fill (lines 21–31). We define the StartPoint and EndPoint of the gradient in line 23. You must assign logical points to these properties, meaning the \( x \)- and \( y \)-coordinates take values between 0 and 1, inclusive. Logical points are used to reference locations in the control independent of the actual size. The point (0,0) represents the top-left corner while the point (1,1) represents the bottom-right corner. The gradient will transition linearly from the start to the end—for RadialGradientBrush, the StartPoint represents the center of the gradient. The values in line 23 indicate that the gradient should start at the left and be displayed horizontally from left to right.

A gradient is defined using GradientStops. A GradientStop defines a single color along the gradient. You can define as many stops as you want by embedding them in the brush element. A GradientStop is defined by its Offset and Color properties. The Color property defines the color you want the gradient to transition to—lines 25–26 and 27–28 indicate that the gradient transitions through white and black. The Offset property defines where along the linear transition you want the color to appear. You can assign any double value between 0 and 1, inclusive, which represent the start and end of the gradient. In the example we use 0.0 and 1.0 offsets (lines 25 and 27), indicating that these colors appear at the start and end of the gradient (which were defined in line 23), respectively. The code in Fig. 24.7 allows the user to set the Colors of the two stops.

When fromButton is clicked, we use the Text properties of the corresponding TextBoxes to obtain the RGBA values and create a new color. We then assign it to the Color property of startGradient (Fig. 24.7, lines 21–25). When the toButton is clicked, we do the same for stopGradient’s Color (lines 32–36).

```csharp
// Fig. 24.7: MainWindow.xaml.cs
// Customizing gradients.
using System;
using System.Windows;
using System.Windows.Media;
namespace UsingGradients
{
    public partial class MainWindow : Window
    {
```

Fig. 24.6 | Defining gradients in XAML. (Part 3 of 3.)

Fig. 24.7 | Customizing gradients. (Part 1 of 2.)
A transform can be applied to any UI element to reposition or reorient the graphic. There are several types of transforms. Here we discuss TranslateTransform, RotateTransform, SkewTransform and ScaleTransform. A TranslateTransform moves an object to a new location. A RotateTransform rotates the object around a point and by a specified RotationAngle. A SkewTransform skews (or shear) the object. A ScaleTransform scales the object’s x- and y-coordinate points by different specified amounts. See Section 24.7 for an example using a SkewTransform and a ScaleTransform.

The next example draws a star using the Polygon control and uses RotateTransforms to create a circle of randomly colored stars. Figure 24.8 shows the XAML code and a sample output. Lines 10–11 define a Polygon in the shape of a star. The Polygon’s Points property is defined here in a new syntax. Each Point in the collection is defined with a comma separating the x- and y-coordinates. A single space separates each Point. We defined ten Points in the collection. The code-behind file is shown in Fig. 24.9.

In the code-behind, we replicate star 18 times and apply a different RotateTransform to each to get the circle of Polygons shown in the screen capture of Fig. 24.8. Each iteration of the loop duplicates star by creating a new Polygon with the same set of points (Fig. 24.9, lines 22–23). To generate the random colors for each star, we use the Random

```csharp
11    // constructor
12    public MainWindow()
13    {
14        InitializeComponent();
15    } // end constructor
16
17    // change the starting color of the gradient when the user clicks
18    private void fromButton_Click( object sender, RoutedEventArgs e )
19    {
20        // change the color to use the ARGB values specified by user
21        startGradient.Color = Color.FromArgb(
22            Convert.ToByte( fromAlpha.Text ),
23            Convert.ToByte( fromRed.Text ),
24            Convert.ToByte( fromGreen.Text ),
25            Convert.ToByte( fromBlue.Text ) );
26    } // end method fromButton_Click
27
28    // change the ending color of the gradient when the user clicks
29    private void toButton_Click( object sender, RoutedEventArgs e )
30    {
31        // change the color to use the ARGB values specified by user
32        stopGradient.Color = Color.FromArgb(
33            Convert.ToByte( toAlpha.Text ),
34            Convert.ToByte( toRed.Text ),
35            Convert.ToByte( toGreen.Text ),
36            Convert.ToByte( toBlue.Text ) );
37    } // end method toButton_Click
38 } // end class MainWindow
39 } // end namespace UsingGradients

Fig. 24.7 | Customizing gradients. (Part 2 of 2.)

24.6 Transforms

A transform can be applied to any UI element to reposition or reorient the graphic. There are several types of transforms. Here we discuss TranslateTransform, RotateTransform, SkewTransform and ScaleTransform. A TranslateTransform moves an object to a new location. A RotateTransform rotates the object around a point and by a specified RotationAngle. A SkewTransform skews (or shear) the object. A ScaleTransform scales the object’s x- and y-coordinate points by different specified amounts. See Section 24.7 for an example using a SkewTransform and a ScaleTransform.

The next example draws a star using the Polygon control and uses RotateTransforms to create a circle of randomly colored stars. Figure 24.8 shows the XAML code and a sample output. Lines 10–11 define a Polygon in the shape of a star. The Polygon’s Points property is defined here in a new syntax. Each Point in the collection is defined with a comma separating the x- and y-coordinates. A single space separates each Point. We defined ten Points in the collection. The code-behind file is shown in Fig. 24.9.

In the code-behind, we replicate star 18 times and apply a different RotateTransform to each to get the circle of Polygons shown in the screen capture of Fig. 24.8. Each iteration of the loop duplicates star by creating a new Polygon with the same set of points (Fig. 24.9, lines 22–23). To generate the random colors for each star, we use the Random
24.6 Transforms

<!-- Fig. 24.8: MainWindow.xaml -->
<!-- Defining a Polygon representing a star in XAML. -->
<Window x:Class="DrawStars.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DrawStars" Height="330" Width="330" Name="DrawStars">
    <Canvas Name="mainCanvas" <!-- Main canvas of the app -->
        <!-- Polygon with points that make up a star -->
        <Polygon Name="star" Fill="Green" Points="205,150 217,186 259,186 223,204 233,246 205,222 177,246 187,204 151,186 193,186" />
    </Canvas>
</Window>

FIG. 24.8   |   Defining a Polygon representing a star in XAML.

// Fig. 24.9: MainWindow.xaml.cs
// Applying transforms to a Polygon.
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Shapes;

namespace DrawStars
{
    public partial class MainWindow : Window
    {
        // constructor
        public MainWindow()
        {
            InitializeComponent();
            Random random = new Random(); // get random values for colors
            // create 18 more stars
            for ( int count = 0; count < 18; ++count )
            {
            }
        }
    }
}

FIG. 24.9   |   Applying transforms to a Polygon. (Part 1 of 2.)
24-18  Chapter 24  WPF Graphics and Multimedia

22 Polygon newStar = new Polygon(); // create a polygon object
23 newStar.Points = star.Points; // copy the points collection
24
25 byte[] colorValues = new byte[4]; // create a Byte array
26 random.NextBytes(colorValues); // create four random values
27
28 newStar.Fill = new SolidColorBrush(Color.FromArgb(
29   colorValues[0], colorValues[1], colorValues[2],
30   colorValues[3])); // creates a random color brush
31
32 // apply a rotation to the shape
33 RotateTransform rotate =
34   new RotateTransform(count * 20, 150, 150);
35 newStar.RenderTransform = rotate;
36 mainCanvas.Children.Add(newStar);
37 } // end for
38 } // end constructor
39 } // end class MainWindow
40 } // end namespace DrawStars

Fig. 24.9  |  Applying transforms to a Polygon. (Part 2 of 2.)

class’s NextBytes method, which assigns a random value in the range 0–255 to each element in its Byte array argument. Lines 25–26 define a four-element Byte array and supply the array to the NextBytes method. We then create a new Brush with a color that uses the four randomly generated values as its RGBA values (lines 27–29).

To apply a rotation to the new Polygon, we set the RenderTransform property to a new RotateTransform object (lines 32–34). Each iteration of the loop assigns a new rotation-angle value by using the control variable multiplied by 20 as the RotationAngle argument. The first argument in the RotateTransform’s constructor is the angle by which to rotate the object. The next two arguments are the x- and y-coordinates of the point of rotation. The center of the circle of stars is the point (150,150) because all 18 stars were rotated about that point. Each new shape is added as a new Child element to mainCanvas (line 35) so it can be rendered on screen.

24.7  WPF Customization: A Television GUI

In Chapter 23, we introduced several techniques for customizing the appearance of WPF controls. We revisit them in this section, now that we have a basic understanding of how to create and manipulate 2-D graphics in WPF. You’ll learn to apply combinations of shapes, brushes and transforms to define every aspect of a control’s appearance and to create graphically sophisticated GUIs.

This case study models a television. The GUI depicts a 3-D-looking environment featuring a TV that can be turned on and off. When it’s on, the user can play, pause and stop the TV’s video. When the video plays, a semitransparent reflection plays simultaneously on what appears to be a flat surface in front of the screen (Fig. 24.10).

The TV GUI may appear overwhelmingly complex, but it’s actually just a basic WPF GUI built using controls with modified appearances. This example demonstrates the use of WPF bitmap effects to apply simple visual effects to some of the GUI elements. In addition, it introduces opacity masks, which can be used to hide parts of an element. Other
than these two new concepts, the TV app is created using only the WPF elements and concepts that you’ve already learned. Figure 24.11 presents the XAML markup and a screen capture of the app when it first loads. The video used in this case study is a public-domain video from www.nasa.gov/multimedia/videogallery/index.html.

```
<!-- Fig. 24.11: MainWindow.xaml -->
<!-- TV GUI showing the versatility of WPF customization. -->
<Window x:Class="TV.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TV" Height="720" Width="720">
  <Window.Resources>
    <!-- define template for play, pause and stop buttons -->
    <ControlTemplate x:Key="RadioButtonTemplate" TargetType="RadioButton">
      <Grid>
        <!-- create a circular border -->
        <Ellipse Width="25" Height="25" Fill="Silver" />
        <!-- create an "illuminated" background -->
        <Ellipse Name="backgroundEllipse" Width="22" Height="22">
          <Ellipse.Fill> <!-- enabled and unchecked state -->
```

Fig. 24.10  GUI representing a television.

Fig. 24.11  TV GUI showing the versatility of WPF customization. (Part 1 of 5.)
Chapter 24   WPF Graphics and Multimedia

Fig. 24.11 | TV GUI showing the versatility of WPF customization. (Part 2 of 5.)
Fig. 24.11  |  TV GUI showing the versatility of WPF customization. (Part 3 of 5.)
Chapter 24  WPF Graphics and Multimedia

<RadialGradientBrush> <!-- dim "light" -->
  <GradientStop Offset="0"
    Color="LightGray" />
  <GradientStop Offset="1.25"
    Color="Black" />
</RadialGradientBrush>

<!-- display power-button image-->
<Image Source="Images/power.png" Width="20"
  Height="20" />
</Grid>

<!-- change appearance when state changes -->
<ControlTemplate.Triggers>
  <!-- checked state -->
  <Trigger Property="CheckBox.IsChecked"
    Value="True">
    <Setter TargetName="backgroundEllipse"
      Property="Fill">
      <Setter.Value> <!-- green "light" -->
        <RadialGradientBrush>
          <GradientStop Offset="0"
            Color="LimeGreen" />
          <GradientStop Offset="1.25"
            Color="Black" />
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Trigger>
</ControlTemplate.Triggers>
</CheckBox.Template>
</CheckBox>

<!-- skew "TV" to give a 3-D appearance -->
<Border.RenderTransform>
  <SkewTransform AngleY="15" />
</Border.RenderTransform>

<!-- apply shadow effect to "TV" -->
<Border.Effect>
  <DropShadowEffect Color="Gray" ShadowDepth="15" />
</Border.Effect>

<!-- define reflection -->
<Border Canvas.Left="185" Canvas.Top="410" Height="300"
  Width="400">
  <Rectangle Name="reflectionRectangle">
  <Rectangle.Fill>
  </Rectangle.Fill>
</Border>

Fig. 24.11  |  TV GUI showing the versatility of WPF customization. (Part 4 of 5.)
24.7 WPF Customization: A Television GUI

```xml
<Rectangle.Fill>
    <VisualBrush Visual="{Binding ElementName=videoMediaElement}"
                   VisualBrush.RelativeTransform>
        <ScaleTransform ScaleY="-1" CenterY="0.5" />
    </VisualBrush.RelativeTransform>
</Rectangle.Fill>

<Rectangle.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="Black" Offset="-0.25" />
        <GradientStop Color="Transparent" Offset="0.5" />
    </LinearGradientBrush>
</Rectangle.OpacityMask>

<!-- skew reflection to look 3-D -->
<Border.RenderTransform>
    <SkewTransform AngleY="15" AngleX="-45" />
</Border.RenderTransform>
```

**Fig. 24.11** TV GUI showing the versatility of WPF customization. (Part 5 of 5.)
Chapter 24  WPF Graphics and Multimedia

WPF Effects
WPF allows you to apply graphical effects to any GUI element. There are two predefined effects—the DropShadowEffect, which gives an element a shadow as if a light were shining at it (Fig. 24.11, lines 163–165), and the BlurEffect, which makes an element's appearance blurry. The System.Windows.Media.Effects namespace also contains the more generalized ShaderEffect class, which allows you to build and use your own custom shader effects. For more information on the ShaderEffect class, visit Microsoft's developer center:

bit.ly/shadereffect

You can apply an effect to any element by setting its Effect property. Each Effect has its own unique properties. For example, DropShadowEffect's ShadowDepth property specifies the distance from the element to the shadow (line 164), while a BlurEffect's KernelType property specifies the type of blur filter it uses and its Radius property specifies the filter's size.

Creating Buttons on the TV
The representations of TV buttons in this example are not Button controls. The play, pause, and stop buttons are RadioButton, and the power button is a CheckBox. Lines 9–55 and 110–152 define the ControlTemplates used to render the RadioButtons and CheckBox, respectively. The two templates are defined similarly, so we discuss only the RadioButton template in detail.

In the background of each button are two circles, defined by Ellipse objects. The larger Ellipse (line 13) acts as a border. The smaller Ellipse (lines 16–23) is colored by a RadialGradientBrush. The gradient is a light color in the center and becomes black as it extends farther out. This makes it appear to be a source of light. The content of the RadioButton is then applied on top of the two Ellipses (line 26).

The images used in this example are transparent outlines of the play, pause, and stop symbols on a black background. When the button is applied over the RadialGradientBrush, it appears to be illuminated. In its default state (enabled and unchecked), each playback button glows red. This represents the TV being on, with the playback option not active. When the app first loads, the TV is off, so the playback buttons are disabled. In this state, the background gradient is gray. When a playback option is active (i.e., RadioButton is checked), it glows green. The latter two deviations in appearance when the control changes states are defined by triggers (lines 30–54).

The power button, represented by a CheckBox, behaves similarly. When the TV is off (i.e., CheckBox is unchecked), the control is gray. When the user presses the power button and turns the TV on (i.e., CheckBox becomes checked), the control turns green. The power button is never disabled.

Creating the TV Interface
The TV panel is represented by a beveled Border with a gray background (lines 61–166). Recall that a Border is a ContentControl and can host only one direct child element. Thus, all of the Border's elements are contained in a Grid layout container. Nested within the TV panel is another Border with a black background containing a MediaElement control (lines 70–76). This portrays the TV's screen. The power button is placed in the bot-
Creating the Reflection of the TV Screen

Lines 169–196 define the GUI’s video reflection using a Rectangle element nested in a Border. The Rectangle’s Fill is a VisualBrush that’s bound to the MediaElement (lines 172–180). To invert the video, we define a ScaleTransform and specify it as the RelativeTransform property, which is common to all brushes (lines 176–178). You can invert an element by setting the ScaleX or ScaleY—the amounts by which to scale the respective coordinates—property of a ScaleTransform to a negative number. In this example, we set ScaleY to -1 and CenterY to 0.5, inverting the VisualBrush vertically centered around the midpoint. The CenterX and CenterY properties specify the point from which the image expands or contracts. When you scale an image, most of the points move as a result of the altered size. The center point is the only point that stays at its original location when ScaleX and ScaleY are set to values other than 1.

To achieve the semitransparent look, we applied an opacity mask to the Rectangle by setting the OpacityMask property (lines 184–189). The mask uses a LinearGradientBrush that changes from black near the top to transparent near the bottom. When the gradient is applied as an opacity mask, the gradient translates to a range from completely opaque, where it’s black, to completely transparent. In this example, we set the Offset of the black GradientStop to -0.25, so that even the opaque edge of the mask is slightly transparent. We also set the Offset of the transparent GradientStop to 0.5, indicating that only the top half of the Rectangle (or bottom half of the movie) should display.

Skewing the GUI Components to Create a 3-D Look

When you draw a three-dimensional object on a two-dimensional plane, you are creating a 2-D projection of that 3-D environment. For example, to represent a simple box, you draw three adjoining parallelograms. Each face of the box is actually a flat, skewed rectangle rather than a 2-D view of a 3-D object. You can apply the same concept to create simple 3-D-looking GUIs without using a 3-D engine.

In this case study, we applied a SkewTransform to the TV representation, skewing it vertically by 15 degrees clockwise from the x-axis (lines 158–160). The reflection is then skewed (lines 193–195) vertically by 15 degrees clockwise from the x-axis (using AngleY) and horizontally by 45 degrees clockwise from the y-axis (using AngleX). Thus the GUI becomes a 2-D orthographic projection of a 3-D space with the axes 105, 120, and 135 degrees from each other, as shown in Fig. 24.12. Unlike a perspective projection, an orthographic projection does not show depth. Thus, the TV GUI does not present a realistic 3-D view, but rather a graphical representation.

Examining the Code-Behind Class

Figure 24.13 presents the code-behind class that provides the functionality for the TV app. When the user turns on the TV (i.e., checks the powerCheckBox), the reflection is made visible and the playback options are enabled (lines 16–26). When the user turns off the TV, the MediaElement’s Close method is called to close the media. In addition, the reflection is made invisible and the playback options are disabled (lines 29–45).

Whenever one of the RadioButtons that represent each playback option is checked, the MediaElement executes the corresponding task (lines 48–66). The methods that execute...
Chapter 24 WPF Graphics and Multimedia

---

**Fig. 24.12** The effect of skewing the TV app’s GUI components.

---

```csharp
// Fig. 24.13: MainWindow.xaml.cs
// TV GUI showing the versatility of WPF customization (code-behind).
using System.Windows;

namespace TV
{
    public partial class MainWindow : Window
    {
        // constructor
        public MainWindow()
        {
            InitializeComponent();
        } // end constructor

        // turns "on" the TV
        private void powerCheckBox_Checked( object sender, RoutedEventArgs e )
        {
            // render the reflection visible
            reflectionRectangle.Visibility = Visibility.Visible;

            // enable play, pause, and stop buttons
            playRadioButton.IsEnabled = true;
pauseRadioButton.IsEnabled = true;
        }
    }
}
```

---

**Fig. 24.13** TV GUI showing the versatility of WPF customization (code-behind). (Part 1 of 2.)
24.7 WPF Customization: A Television GUI

these tasks are built into the MediaElement control. Playback can be modified programmatically only if the LoadedBehavior is Manual (line 75 in Fig. 24.11).

stopRadioButton.IsEnabled = true;
} // end method powerCheckBox_Checked

// turns "off" the TV
private void powerCheckBox_Unchecked( object sender, RoutedEventArgs e )
{
    // shut down the screen
    videoMediaElement.Close();
    // hide the reflection
    reflectionRectangle.Visibility = Visibility.Hidden;
    // disable the play, pause, and stop buttons
    playRadioButton.IsChecked = false;
    pauseRadioButton.IsChecked = false;
    stopRadioButton.IsChecked = false;
    playRadioButton.IsEnabled = false;
    pauseRadioButton.IsEnabled = false;
    stopRadioButton.IsEnabled = false;
} // end method powerCheckBox_Unchecked

// plays the video
private void playRadioButton_Checked( object sender, RoutedEventArgs e )
{
    videoMediaElement.Play();
} // end method playRadioButton_Checked

// pauses the video
private void pauseRadioButton_Checked( object sender, RoutedEventArgs e )
{
    videoMediaElement.Pause();
} // end method pauseRadioButton_Checked

// stops the video
private void stopRadioButton_Checked( object sender, RoutedEventArgs e )
{
    videoMediaElement.Stop();
} // end method stopRadioButton_Checked
} // end class MainWindow
} // end namespace TV

Fig. 24.13 | TV GUI showing the versatility of WPF customization (code-behind). (Part 2 of 2.)
24.8 Animations

An animation in WPF apps simply means a transition of a property from one value to another in a specified amount of time. Most graphic properties of a control can be animated. The UsingAnimations example (Fig. 24.14) shows a video’s size being animated. A MediaElement along with two input TextBoxes—one for Width and one for Height—and an animate Button are created in the GUI. When you click the animate Button, the video’s Width and Height properties animate to the values typed in the corresponding TextBoxes by the user.

```xml
<Window x:Class="UsingAnimations.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="UsingAnimations" Height="400" Width="500">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <MediaElement Name="video" Height="100" Width="100"
                  Stretch="UniformToFill" Source="media.mp4" />
    <StackPanel Grid.Column="1">
      <!-- TextBox will contain the new Width for the video -->
      <TextBlock TextWrapping="Wrap" Text="Width:" Margin="5,0,0,0" />
      <TextBox Name="widthValue" Width="75" Margin="5">100</TextBox>
      <!-- TextBox will contain the new Height for the video -->
      <TextBlock TextWrapping="Wrap" Text="Height:" Margin="5,0,0,0" />
      <TextBox Name="heightValue" Width="75" Margin="5">100</TextBox>
      <!-- When clicked, rectangle animates to the input values -->
      <Button Content="Animate" Width="75" Margin="5">
        <Button.Triggers>
          <!-- Use trigger to call animation -->
          <EventTrigger RoutedEvent="Button.Click">
            <BeginStoryboard>
              <Storyboard Storyboard.TargetName="video">
                <!-- Animates the Width -->
                <DoubleAnimation Duration="0:0:2"
                                  Storyboard.TargetProperty="Width"
                                  To="{Binding ElementName=widthValue, Path=Text}" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger>
          <!-- When button is clicked -->
          <EventTrigger RoutedEvent="Button.Click">
            <BeginStoryboard>
              <Storyboard Storyboard.TargetName="video">
                <!-- Animates the Height -->
                <DoubleAnimation Duration="0:0:2"
                                  Storyboard.TargetProperty="Height"
                                  To="{Binding ElementName=heightValue, Path=Text}" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger>
        </Button.Triggers>
      </Button>
    </StackPanel>
  </Grid>
</Window>
```

Fig. 24.14 | Animating graphic elements with Storyboards. (Part 1 of 2.)
As you can see, the animations create a smooth transition from the original Height and Width to the new values. Lines 31–43 define a **Storyboard** element embedded in the
Chapter 24  WPF Graphics and Multimedia

Button’s click event Trigger. A Storyboard contains embedded animation elements. When the Storyboard begins executing (line 30), all embedded animations execute. A Storyboard has two important properties—TargetName and TargetProperty. The TargetName (line 31) specifies which control to animate. The TargetProperty specifies which property of the animated control to change. In this case, the Width (line 34) and Height (line 40) are the TargetProperties, because we’re changing the size of the video. Both the TargetName and TargetProperty can be defined in the Storyboard or in the animation element itself.

To animate a property, you can use one of several animation classes available in WPF. We use the DoubleAnimation for the size properties—PointAnimations and ColorAnimations are two other commonly used animation classes. A DoubleAnimation animates properties of type Double. The Width and Height animations are defined in lines 33–36 and 39–42, respectively. Lines 35–36 define the To property of the Width animation, which specifies the value of the Width at the end of the animation. We use data binding to set this to the value in the widthValue TextBox. The animation also has a Duration property that specifies how long the animation takes. Notice in line 33 that we set the Duration of the Width animation to 0:0:2, meaning the animation takes 0 hours, 0 minutes and 2 seconds. You can specify fractions of a second by using a decimal point. Hour and minute values must be integers. Animations also have a From property which defines a constant starting value of the animated property.

Since we’re animating the video’s Width and Height properties separately, it’s not always displayed at its original width and height. In line 14, we define the MediaElement’s Stretch property. This is a property for graphic elements and determines how the media stretches to fit the size of its enclosure. This property can be set to None, Uniform, UniformToFill or Fill. None allows the media to stay at its native size regardless of the container’s size. Uniform resizes the media to its largest possible size while maintaining its native aspect ratio. UniformToFill resizes the media to completely fill the container while still keeping its aspect ratio—as a result, it could be cropped. When an image or video is cropped, the pieces of the edges are cut off from the media in order to fit the shape of the container. Fill forces the media to be resized to the size of the container (aspect ratio is not preserved). In the example, we use Fill to show the changing size of the container.

24.9 Speech Synthesis and Speech Recognition

Speech-based interfaces make computers easier to use for people with disabilities (and others). Speech synthesizers, or text-to-speech (TTS) systems, read text out loud and are an ideal method for communicating information to sight-impaired individuals. Speech recognizers, or speech-to-text (STT) systems, transform human speech (input through a microphone) into text and are a good way to gather input or commands from users who have difficulty with keyboards and mice. .NET provides powerful tools for working with speech synthesis and recognition. The program shown in Figs. 24.15–24.16 provides explanations of the various kinds of programming tips found in this book using an STT system (and the mouse) as input and a TTS system (and text) as output.

Our speech app’s GUI (Fig. 24.15) consists of a vertical StackPanel containing a TextBox, a Button and a series of horizontal StackPanels containing Images and TextBlocks that label those Images.
24.9 Speech Synthesis and Speech Recognition

Fig. 24.15 | Text-To-Speech and Speech-To-Text. (Part 1 of 2.)
Chapter 24  WPF Graphics and Multimedia

Figure 24.16 provides the speech app’s functionality. The user either clicks an Image or speaks its name into a microphone, then the GUI displays a text description of the concept which that image or phrase represents, and a speech synthesizer speaks this description. To use .NET’s speech synthesis and recognition classes, you must add a reference to System.Speech as follows:

1. Right click the project name in the Solution Explorer then select Add Reference....
2. In the Reference Manager dialog under Assemblies > Framework, locate and select System.Speech and click OK.


```
// Fig. 24.16: MainWindow.xaml.cs
// Text-To-Speech and Speech-To-Text code-behind file. (Part 1 of 4.)

using System;
```
using System.Collections.Generic;
using System.Speech.Synthesis;
using System.Speech.Recognition;
using System.Windows;
using System.Windows.Controls;

namespace SpeechApp
{
    public partial class MainWindow : Window
    {
        // keeps track of which description is to be printed and spoken
        private string displayString;

        public MainWindow()
        {
            InitializeComponent();

            // define the input phrases

            // add the phrases to a Choices collection
            Choices theChoices = new Choices( phrases );

            // build a Grammar around the Choices and set up the listener to use this grammar
            myGrammar = new Grammar( new GrammarBuilder( theChoices ) );
            listener.Enabled = true;
            listener.LoadGrammar( myGrammar );
            myGrammar.SpeechRecognized += myGrammar_SpeechRecognized;
        }
    }
}
Chapter 24  WPF Graphics and Multimedia

// define the descriptions for each icon/phrase
string[] descriptions = {
    "Good Programming Practices highlight " +
        "techniques for writing programs that are clearer, more " +
        "understandable, more debuggable, and more maintainable.",
    "Software Engineering Observations highlight " +
        "architectural and design issues that affect the " +
        "construction of complex software systems.",
    "Performance Tips highlight opportunities " +
        "for improving program performance.",
    "Portability Tips help students write " +
        "portable code that can execute on different platforms.",
    "Look-and-Feel Observations highlight " +
        "graphical user interface conventions. These " +
        "observations help students design their own graphical " +
        "user interfaces in conformance with industry standards.",
    "Error-Prevention Tips tell people how to " +
        "test and debug their programs. Many of the tips also " +
        "describe aspects of creating programs that " +
        "reduce the likelihood of 'bugs' and thus simplify the " +
        "testing and debugging process.",
    "Common Programming Errors focus the " +
        "students' attention on errors commonly made by " +
        "beginning programmers. This helps students avoid " +
        "making the same errors. It also helps reduce the long " +
        "lines outside instructors' offices during " +
        "office hours!"};

// map each image to its corresponding description
imageDescriptions.Add( GoodPracticesImage, descriptions[ 0 ] );
imageDescriptions.Add( ObservationsImage, descriptions[ 1 ] );
imageDescriptions.Add( PerformanceImage, descriptions[ 2 ] );
imageDescriptions.Add( PortabilityImage, descriptions[ 3 ] );
imageDescriptions.Add( LookAndFeelImage, descriptions[ 4 ] );
imageDescriptions.Add( PreventionImage, descriptions[ 5 ] );
imageDescriptions.Add( ErrorImage, descriptions[ 6 ] );

// loop through the phrases and descriptions and map accordingly
for ( int index = 0; index <= 6; ++index )
    phraseDescriptions.Add( phrases[ index ],
                            descriptions[ index ] );

talker.Rate = -4; // slows down the speaking rate
}

// when the user clicks on the speech-synthesis button, speak the
// contents of the related text box
private void SpeechButton_Click( object sender, RoutedEventArgs e )
{
    talker.SpeakAsync( SpeechBox.Text );
}

Fig. 24.16  |  Text-To-Speech and Speech-To-Text code-behind file. (Part 3 of 4.)
Instance Variables
You can now add instance variables of types `SpeechRecognizer`, `Grammar` and `SpeechSynthesizer` (lines 15, 18 and 21). The `SpeechRecognizer` class has several ways to recognize input phrases. The most reliable involves building a `Grammar` containing the exact phrases that the `SpeechRecognizer` can receive as spoken input. The `SpeechSynthesizer` object speaks text, using one of several `voices`. Variable `displayString` (line 24) keeps track of the description that will be displayed and spoken. Lines 27–28 and 31–32 declare two objects of type `Dictionary` (namespace `System.Collections.Generic`). A `Dictionary` is a collection of `key–value` pairs, in which each key has a corresponding value. The `Dictionary` objects associate each input phrase and each clickable `Image` with the corresponding description phrase to be displayed and spoken.

Constructor
In the constructor (lines 34–97), the app initializes the input phrases and places them in a `Choices` collection (lines 39–45). A `Choices` collection is used to build a `Grammar` (lines 106–107).
49–51). Line 52 registers the listener for the Grammar's SpeechRecognized event. Lines 55–80 create an array of the programming-tip descriptions. Lines 83–89 add each image and its corresponding description to the ImageDescriptions Dictionary. Lines 92–94 add each programming-tip name and corresponding description to the phraseDescriptions Dictionary. Finally, line 96 sets the SpeechSynthesizer object's Rate property to -4 to slow down the default rate of speech.

**Method SpeechButton_Click**

Method SpeechButton_Click (lines 101–104) calls the SpeechSynthesizer's SpeakAsync method to speak the contents of SpeechBox. SpeechSynthesizers also have a Speak method, which is not asynchronous, and SpeakSml and SpeakSmlAsynch, methods specifically for use with Speech Synthesis Markup Language (SSML)—an XML vocabulary created particularly for TTS systems. For more information on SSML, visit [www.xml.com/pub/a/2004/10/20/ssml.html](http://www.xml.com/pub/a/2004/10/20/ssml.html).

**Method Image_MouseDown**

Method Image_MouseDown (lines 106–113) handles the MouseDown events for all the Image objects. When the user clicks an Image, the program casts sender to type Image, then passes the results as input into the ImageDescriptions Dictionary to retrieve the corresponding description string. This string is assigned to displayString (line 111). We then call DisplaySpeak to display displayString at the bottom of the window and cause the SpeechSynthesizer to speak it.

**Method myGrammar_SpeechRecognized**

Method myGrammar_SpeechRecognized (lines 117–127) is called whenever the SpeechRecognizer detects that one of the input phrases defined in myGrammar was spoken. The Result property of the RecognitionEventArgs parameter contains the recognized text. We use the phraseDescriptions Dictionary object to determine which description to display (line 122). We cannot call DisplaySpeak directly here, because GUI events and the SpeechRecognizer events operate on different threads—they are processes being executed in parallel, independently from one another and without access to each other's methods. Every method that modifies the GUI must be called via the GUI thread of execution. To do this, we use a Dispatcher object (lines 125–126) to invoke the method. The method to call must be wrapped in a so-called delegate object. An Action delegate object represents a method with no parameters.

**Method DisplaySpeak**

Method DisplaySpeak (lines 131–135) outputs displayString to the screen by updating InfoBlock's Text property and to the speakers by calling the SpeechSynthesizer's SpeakAsync method.

### 24.10 Wrap-Up

In this chapter you learned how to manipulate graphic elements in your WPF app. We introduced how to control fonts using the properties of TextBlocks. You learned to change the TextBlock's FontFamily, FontSize, FontWeight and FontStyle in XAML. We also demonstrated the TextDecorations Underline, Overline, Baseline and
Strikethrough. Next, you learned how to create basic shapes such as Lines, Rectangles and Ellipses. You set the Fill and Stroke of these shapes. We then discussed an app that created a Polyline and two Polygons. These controls allow you to create multisided objects using a set of Points in a PointCollection.

You learned that there are several types of brushes for customizing an object’s Fill. We demonstrated the SolidColorBrush, the ImageBrush, the VisualBrush and the LinearGradientBrush. Though the VisualBrush was used only with a MediaElement, this brush has a wide range of capabilities (msdn.microsoft.com/library/ms749021.aspx).

We explained how to apply transforms to an object to reposition or reorient any graphic element. You used transforms such as the TranslateTransform, the RotateTransform, the SkewTransform and the ScaleTransform to manipulate various controls.

The television GUI app used ControlTemplates and BitmapEffects to create a completely customized 3-D-looking television set. You saw how to use ControlTemplates to customize the look of RadioButtons and CheckBoxes. The app also included an opacity mask, which can be used on any shape to define the opaque or transparent regions of the control. Opacity masks are particularly useful with images and video where you cannot change the Fill to directly control transparency.

We showed how animations can be applied to transition properties from one value to another. Common 2-D animation types include DoubleAnimations, PointAnimations and ColorAnimations.

Finally, we introduced the speech synthesis and speech recognition APIs. You learned how to make computers speak text and receive voice input. You also learned how to create a Grammar of phrases that the user can speak to control the program.