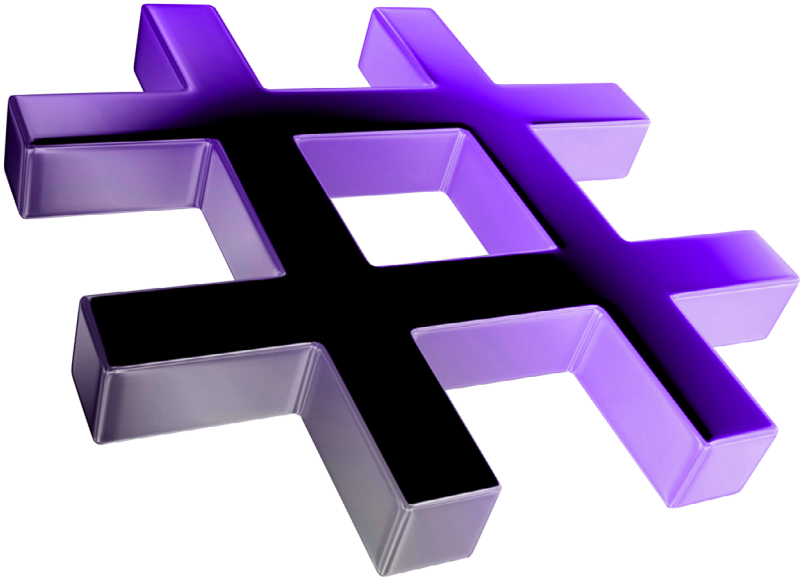
WPF Graphics and Multimedia

26



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.

Outline

26.1 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.

26.2 Controlling Fonts

This section introduces how to control fonts by modifying the font properties of a **`TextBlock`** control in the XAML code. Figure 26.1 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.

```
1  <!-- Fig. 33.1: MainWindow.xaml -->
2  <!-- Formatting fonts in XAML code. -->
3  <Window x:Class="UsingFonts.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="UsingFonts" Height="120" Width="400">
7
8     <StackPanel>
9        <!-- make a font bold using the FontWeight -->
10       <TextBlock TextWrapping="Wrap" Text="Arial 14 bold."
11          FontFamily="Arial" FontSize="14" FontWeight="Bold"/>
12
13       <!-- if no font size is specified, default size is used -->
14       <TextBlock TextWrapping="Wrap"
15          Text="Times New Roman plain, default size."
16          FontFamily="Times New Roman"/>
17
18       <!-- specifying a different font size and using FontStyle -->
19       <TextBlock TextWrapping="Wrap"
20          Text="Courier New 16 bold and italic."
21          FontFamily="Courier New" FontSize="16"
22          FontStyle="Italic" FontWeight="Bold"/>
23
24       <!-- using Overline and Baseline TextDecorations -->
25       <TextBlock TextWrapping="Wrap"
26          Text="Default font with overline and baseline.">
27          <TextBlock.TextDecorations>
28             <TextDecoration Location="Overline"/>
29             <TextDecoration Location="Baseline"/>
30          </TextBlock.TextDecorations>
31       </TextBlock>
32
33       <!-- using Strikethrough and Underline TextDecorations -->
34       <TextBlock TextWrapping="Wrap"
35          Text="Default font with strikethrough and underline.">
36          <TextBlock.TextDecorations>
37             <TextDecoration Location="Strikethrough"/>
38             <TextDecoration Location="Underline"/>
39          </TextBlock.TextDecorations>
40       </TextBlock>
41    </StackPanel>
42 </Window>
```

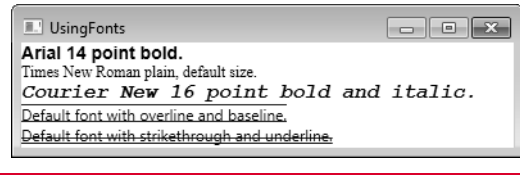


Fig. 26.1 | Formatting fonts in XAML code.

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.

`TextBlock`s 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. 26.1—create lines above the text and at the base of the text, respectively (lines 28–29). **`Strikethrough`** 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.

26.3 Basic Shapes

WPF has several built-in shapes. The `BasicShapes` example (Fig. 26.2) shows you how to display `Line`s, `Rectangle`s and `Ellipse`s 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.

```
1  <!-- Fig. 26.2: MainWindow.xaml -->
2  <!-- Drawing basic shapes in XAML. -->
3  <Window x:Class="BasicShapes.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="BasicShapes" Height="200" Width="500">
7     <Canvas>
8        <!-- Rectangle with fill but no stroke -->
9        <Rectangle Canvas.Left="90" Canvas.Top="30" Width="150" Height="90"
10          Fill="LightBlue" />
11
12       <!-- Lines defined by starting points and ending points-->
13       <Line X1="90" Y1="30" X2="110" Y2="40" Stroke="Black" />
14       <Line X1="90" Y1="120" X2="110" Y2="130" Stroke="Black" />
15       <Line X1="240" Y1="30" X2="260" Y2="40" Stroke="Black" />
16       <Line X1="240" Y1="120" X2="260" Y2="130" Stroke="Black" />
```

Fig. 26.2 | Drawing basic shapes in XAML. (Part 1 of 2.)

```
17
18          <!-- Rectangle with stroke but no fill -->
19          <Rectangle Canvas.Left="110" Canvas.Top="40" Width="150"
20             Height="90" Stroke="Black" />
21
22          <!-- Ellipse with fill and no stroke -->
23          <Ellipse Canvas.Left="280" Canvas.Top="75" Width="100" Height="50"
24             Fill="Orange" />
25          <Line X1="380" Y1="55" X2="380" Y2="100" Stroke="Black" />
26          <Line X1="280" Y1="55" X2="280" Y2="100" Stroke="Black" />
27
28          <!-- Ellipse with stroke and no fill -->
29          <Ellipse Canvas.Left="280" Canvas.Top="30" Width="100" Height="50"
30             Stroke="Black" />
31       </Canvas>
32    </Window>
```

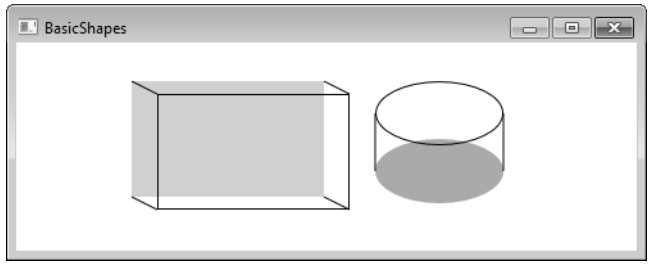


Fig. 26.2 | Drawing basic shapes in XAML. (Part 2 of 2.)

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. `Rectangle`s 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`. `Line`s have a `Stroke` property that defines the *color* of the line. In this example, the lines are all set to have black `Stroke`s (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).

26.4 Polygons and Polylines

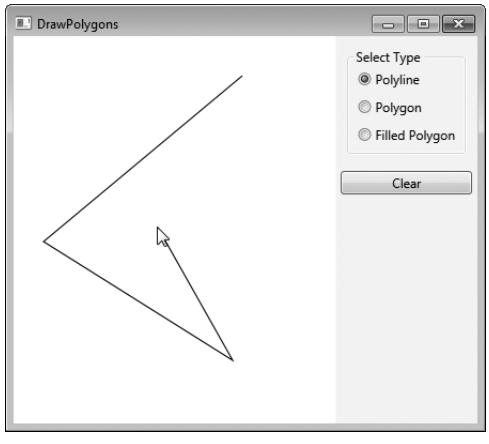
There are two shape controls for drawing *multisided shapes*—**`Polyline`** and **`Polygon`**. `Polyline` draws a series of connected lines defined by a set of points, while `Polygon` does the same but connects the start and end points to make a closed figure. The app `DrawPolygons` (Fig. 26.3) allows 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 `RadioButtons` 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.

```
1  <!-- Fig. 26.3: MainWindow.xaml -->
2  <!-- Defining Polylines and Polygons in XAML. -->
3  <Window x:Class="DrawPolygons.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="DrawPolygons" Height="400" Width="450" Name="mainWindow">
7     <Grid>
8        <Grid.ColumnDefinitions>
9           <ColumnDefinition />
10          <ColumnDefinition Width="Auto" />
11       </Grid.ColumnDefinitions>
12
13       <!-- Canvas contains two polygons and a polyline -->
14       <!-- only the shape selected by the radio button is visible -->
15       <Canvas Name="drawCanvas" Grid.Column="0" Background="White"
16          MouseDown="drawCanvas_MouseDown">
17          <Polyline Name="polyLine" Stroke="Black"
18             Visibility="Collapsed" />
19          <Polygon Name="polygon" Stroke="Black" Visibility="Collapsed" />
20          <Polygon Name="filledPolygon" Fill="DarkBlue"
21             Visibility="Collapsed" />
22       </Canvas>
23
24       <!-- StackPanel containing the RadioButton options -->
25       <StackPanel Grid.Column="1" Orientation="Vertical"
26          Background="WhiteSmoke">
27          <GroupBox Header="Select Type" Margin="10">
28             <StackPanel>
29                <!-- Polyline option -->
30                <RadioButton Name="lineRadio" Content="Polyline"
31                   Margin="5" Checked="lineRadio_Checked"/>
32
33                <!-- unfilled Polygon option -->
34                <RadioButton Name="polygonRadio" Content="Polygon"
35                   Margin="5"Checked="polygonRadio_Checked"/>
36
```

Fig. 26.3 | Defining Polylines and Polygons in XAML. (Part 1 of 2.)

```
37                  <!-- filled Polygon option -->
38                  <RadioButton Name="filledPolygonRadio"
39                     Content="Filled Polygon" Margin="5"
40                     Checked="filledPolygonRadio_Checked"/>
41               </StackPanel>
42            </GroupBox>
43
44            <!-- Button clears the shape from the canvas -->
45            <Button Name="clearButton" Content="Clear"
46               Click="clearButton_Click" Margin="5"/>
47         </StackPanel>
48      </Grid>
49   </Window>
```

a) App with the Polyline option selected



b) App with the Filled Polygon option selected

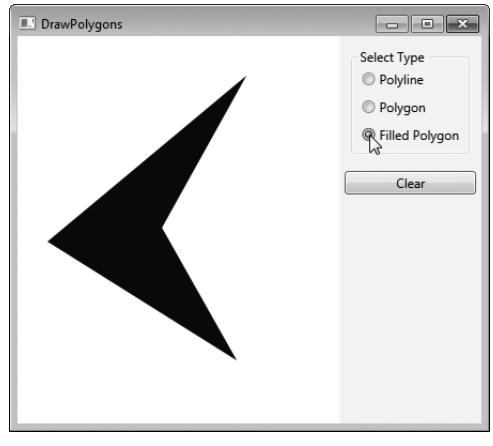


Fig. 26.3 | Defining Polylines and Polygons in XAML. (Part 2 of 2.)

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. 26.4.

To allow the user to specify a variable number of points, line 12 in Fig. 26.4 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

```
1  // Fig. 26.4: MainWindow.xaml.cs
2  // Drawing Polylines and Polygons.
3  using System.Windows;
4  using System.Windows.Input;
5  using System.Windows.Media;
6
7  namespace DrawPolygons
8  {
9     public partial class MainWindow : Window
10    {
11       // stores the collection of points for the multisided shapes
12       private PointCollection points = new PointCollection();
13
14       // initialize the points of the shapes
15       public MainWindow()
16       {
17          InitializeComponent();
18
19          polyLine.Points = points; // assign Polyline points
20          polygon.Points = points; // assign Polygon points
21          filledPolygon.Points = points; // assign filled Polygon points
22       } // end constructor
23
24       // adds a new point when the user clicks on the canvas
25       private void drawCanvas_MouseDown( object sender,
26          MouseButtonEventArgs e )
27       {
28          // add point to collection
29          points.Add( e.GetPosition( drawCanvas ) );
30       } // end method drawCanvas_MouseDown
31
32       // when the clear Button is clicked
33       private void clearButton_Click( object sender, RoutedEventArgs e )
34       {
35          points.Clear(); // clear the points from the collection
36       } // end method clearButton_Click
37
38       // when the user selects the Polyline
39       private void lineRadio_Checked( object sender, RoutedEventArgs e )
40       {
41          // Polyline is visible, the other two are not
42          polyLine.Visibility = Visibility.Visible;
43          polygon.Visibility = Visibility.Collapsed;
44          filledPolygon.Visibility = Visibility.Collapsed;
45       } // end method lineRadio_Checked
46
47       //  when the user selects the Polygon
48       private void polygonRadio_Checked( object sender,
49          RoutedEventArgs e )
50       {
51          // Polygon is visible, the other two are not
52          polyLine.Visibility = Visibility.Collapsed;
53          polygon.Visibility = Visibility.Visible;
```

Fig. 26.4 | Drawing `Polylines` and `Polygons`. (Part 1 of 2.)

```
54             filledPolygon.Visibility = Visibility.Collapsed;
55          } // end method polygonRadio_Checked
56
57          // when the user selects the filled Polygon
58          private void filledPolygonRadio_Checked( object sender,
59             RoutedEventArgs e )
60          {
61             // filled Polygon is visible, the other two are not
62             polyLine.Visibility = Visibility.Collapsed;
63             polygon.Visibility = Visibility.Collapsed;
64             filledPolygon.Visibility = Visibility.Visible;
65          } // end method filledPolygonRadio_Checked
66       } // end class MainWindow
67    } // end namespace DrawPolygons
```

Fig. 26.4 | Drawing `Polylines` and `Polygons`. (Part 2 of 2.)

collection. 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.

26.5 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. 26.5) to see `Brushes` applied to `TextBlocks` and `Ellipses`.

```
1  <!-- Fig. 26.5: MainWindow.xaml -->
2  <!-- Applying brushes to various XAML elements. -->
3  <Window x:Class="UsingBrushes.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="UsingBrushes" Height="470" Width="720">
```

Fig. 26.5 | Applying brushes to various XAML elements. (Part 1 of 4.)

```
7     <Grid>
8        <Grid.RowDefinitions>
9           <RowDefinition />
10          <RowDefinition />
11          <RowDefinition />
12       </Grid.RowDefinitions>
13
14       <Grid.ColumnDefinitions>
15          <ColumnDefinition />
16          <ColumnDefinition />
17       </Grid.ColumnDefinitions>
18
19       <!-- TextBlock with a SolidColorBrush -->
20       <TextBlock TextWrapping="Wrap" Text="Color" FontSize="100"
21          FontWeight="999">
22          <TextBlock.Foreground>
23             <SolidColorBrush Color="#FF5F2CAE" />
24          </TextBlock.Foreground>
25       </TextBlock>
26
27       <!-- Ellipse with a SolidColorBrush (just a Fill) -->
28       <Ellipse Grid.Column="1" Width="300" Height="100" Fill="#FF5F2CAE"/>
29
30       <!-- TextBlock with an ImageBrush -->
31       <TextBlock TextWrapping="Wrap" Text="Image" Grid.Row="1"
32          FontSize="100" FontWeight="999">
33          <TextBlock.Foreground>
34             <!-- Flower image as an ImageBrush -->
35             <ImageBrush ImageSource="flowers.jpg"
36                Stretch="UniformToFill"/>
37          </TextBlock.Foreground>
38       </TextBlock>
39
40       <!-- Ellipse with an ImageBrush -->
41       <Ellipse Grid.Row="1" Grid.Column="1" Width="300" Height="100">
42          <Ellipse.Fill>
43             <ImageBrush ImageSource="flowers.jpg"
44                Stretch="UniformToFill"/>
45          </Ellipse.Fill>
46       </Ellipse>
47
48       <!-- TextBlock with a MediaElement as a VisualBrush -->
49       <TextBlock TextWrapping="Wrap" Text="Video" Grid.Row="2"
50          FontSize="100" FontWeight="999">
51          <TextBlock.Foreground>
52             <!-- VisualBrush with an embedded MediaElement-->
53             <VisualBrush Stretch="UniformToFill">
54                <VisualBrush.Visual>
55                   <MediaElement Source="media.mp4"/>
56                </VisualBrush.Visual>
57             </VisualBrush>
58          </TextBlock.Foreground>
59       </TextBlock>
```

Fig. 26.5 | Applying brushes to various XAML elements. (Part 2 of 4.)

```
60
61          <!-- Ellipse with a MediaElement as a VisualBrush -->
62          <Ellipse Grid.Row="2" Grid.Column="1" Width="300" Height="100">
63             <Ellipse.Fill>
64                <VisualBrush Stretch="UniformToFill">
65                   <VisualBrush.Visual>
66                      <MediaElement Source="media.mp4" IsMuted="True"/>
67                   </VisualBrush.Visual>
68                </VisualBrush>
69             </Ellipse.Fill>
70          </Ellipse>
71       </Grid>
72    </Window>
```

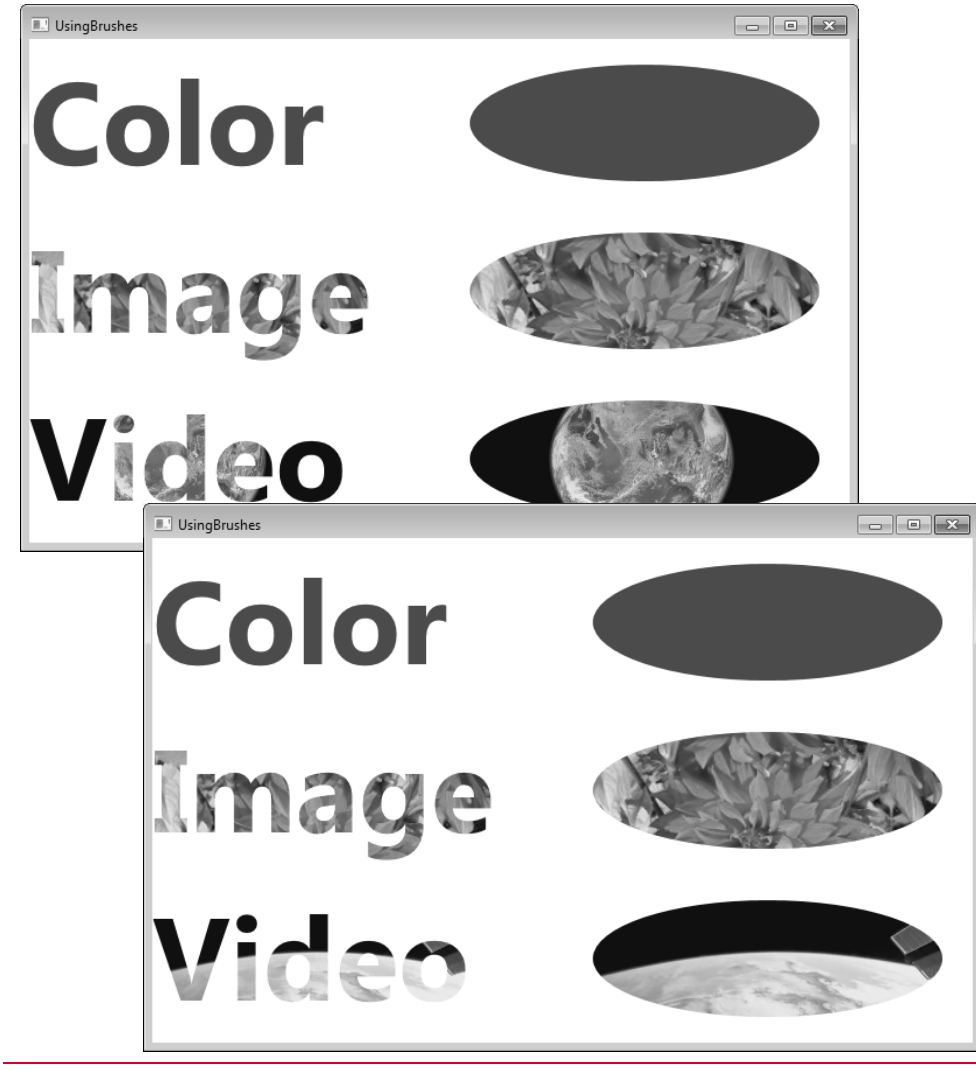


Fig. 26.5 | Applying brushes to various XAML elements. (Part 3 of 4.)

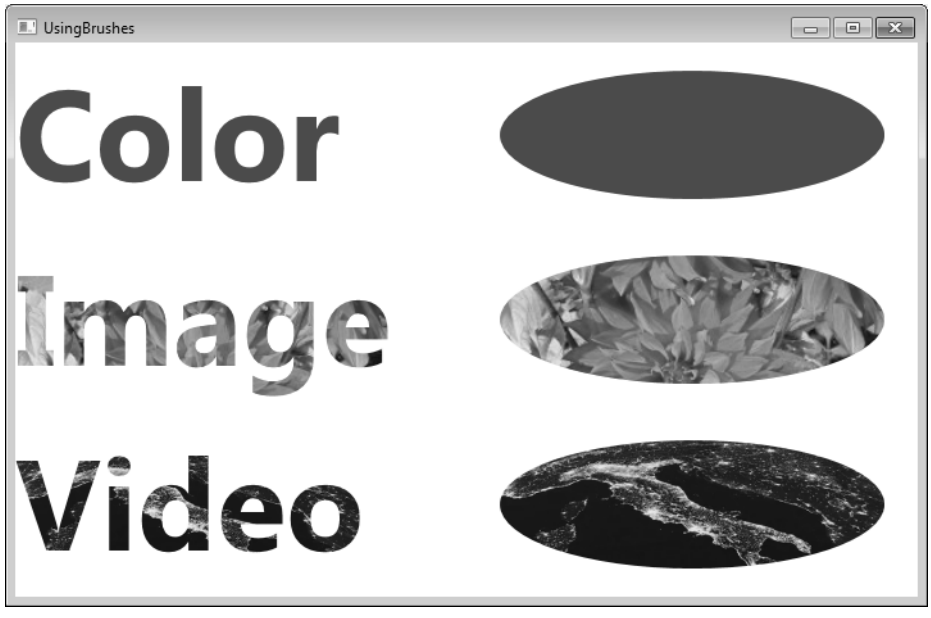


Fig. 26.5 | Applying brushes to various XAML elements. (Part 4 of 4.)

`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-

erty 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. 26.5, 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. 26.6–26.7), 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 `TextBox`es 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. 26.6.

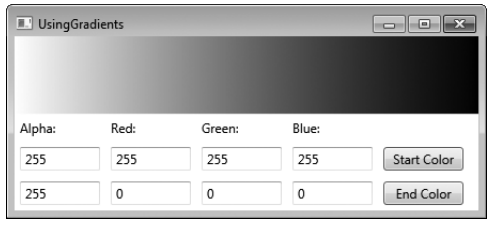
```
1  <!-- Fig. 26.6: MainWindow.xaml -->
2  <!-- Defining gradients in XAML. -->
3  <Window x:Class="UsingGradients.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="UsingGradients" Height="200" Width="450">
7     <Grid>
8        <Grid.RowDefinitions>
9           <RowDefinition />
10          <RowDefinition Height="Auto" />
11          <RowDefinition Height="Auto" />
12          <RowDefinition Height="Auto" />
13       </Grid.RowDefinitions>
14
15       <!-- TextBlock in the background to show transparency -->
16       <TextBlock TextWrapping="Wrap" Text="Transparency Test"
17          FontSize="30" HorizontalAlignment="Center"
18          VerticalAlignment="Center"/>
19
20       <!-- sample rectangle with linear gradient fill -->
21       <Rectangle>
22          <Rectangle.Fill>
```

Fig. 26.6 | Defining gradients in XAML. (Part 1 of 3.)

```
23              <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
24                 <!-- gradient stop can define a color at any offset -->
25                 <GradientStop x:Name="startGradient" Offset="0.0"
26                    Color="White" />
27                 <GradientStop x:Name="stopGradient" Offset="1.0"
28                    Color="Black" />
29              </LinearGradientBrush>
30           </Rectangle.Fill>
31        </Rectangle>
32
33        <!-- shows which TextBox corresponds with which ARGB value-->
34        <StackPanel Grid.Row="1" Orientation="Horizontal">
35           <TextBlock TextWrapping="Wrap" Text="Alpha:"
36              Width="75" Margin="5">
37           <TextBlock TextWrapping="Wrap" Text="Red:"
38              Width="75" Margin="5">
39           <TextBlock TextWrapping="Wrap" Text="Green:"
40              Width="75" Margin="5">
41           <TextBlock TextWrapping="Wrap" Text="Blue:"
42              Width="75" Margin="5">
43        </StackPanel>
44
45        <!-- GUI to select the color of the first GradientStop -->
46        <StackPanel Grid.Row="2" Orientation="Horizontal">
47           <TextBox Name="fromAlpha" TextWrapping="Wrap" Text="255"
48              Width="75" Margin="5"/>
49           <TextBox Name="fromRed" TextWrapping="Wrap" Text="255"
50              Width="75" Margin="5"/>
51           <TextBox Name="fromGreen" TextWrapping="Wrap" Text="255"
52              Width="75" Margin="5"/>
53           <TextBox Name="fromBlue" TextWrapping="Wrap" Text="255"
54              Width="75" Margin="5"/>
55           <Button Name="fromButton" Content="Start Color" Width="75"
56              Margin="5" Click="fromButton_Click"/>
57        </StackPanel>
58
59        <!-- GUI to select the color of second GradientStop -->
60        <StackPanel Grid.Row="3" Orientation="Horizontal">
61           <TextBox Name="toAlpha" TextWrapping="Wrap" Text="255"
62              Width="75" Margin="5"/>
63           <TextBox Name="toRed" TextWrapping="Wrap" Text="0"
64              Width="75" Margin="5"/>
65           <TextBox Name="toGreen" TextWrapping="Wrap" Text="0"
66              Width="75" Margin="5"/>
67           <TextBox Name="toBlue" TextWrapping="Wrap" Text="0"
68              Width="75" Margin="5"/>
69           <Button Name="toButton" Content="End Color" Width="75"
70              Margin="5" Click="toButton_Click"/>
71        </StackPanel>
72     </Grid>
73  </Window>
```

Fig. 26.6 | Defining gradients in XAML. (Part 2 of 3.)

a) The app immediately after it's loaded



b) The app after changing the start and end colors

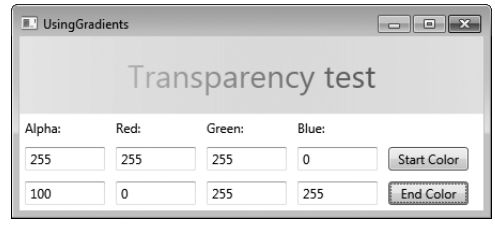


Fig. 26.6 | Defining gradients in XAML. (Part 3 of 3.)

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. 26.7 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. 26.7, lines 21–25). When the `toButton` is clicked, we do the same for `stopGradient`'s `Color` (lines 32–36).

```
1  // Fig. 26.7: MainWindow.xaml.cs
2  // Customizing gradients.
3  using System;
4  using System.Windows;
5  using System.Windows.Media;
6
7  namespace UsingGradients
8  {
9     public partial class MainWindow : Window
10    {
```

Fig. 26.7 | Customizing gradients. (Part 1 of 2.)

```
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. 26.7 | Customizing gradients. (Part 2 of 2.)

26.6 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 *shears*) the object. A `ScaleTransform` scales the object's *x*- and *y*-coordinate points by different specified amounts. See Section 26.7 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 26.8 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. 26.9.

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. 26.8. Each iteration of the loop duplicates `star` by creating a new `Polygon` with the same set of points (Fig. 26.9, lines 22–23). To generate the random colors for each star, we use the `Random`

```
<!-- Fig. 26.8: 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>
```

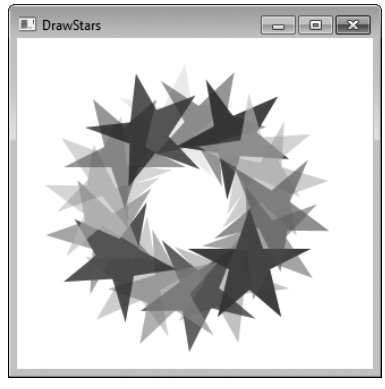


Fig. 26.8 | Defining a `Polygon` representing a star in XAML.

```
// Fig. 26.9: 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. 26.9 | Applying transforms to a `Polygon`. (Part 1 of 2.)

```
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              newStar.Fill = new SolidColorBrush( Color.FromArgb(
28                 colorValues[ 0 ], colorValues[ 1 ], colorValues[ 2 ],
29                 colorValues[ 3 ] ) ); // creates a random color brush
30
31              // apply a rotation to the shape
32              RotateTransform rotate =
33                 new RotateTransform( count * 20, 150, 150 );
34              newStar.RenderTransform = rotate;
35              mainCanvas.Children.Add( newStar );
36           } // end for
37        } // end constructor
38     } // end class MainWindow
39  } // end namespace DrawStars
```

Fig. 26.9 | 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.

26.7 WPF Customization: A Television GUI

In Chapter 32, 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. 26.10).

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

Fig. 26.10 | GUI representing a television.

than these two new concepts, the TV app is created using only the WPF elements and concepts that you've already learned. Figure 26.11 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`.

```
1  <!-- Fig. 26.11: MainWindow.xaml -->
2  <!-- TV GUI showing the versatility of WPF customization. -->
3  <Window x:Class="TV.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="TV" Height="720" Width="720">
7     <Window.Resources>
8        <!-- define template for play, pause and stop buttons -->
9        <ControlTemplate x:Key="RadioButtonTemplate"
10          TargetType="RadioButton">
11          <Grid>
12             <!-- create a circular border -->
13             <Ellipse Width="25" Height="25" Fill="Silver" />
14
15             <!-- create an "illuminated" background -->
16             <Ellipse Name="backgroundEllipse" Width="22" Height="22">
17                <Ellipse.Fill> <!-- enabled and unchecked state -->
```

Fig. 26.11 | TV GUI showing the versatility of WPF customization. (Part 1 of 5.)

```
18                   <RadialGradientBrush> <!-- red "light" -->
19                      <GradientStop Offset="0" Color="Red" />
20                      <GradientStop Offset="1.25" Color="Black" />
21                   </RadialGradientBrush>
22                </Ellipse.Fill>
23             </Ellipse>
24
25             <!-- display button image -->
26             <ContentPresenter Content="{TemplateBinding Content}" />
27          </Grid>
28
29          <!-- change appearance when state changes -->
30          <ControlTemplate.Triggers>
31             <!-- disabled state -->
32             <Trigger Property="RadioButton.IsEnabled" Value="False">
33                <Setter TargetName="backgroundEllipse" Property="Fill">
34                   <Setter.Value>
35                      <RadialGradientBrush> <!-- dim "light" -->
36                         <GradientStop Offset="0" Color="LightGray" />
37                         <GradientStop Offset="1.25" Color="Black" />
38                      </RadialGradientBrush>
39                   </Setter.Value>
40                </Setter>
41             </Trigger>
42
43             <!-- checked state -->
44             <Trigger Property="RadioButton.IsChecked" Value="True">
45                <Setter TargetName="backgroundEllipse" Property="Fill">
46                   <Setter.Value>
47                      <RadialGradientBrush> <!-- green "light" -->
48                         <GradientStop Offset="0" Color="LimeGreen" />
49                         <GradientStop Offset="1.25" Color="Black" />
50                      </RadialGradientBrush>
51                   </Setter.Value>
52                </Setter>
53             </Trigger>
54          </ControlTemplate.Triggers>
55       </ControlTemplate>
56    </Window.Resources>
57
58    <!-- define the GUI -->
59    <Canvas>
60       <!-- define the "TV" -->
61       <Border Canvas.Left="150" Height="370" Width="490"
62          Canvas.Top="20" Background="DimGray">
63          <Grid>
64             <Grid.RowDefinitions>
65                <RowDefinition />
66                <RowDefinition Height="Auto" />
67             </Grid.RowDefinitions>
68
```

Fig. 26.11 | TV GUI showing the versatility of WPF customization. (Part 2 of 5.)

```
69          <!-- define the screen -->
70          <Border Margin="0,20,0,10" Background="Black"
71             HorizontalAlignment="Center" VerticalAlignment="Center"
72             BorderThickness="2" BorderBrush="Silver" CornerRadius="2">
73             <MediaElement Height="300" Width="400"
74                Name="videoMediaElement" Source="Video/future_nasa.wmv"
75                LoadedBehavior="Manual" Stretch="Fill" />
76          </Border>
77
78          <!-- define the play, pause, and stop buttons -->
79          <StackPanel Grid.Row="1" HorizontalAlignment="Right"
80             Orientation="Horizontal">
81             <RadioButton Name="playRadioButton" IsEnabled="False"
82                Margin="0,0,5,15"
83                Template="{StaticResource RadioButtonTemplate}"
84                Checked="playRadioButton_Checked">
85                <Image Height="20" Width="20"
86                   Source="Images/play.png" Stretch="Uniform" />
87             </RadioButton>
88             <RadioButton Name="pauseRadioButton" IsEnabled="False"
89                Margin="0,0,5,15"
90                Template="{StaticResource RadioButtonTemplate}"
91                Checked="pauseRadioButton_Checked">
92                <Image Height="20" Width="20"
93                   Source="Images/pause.png" Stretch="Uniform" />
94             </RadioButton>
95             <RadioButton Name="stopRadioButton" IsEnabled="False"
96                Margin="0,0,15,15"
97                Template="{StaticResource RadioButtonTemplate}"
98                Checked="stopRadioButton_Checked">
99                <Image Height="20" Width="20"
100                  Source="Images/stop.png" Stretch="Uniform" />
101            </RadioButton>
102         </StackPanel>
103
104         <!-- define the power button -->
105         <CheckBox Name="powerCheckBox" Grid.Row="1" Width="25"
106            Height="25" HorizontalAlignment="Left"
107            Margin="15,0,0,15" Checked="powerCheckBox_Checked"
108            Unchecked="powerCheckBox_Unchecked">
109            <CheckBox.Template> <!-- set the template -->
110               <ControlTemplate TargetType="CheckBox">
111                  <Grid>
112                     <!-- create a circular border -->
113                     <Ellipse Width="25" Height="25"
114                        Fill="Silver" />
115
116                     <!-- create an "illuminated" background -->
117                     <Ellipse Name="backgroundEllipse" Width="22"
118                        Height="22">
119                        <Ellipse.Fill> <!-- unchecked state -->
```

Fig. 26.11 | TV GUI showing the versatility of WPF customization. (Part 3 of 5.)

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

                          <!-- 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>
                    </ControlTemplate>
                 </CheckBox.Template>
              </CheckBox>
           </Grid>

           <!-- 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>
        </Border>

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

Fig. 26.11 | TV GUI showing the versatility of WPF customization. (Part 4 of 5.)

```
                <!-- create a reflection of the video -->
                <VisualBrush
                   Visual="{Binding ElementName=videoMediaElement}">
                   <VisualBrush.RelativeTransform>
                      <ScaleTransform ScaleY="-1" CenterY="0.5" />
                   </VisualBrush.RelativeTransform>
                </VisualBrush>
             </Rectangle.Fill>

             <!-- make reflection more transparent the further it gets
                from the screen -->
             <Rectangle.OpacityMask>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                   <GradientStop Color="Black" Offset="-0.25" />
                   <GradientStop Color="Transparent" Offset="0.5" />
                </LinearGradientBrush>
             </Rectangle.OpacityMask>
          </Rectangle>

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

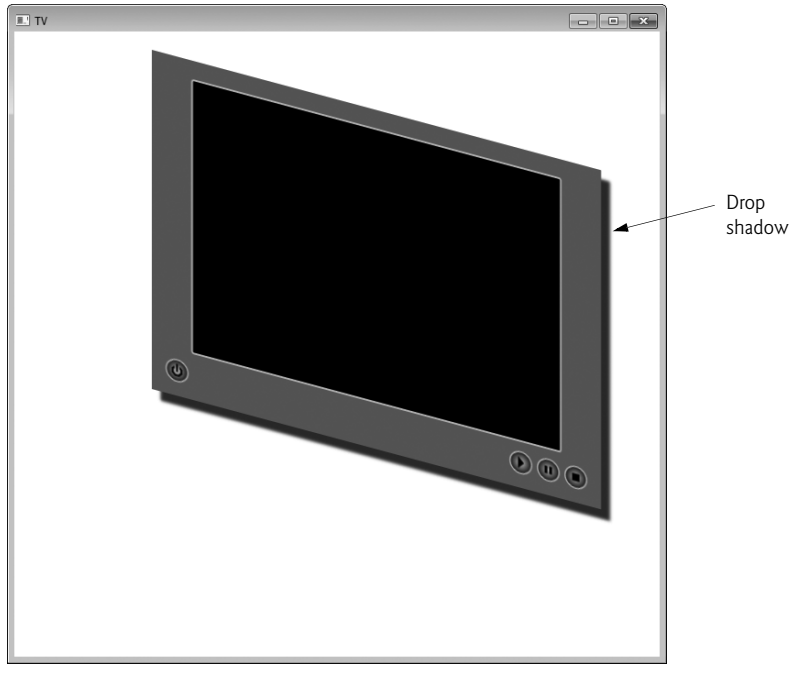


Fig. 26.11 | TV GUI showing the versatility of WPF customization. (Part 5 of 5.)

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. 26.11, 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`s, and the power button is a `CheckBox`. Lines 9–55 and 110–152 define the `ControlTemplate`s used to render the `RadioButton`s 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 `Ellipse`s (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-

tom-left corner, and the playback buttons are bound in a `StackPanel` in the bottom-right corner (lines 79–154).

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. 26.12. 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 26.13 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

Fig. 26.12 | The effect of skewing the TV app's GUI components.

```
1  // Fig. 26.13: MainWindow.xaml.cs
2  // TV GUI showing the versatility of WPF customization (code-behind).
3  using System.Windows;
4
5  namespace TV
6  {
7     public partial class MainWindow : Window
8     {
9        // constructor
10       public MainWindow()
11       {
12          InitializeComponent();
13       } // end constructor
14
15       // turns "on" the TV
16       private void powerCheckBox_Checked( object sender,
17          RoutedEventArgs e )
18       {
19          // render the reflection visible
20          reflectionRectangle.Visibility = Visibility.Visible;
21
22          // enable play, pause, and stop buttons
23          playRadioButton.IsEnabled = true;
24          pauseRadioButton.IsEnabled = true;
```

Fig. 26.13 | TV GUI showing the versatility of WPF customization (code-behind). (Part 1 of 2.)

```
25            stopRadioButton.IsEnabled = true;
26         } // end method powerCheckBox_Checked
27
28         // turns "off" the TV
29         private void powerCheckBox_Unchecked( object sender,
30            RoutedEventArgs e )
31         {
32            // shut down the screen
33            videoMediaElement.Close();
34
35            // hide the reflection
36            reflectionRectangle.Visibility = Visibility.Hidden;
37
38            // disable the play, pause, and stop buttons
39            playRadioButton.IsChecked = false;
40            pauseRadioButton.IsChecked = false;
41            stopRadioButton.IsChecked = false;
42            playRadioButton.IsEnabled = false;
43            pauseRadioButton.IsEnabled = false;
44            stopRadioButton.IsEnabled = false;
45         } // end method powerCheckBox_Unchecked
46
47         // plays the video
48         private void playRadioButton_Checked( object sender,
49            RoutedEventArgs e )
50         {
51            videoMediaElement.Play();
52         } // end method playRadioButton_Checked
53
54         // pauses the video
55         private void pauseRadioButton_Checked( object sender,
56            RoutedEventArgs e )
57         {
58            videoMediaElement.Pause();
59         } // end method pauseRadioButton_Checked
60
61         // stops the video
62         private void stopRadioButton_Checked( object sender,
63            RoutedEventArgs e )
64         {
65            videoMediaElement.Stop();
66         } // end method stopRadioButton_Checked
67      } // end class MainWindow
68   } // end namespace TV
```

Fig. 26.13 | TV GUI showing the versatility of WPF customization (code-behind). (Part 2 of 2.)

these tasks are built into the `MediaElement` control. Playback can be modified *programmatically* only if the `LoadedBehavior` is `Manual` (line 75 in Fig. 26.11).

26.8 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. 26.14) 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.

```
1  <!-- Fig. 26.14: MainWindow.xaml -->
2  <!-- Animating graphic elements with Storyboards. -->
3  <Window x:Class="UsingAnimations.MainWindow"
4     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6     Title="UsingAnimations" Height="400" Width="500">
7     <Grid>
8        <Grid.ColumnDefinitions>
9           <ColumnDefinition />
10          <ColumnDefinition Width="Auto" />
11       </Grid.ColumnDefinitions>
12
13       <MediaElement Name="video" Height="100" Width="100"
14          Stretch="UniformToFill" Source="media.mp4" />
15
16       <StackPanel Grid.Column="1">
17          <!-- TextBox will contain the new Width for the video -->
18          <TextBlock TextWrapping="Wrap" Text="Width:" Margin="5,0,0,0"/>
19          <TextBox Name="widthValue" Width="75" Margin="5">100</TextBox>
20
21          <!-- TextBox will contain the new Height for the video -->
22          <TextBlock TextWrapping="Wrap" Text="Heigth:" Margin="5,0,0,0"/>
23          <TextBox Name="heightValue" Width="75" Margin="5">100</TextBox>
24
25          <!-- When clicked, rectangle animates to the input values -->
26          <Button Content="Animate" Width="75" Margin="5">
27             <Button.Triggers> <!-- Use trigger to call animation -->
28                <!-- When button is clicked -->
29                <EventTrigger RoutedEvent="Button.Click">
30                   <BeginStoryboard> <!-- Begin animation -->
31                      <Storyboard Storyboard.TargetName="video">
32                         <!-- Animates the Width -->
33                         <DoubleAnimation Duration="0:0:2"
34                            Storyboard.TargetProperty="Width"
35                            To="{Binding ElementName=widthValue,
36                            Path=Text}" />
37
38                         <!-- Animates the Height -->
39                         <DoubleAnimation Duration="0:0:2"
40                            Storyboard.TargetProperty="Height"
41                            To="{Binding ElementName=heightValue,
42                            Path=Text}" />
```

Fig. 26.14 | Animating graphic elements with `Storyboards`. (Part 1 of 2.)

```
43                          </Storyboard>

44                       </BeginStoryboard>
45                    </EventTrigger>
46                 </Button.Triggers>
47              </Button>
48           </StackPanel>
49        </Grid>
50     </Window>
```

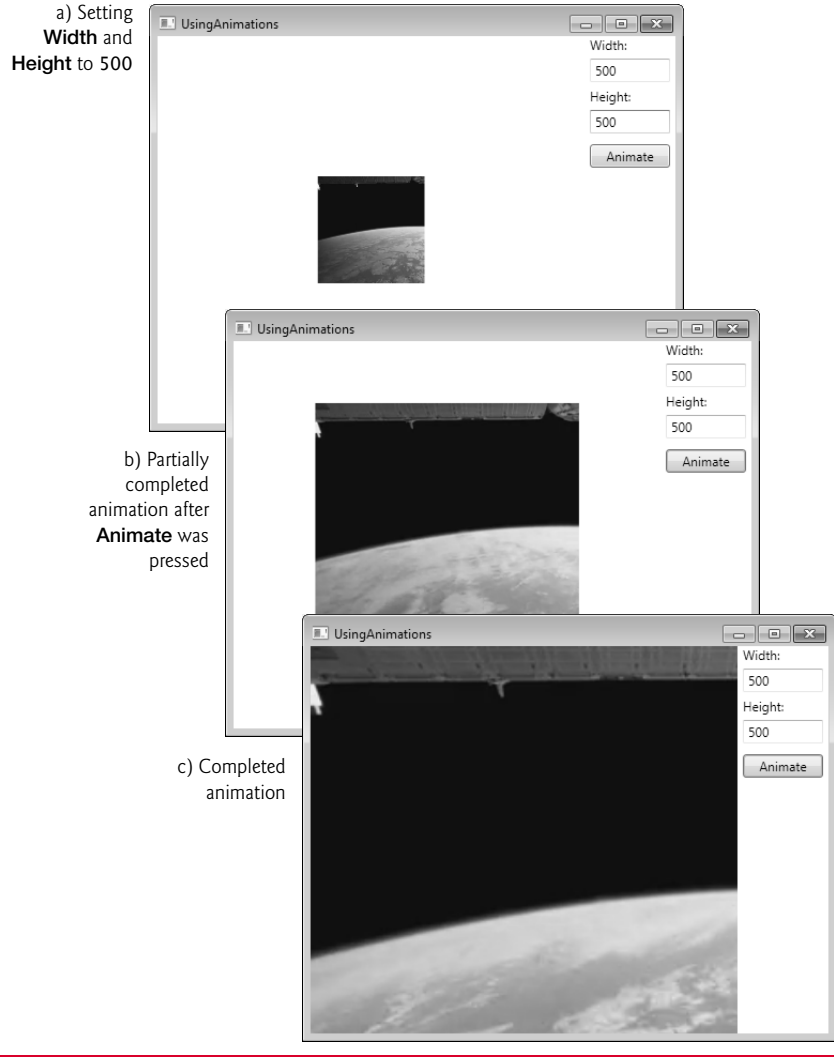


Fig. 26.14 | Animating graphic elements with `Storyboard`s. (Part 2 of 2.)

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

`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.

26.9 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. 26.15–26.16 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. 26.15) consists of a vertical `StackPanel` containing a `TextBox`, a `Button` and a series of horizontal `StackPanels` containing `Images` and `TextBlocks` that label those `Images`.

```
<!-- Fig. 26.15: MainWindow.xaml -->
<!-- Text-To-Speech and Speech-To-Text -->
<Window x:Class="SpeechApp.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Speech App" Height="580" Width="350">
   <Grid>
      <StackPanel Orientation="Vertical">
         <TextBox x:Name="SpeechBox" Text="Enter text to speak here"/>
         <Button x:Name="SpeechButton"
            Content="Click to hear the text above."
            Click="SpeechButton_Click" />
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="center">
            <Image Source="images/CPE_100h.gif" Name="ErrorImage"
               MouseDown="Image_MouseDown" />
            <Image Source="images/EPT_100h.gif" Name="PreventionImage"
               MouseDown="Image_MouseDown" />
            <Image Source="images/GPP_100h.gif"
               Name="GoodPracticesImage" MouseDown="Image_MouseDown" />
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center">
            <TextBlock Width="110" Text="Common Programming Errors"
               TextWrapping="wrap" TextAlignment="Center"/>
            <TextBlock Width="110" Text="Error-Prevention Tips"
               TextWrapping="wrap" TextAlignment="Center" />
            <TextBlock Width="110" Text="Good Programming Practices"
               TextWrapping="wrap" TextAlignment="Center"/>
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="center">
            <Image Source="images/GUI_100h.gif"
               Name="LookAndFeelImage" MouseDown="Image_MouseDown" />
            <Image Source="images/PERF_100h.gif"
               Name="PerformanceImage" MouseDown="Image_MouseDown" />
            <Image Source="images/PORT_100h.gif"
               Name="PortabilityImage" MouseDown="Image_MouseDown" />
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center">
            <TextBlock Width="110" Text="Look-and-Feel Observations"
               TextWrapping="wrap" TextAlignment="Center"/>
            <TextBlock Width="110" Text="Performance Tips"
               TextWrapping="wrap" TextAlignment="Center" />
            <TextBlock Width="110" Text="Portability Tips"
               TextWrapping="wrap" TextAlignment="Center"/>
         </StackPanel>
         <Image Source="images/SEO_100h.gif" Height="100" Width="110"
            Name="ObservationsImage" MouseDown="Image_MouseDown" />
         <TextBlock Width="110" Text="Software Engineering
            Observations" TextWrapping="wrap" TextAlignment="Center" />
```

Fig. 26.15 | Text-To-Speech and Speech-To-Text. (Part 1 of 2.)

```
            <TextBlock x:Name="InfoBlock" Margin="5"
               Text="Click an icon or say its name to view details."
               TextWrapping="Wrap"/>
         </StackPanel>
      </Grid>
   </Window>
```

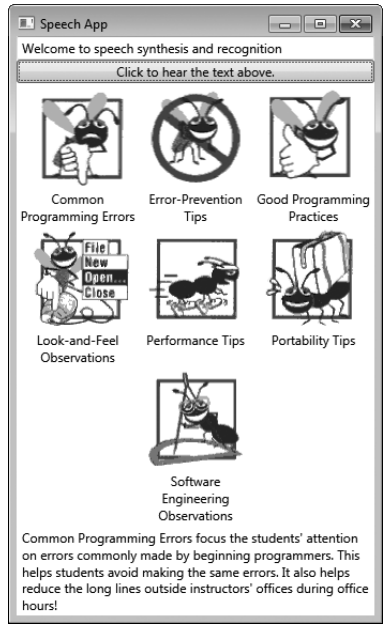


Fig. 26.15 | Text-To-Speech and Speech-To-Text. (Part 2 of 2.)

Figure 26.16 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` to the project 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**.

You must also import the **`System.Speech.Synthesis`** and **`System.Speech.Recognition`** namespaces (lines 5–6).

```
1  // Fig. 33.16: MainWindow.xaml.cs
2  // Text-To-Speech and Speech-To-Text
3  using System;
```

Fig. 26.16 | Text-To-Speech and Speech-To-Text code-behind file. (Part 1 of 4.)

```
4  using System.Collections.Generic;
5  using System.Speech.Synthesis;
6  using System.Speech.Recognition;
7  using System.Windows;
8  using System.Windows.Controls;
9
10 namespace SpeechApp
11 {
12    public partial class MainWindow : Window
13    {
14       // listens for speech input
15       private SpeechRecognizer listener = new SpeechRecognizer();
16
17       // gives the listener choices of possible input
18       private Grammar myGrammar;
19
20       // sends speech output to the speakers
21       private SpeechSynthesizer talker = new SpeechSynthesizer();
22
23       // keeps track of which description is to be printed and spoken
24       private string displayString;
25
26       // maps images to their descriptions
27       private Dictionary< Image, string > imageDescriptions =
28          new Dictionary< Image, string >();
29
30       // maps input phrases to their descriptions
31       private Dictionary< string, string > phraseDescriptions =
32          new Dictionary< string, string >();
33
34       public MainWindow()
35       {
36          InitializeComponent();
37
38          // define the input phrases
39          string[] phrases = { "Good Programming Practices",
40             "Software Engineering Observations", "Performance Tips",
41             "Portability Tips", "Look-And-Feel Observations",
42             "Error-Prevention Tips", "Common Programming Errors" };
43
44          // add the phrases to a Choices collection
45          Choices theChoices = new Choices( phrases );
46
47          // build a Grammar around the Choices and set up the
48          // listener to use this grammar
49          myGrammar = new Grammar( new GrammarBuilder( theChoices ) );
50          listener.Enabled = true;
51          listener.LoadGrammar( myGrammar );
52          myGrammar.SpeechRecognized += myGrammar_SpeechRecognized;
53
```

Fig. 26.16 | Text-To-Speech and Speech-To-Text code-behind file. (Part 2 of 4.)

```
54          // define the descriptions for each icon/phrase
55          string[] descriptions = {
56             "Good Programming Practices highlight " +
57                "techniques for writing programs that are clearer, more " +
58                "understandable, more debuggable, and more maintainable.",
59             "Software Engineering Observations highlight " +
60                "architectural and design issues that affect the " +
61                "construction of complex software systems.",
62             "Performance Tips highlight opportunities " +
63                "for improving program performance.",
64             "Portability Tips help students write " +
65                "portable code that can execute on different platforms.",
66             "Look-and-Feel Observations highlight " +
67                "graphical user interface conventions. These " +
68                "observations help students design their own graphical " +
69                "user interfaces in conformance with industry standards.",
70             "Error-Prevention Tips tell people how to " +
71                "test and debug their programs. Many of the tips also " +
72                "describe aspects of creating programs that " +
73                "reduce the likelihood of 'bugs' and thus simplify the " +
74                "testing and debugging process.",
75             "Common Programming Errors focus the " +
76                "students' attention on errors commonly made by " +
77                "beginning programmers. This helps students avoid " +
78                "making the same errors. It also helps reduce the long " +
79                "lines outside instructors' offices during " +
80                "office hours!" };
81
82          // map each image to its corresponding description
83          imageDescriptions.Add( GoodPracticesImage, descriptions[ 0 ] );
84          imageDescriptions.Add( ObservationsImage, descriptions[ 1 ] );
85          imageDescriptions.Add( PerformanceImage, descriptions[ 2 ] );
86          imageDescriptions.Add( PortabilityImage, descriptions[ 3 ] );
87          imageDescriptions.Add( LookAndFeelImage, descriptions[ 4 ] );
88          imageDescriptions.Add( PreventionImage, descriptions[ 5 ] );
89          imageDescriptions.Add( ErrorImage, descriptions[ 6 ] );
90
91          // loop through the phrases and descriptions and map accordingly
92          for ( int index = 0; index <= 6; ++index )
93             phraseDescriptions.Add( phrases[ index ],
94                descriptions[ index ] );
95
96          talker.Rate = -4; // slows down the speaking rate
97       } // end constructor
98
99       // when the user clicks on the speech-synthesis button, speak the
100      // contents of the related text box
101      private void SpeechButton_Click( object sender, RoutedEventArgs e )
102      {
103         talker.SpeakAsync( SpeechBox.Text );
104      } // end method SpeechButton_Click
105
```

Fig. 26.16 | Text-To-Speech and Speech-To-Text code-behind file. (Part 3 of 4.)

```
106         private void Image_MouseDown( object sender,
107            System.Windows.Input.MouseButtonEventArgs e )
108         {
109            // use the image-to-description dictionary to get the
110            // appropriate description for the clicked image
111            displayString = imageDescriptions[ (Image) sender ];
112            DisplaySpeak();
113         } // end method Image_MouseDown
114
115         // when the listener recognizes a phrase from the grammar, set the
116         // display string and call DisplaySpeak
117         void myGrammar_SpeechRecognized(
118            object sender, RecognitionEventArgs e )
119         {
120            // Use the phrase-to-description dictionary to get the
121            // appropriate description for the spoken phrase
122            displayString = phraseDescriptions[ e.Result.Text ];
123
124            // Use the dispatcher to call DisplaySpeak
125            this.Dispatcher.BeginInvoke(
126               new Action( DisplaySpeak ) );
127         } // end method myGrammar_SpeechRecognized
128
129         // Set the appropriate text block to the display string
130         // and order the synthesizer to speak it
131         void DisplaySpeak()
132         {
133            InfoBlock.Text = displayString;
134            talker.SpeakAsync( displayString );
135         } // end method DisplaySpeak
136      } // end class MainWindow
137   } // end namespace SpeechApp
```

Fig. 26.16 | Text-To-Speech and Speech-To-Text code-behind file. (Part 4 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` `imageDescriptions` contains pairs of `Image`s and `string`s, and the `Dictionary` `phraseDescriptions` contains pairs of `string`s and `string`s. These `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

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`. `SpeechSynthesizer`s also have a `Speak` method, which is not asynchronous, and `SpeakSsml` and `SpeakSsmlAsynch`, 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`.

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.

26.10 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.

Summary

Section 26.1 Introduction

- WPF integrates drawing and animation features that were previously available only in special libraries.
- WPF graphics use resolution-independent units of measurement.
- A machine-independent pixel measures 1/96 of an inch.
- WPF graphics use a vector-based system in which calculations determine how to size and scale each element.

Section 26.2 Controlling Fonts

- A TextBlock is a control that displays text.
- The FontFamily property of TextBlock defines the font of the displayed text.
- The FontSize property of TextBlock defines the text size measured in points.
- The FontWeight property of TextBlock defines the thickness of the text and can be assigned to either a numeric value or a predefined descriptive value.
- The FontStyle property of TextBlock can be used to make the text Italic or Oblique.
- You also can define the TextDecorations property of a TextBlock to give the text any of four TextDecorations: Underline, Baseline, Strikethrough and Overline.

Section 26.3 Basic Shapes

- Shape controls have `Height` and `Width` properties as well as `Fill`, `Stroke` and `StrokeThickness` properties to define the appearance of the shape.
- The `Line`, `Rectangle` and `Ellipse` are three basic shapes available in WPF.
- A `Line` is defined by its two endpoints.
- The `Rectangle` and the `Ellipse` are defined by upper-left coordinates, width and height.
- If a `Stroke` or `Fill` of a shape is not specified, that property will be rendered transparently.

Section 26.4 Polygons and Polylines

- A `Polyline` draws a series of connected lines defined by a set of points.
- A `Polygon` draws a series of connected lines defined by a collection of points and connects the first and last points to create a closed figure.
- The `Polyline` and `Polygon` shapes connect the points based on the ordering in the collection.
- The `Visibility` of a graphic control can be set to `Visible`, `Collapsed` or `Hidden`.
- 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.
- A `PointCollection` is a collection that stores `Point` objects.
- `PointCollection`'s `Add` method adds another point to the end of the collection.
- `PointCollection`'s `Clear` method empties the collection.

Section 26.5 Brushes

- Brushes can be used to change the graphic properties of an element, such as the `Fill`, `Stroke` or `Background`.
- An `ImageBrush` paints an image into the property it's assigned to (such as a `Background`).
- A `VisualBrush` can display a fully customized video into the property it's assigned to.
- To use audio and video in a WPF app, you use the `MediaElement` control.
- `LinearGradientBrush` transitions linearly through the colors specified by `GradientStops`.
- `RadialGradientBrush` transitions through the specified colors radially outward from a specified point.
- Logical points are used to reference locations in the control independent of the actual size. The point (0,0) represents the top-left corner and the point (1,1) represents the bottom-right corner.
- A `GradientStop` defines a single color along the gradient.
- The `Offset` property of a `GradientStop` defines where the color appears along the transition.

Section 26.6 Transforms

- A `TranslateTransform` moves the object based on given *x*- and *y*-offset values.
- A `RotateTransform` rotates the object around a `Point` and by a specified `RotationAngle`.
- A `SkewTransform` skews (or shears) the object, meaning it rotates the *x*- or *y*-axis based on specified `AngleX` and `AngleY` values. A `SkewTransform` creates an oblique distortion of an element.
- A `ScaleTransform` scales the image's *x*- and *y*-coordinate points by different specified amounts.
- The `Random` class's `NextBytes` method assigns a random value in the range 0–255 to each element in its `Byte` array argument.
- The `RenderTransform` property of a GUI element contains embedded transforms that are applied to the control.

Section 26.7 WPF Customization: A Television GUI

- WPF bitmap effects can be used to apply simple visual effects to GUI elements. They can be applied by setting an element's `Effect` property
- By setting the `ScaleX` or `ScaleY` property of a `ScaleTransform` to a negative number, you can invert an element horizontally or vertically, respectively.
- An opacity mask translates a partially transparent brush into a mapping of opacity values and applies it to an object. You define an opacity mask by specifying a brush as the `OpacityMask` property.
- An orthographic projection depicts a 3-D space graphically, and does not account for depth. A perspective projection presents a realistic representation of a 3-D space.
- The `MediaElement` control has built-in playback methods. These methods can be called only if the `LoadedBehavior` property of the `MediaElement` is set to `Manual`.

Section 26.8 Animations

- A `Storyboard` contains embedded animation elements. When the `Storyboard` begins executing, all embedded animations execute.
- The `TargetProperty` of a `Storyboard` specifies which property of the control you want to change.
- A `DoubleAnimation` animates properties of type `Double`. `PointAnimations` and `ColorAnimations` are two other commonly used animation controls.
- The `Stretch` property of images and videos determines how the media stretches to fit the size of its enclosure. This property can be set to `None`, `Uniform`, `UniformToFill` or `Fill`.
- `Stretch` property value `None` uses the media's native size.
- `Uniform` value for the `Stretch` property resizes the media to its largest possible size while still being confined inside the container with its native aspect ratio.
- `UniformToFill` value for the `Stretch` property resizes the media to completely fill the container while still keeping its aspect ratio—as a result, it could be cropped.
- `Fill` value for the `Stretch` property forces the media to be resized to the size of the container (aspect ratio is not preserved).

Section 26.9 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.
- To use .NET's speech synthesis and recognition classes, you must add a reference to `System.Speech` to the project. You must also import the `System.Speech.Synthesis` and `System.Speech.Recognition` namespaces.
- A `SpeechRecognizer` 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.
- A `SpeechSynthesizer` object speaks text, using one of several voices.
- A `Dictionary` is a collection of key/value pairs, in which each key has a corresponding value.
- A `SpeechSynthesizer`'s `SpeakAsync` method speaks the specified text.
- The `SpeechRecognized` event occurs whenever a `SpeechRecognizer` detects that one of the input phrases defined in its `Grammar` was spoken.

Terminology

`Add` method of class `PointCollection`
aspect ratio
`Background` property of `TextBlock` control
`BlurEffect`
`CenterX` property of `ScaleTransform`
`CenterY` property of `ScaleTransform`
`Clear` method of class `PointCollection`
`Color` property of `GradientStop`
cropping
`DoubleAnimation`
`DropShadowEffect`
`Duration` property of `DoubleAnimation`
`Ellipse`
`EndPoint` property of `LinearGradientBrush`
`Fill` property of a shape
`Fill` value of `Stretch` property
`FontFamily` property of `TextBlock` control
`FontSize` property of `TextBlock` control
`FontStyle` property of `TextBlock` control
`FontWeight` property of `TextBlock` control
`Foreground` property of `TextBlock` control
`From` property of `DoubleAnimation`
gradient
`GradientStop`
`Grammar`
`ImageBrush` (WPF)
`ImageSource` property of `ImageBrush`
`Line`
`LinearGradientBrush`
`Location` property of `TextDecoration`
logical point
machine-independent pixel
`MediaElement` control
`NextBytes` method of class `Random`
`None` value for `Stretch` property
`Offset` property of `GradientStop`
opacity mask
`OpacityMask` property of `Rectangle`
orthographic projection
perspective projection
`PointCollection` class
`Points` property of `Polyline`
`Polygon`
`Polyline`
`RadialGradientBrush`
`Rectangle`
`RenderTransform` property of a WPF UI element
`RepeatBehavior` property of `Storyboard`
`RotateTransform`
`ScaleTransform`
`ScaleX` property of `ScaleTransform`
`ScaleY` property of `ScaleTransform`
`SkewTransform`
`Source` property of `MediaElement` control
Speech recognizers
Speech synthesizers
`SpeechRecognizer`
`SpeechSynthesizer`
speech-to-text systems
`StartPoint` property of `LinearGradientBrush`
`Storyboard`
`Stretch` property of `MediaElement` control
`Stroke` property of a shape
`StrokeThickness` property of a shape
`System.Speech.Recognition`
`System.Speech.Synthesis`
`TargetName` property of `Storyboard`
`TargetProperty` property of `Storyboard`
`TextBlock` control
`TextDecoration`
`TextDecorations` property of `TextBlock` control
text-to-speech systems
`To` property of `DoubleAnimation`
transform
`TranslateTransform`
`Uniform` value for `Stretch` property
`UniformToFill` value for `Stretch` property
vector-based graphics
`Visibility` property of a WPF UI element
`Visual` property of `VisualBrush`
`VisualBrush`

Self-Review Exercises

26.1 State whether each of the following is *true* or *false*. If *false*, explain why.
a) The unit of measurement for the `FontSize` property is machine independent.
b) A `Line` is defined by its length and its direction.
c) If an object's `Fill` is not defined, it uses the default `White` color.

d) A `Polyline` and `Polygon` are the same, except that the `Polygon` connects the first point in the `PointCollection` with the last point.
e) A `Collapsed` element occupies space in the window, but it's transparent.
f) A `MediaElement` is used for audio or video playback.
g) A `LinearGradientBrush` always defines a gradient that transitions through colors from left to right.
h) A transform can be applied to a WPF UI element to reposition or reorient the graphic.
i) A `Storyboard` is the main control for implementing animations into the app.

26.2 Fill in the blanks in each of the following statements:
a) A(n) _____ control can be used to display text in the window.
b) A(n) _____ can apply `Underlines`, `Overlines`, `Baselines` or `Strikethroughs` to a piece of text.
c) The _____ property of the `DoubleAnimation` defines the final value taken by the animated property.
d) Four types of transforms are _____, _____, `TranslateTransform` and `RotateTransform`.
e) The _____ property of a `GradientStop` defines where along the transition the corresponding color appears.
f) The _____ property of a `Storyboard` defines what property you want to animate.
g) A `Polygon` connects the set of points defined in a(n) _____ object.
h) The three basic available shape controls are `Line`, _____ and _____.
i) A(n) _____ creates an opacity mapping from a brush and applies it to an element.
j) _____ points are used to define the `StartPoint` and `EndPoint` of a gradient to reference locations independently of the control's size.

Answers to Self-Review Exercises

26.1 a) True. b) False. A `Line` is defined by a start point and an end point. c) False. When no `Fill` is defined, the object is transparent. d) True. e) False. A `Collapsed` object has a `Width` and `Height` of 0. f) True. g) False. You can define start and end points for the gradient to change the direction of the transitions. h) True. i) True.

26.2 a) `TextBlock`. b) `TextDecoration`. c) `To`. d) `SkewTransform`, `ScaleTransform`. e) `Offset`. f) `TargetProperty`. g) `PointCollection`. h) `Rectangle`, `Ellipse`. i) opacity mask. j) Logical.

Exercises

26.3 *(Enhanced `UsingGradients` app)* Modify the `UsingGradients` example from Section 26.5 to allow the user to switch between having a `RadialGradient` or a `LinearGradient` in the `Rectangle`. Users can still modify either gradient with the RGBA values as before. At the bottom of the window, place `RadioButtons` that can be used to specify the gradient type. When the user switches between types of gradients, the colors should be kept consistent. In other words, if there is currently a `LinearGradient` on screen with a purple start color and a black stop color, the `RadialGradient` should have those start and stop colors as well when switched to. The GUI should appear as shown in Fig. 26.17.

26.4 *(Enhanced `DrawStars` app)* Modify the `DrawStars` example in Section 26.6 so that all the stars animate in a circular motion. Do this by animating the `Angle` property of the `Rotation` applied to each `Polygon`. The GUI should look as shown in Fig. 26.18, which is how it looked in the example in the chapter. Notice that the stars have changed positions between the two screen captures. [*Hint:* Controls have a `BeginAnimation` method which can be used to apply an animation without predefining it in a `Storyboard` element. For this exercise, the method's first argument should be `RotateTransform.AngleProperty`.]

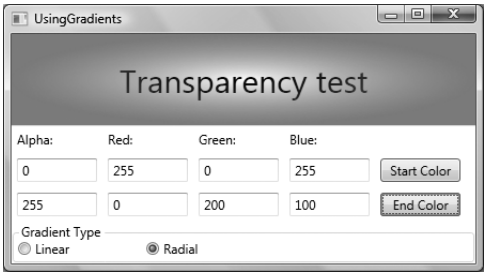


Fig. 26.17 | `UsingGradients` example after `RadioButton` enhancement.

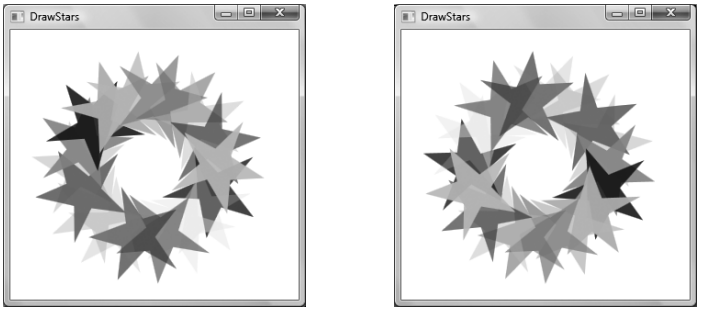


Fig. 26.18 | Animated `Polygon`s rotating along the same circle.

26.5 ***(`Image` reflector app)*** Create an app that has the same GUI as shown in Fig. 26.19(a). The cover images are included in the `ExerciseImages` folder with this chapter's examples. When the mouse hovers over any one of the covers, that cover and its reflection should animate to a larger size. Figure 26.19(b) shows one of the enlarged covers with a mouse over it.

a) Images shown with reflections

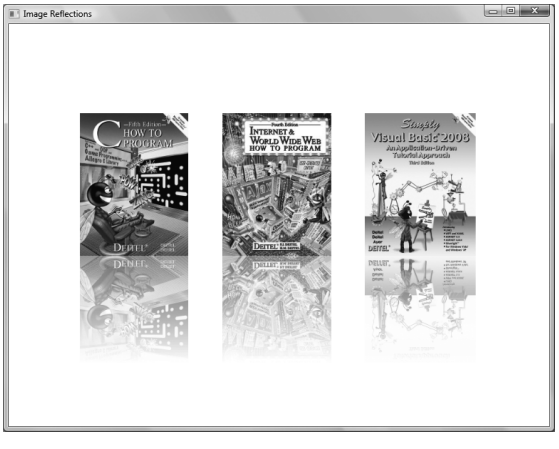


Fig. 26.19 | Cover images and their reflections. (Part 1 of 2.)

b) Image expanded when mouse hovers over it

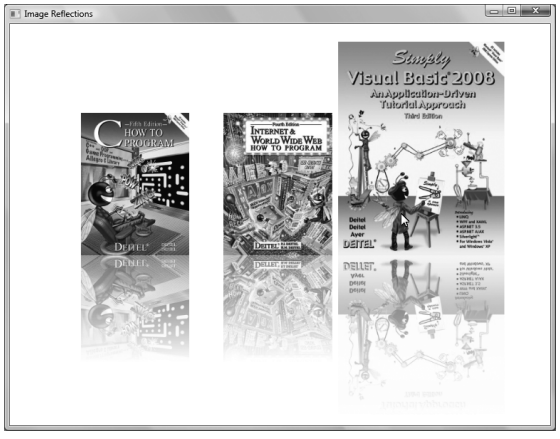


Fig. 26.19 | Cover images and their reflections. (Part 2 of 2.)

26.6 ***(Snake `PolyLine` app)*** Create an app that creates a `Polyline` object that acts like a snake following your cursor around the window. The app, once the `Polyline` is created, should appear as shown in Fig. 26.20. You need to create an `Ellipse` for the head and a `Polyline` for the body of the snake. The head should always be at the location of the mouse cursor, while the `Polyline` continuously follows the head (make sure the length of the snake does not increase forever).

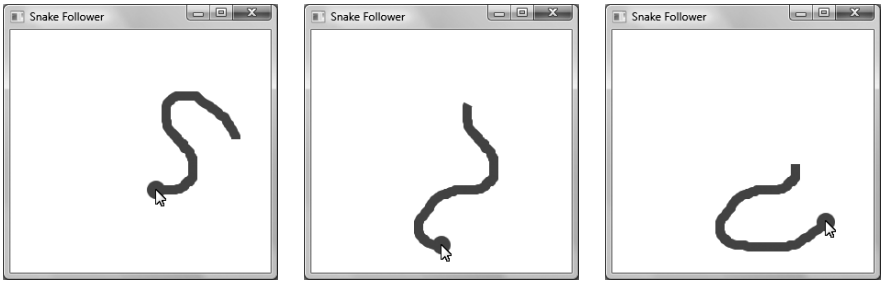


Fig. 26.20 | Snake follows the mouse cursor inside the window.

26.7 ***(Project: Speech-Controlled Drawing App)*** Create a speech-controlled drawing app that's speech controlled. Allow the user to speak the shape to draw, then prompt the user with speech to ask for the dimensions and location for that type of shape. Confirm each value the user speaks, then display the shape with the user-specified size and location. Your app should support lines, rectangles and ellipses.