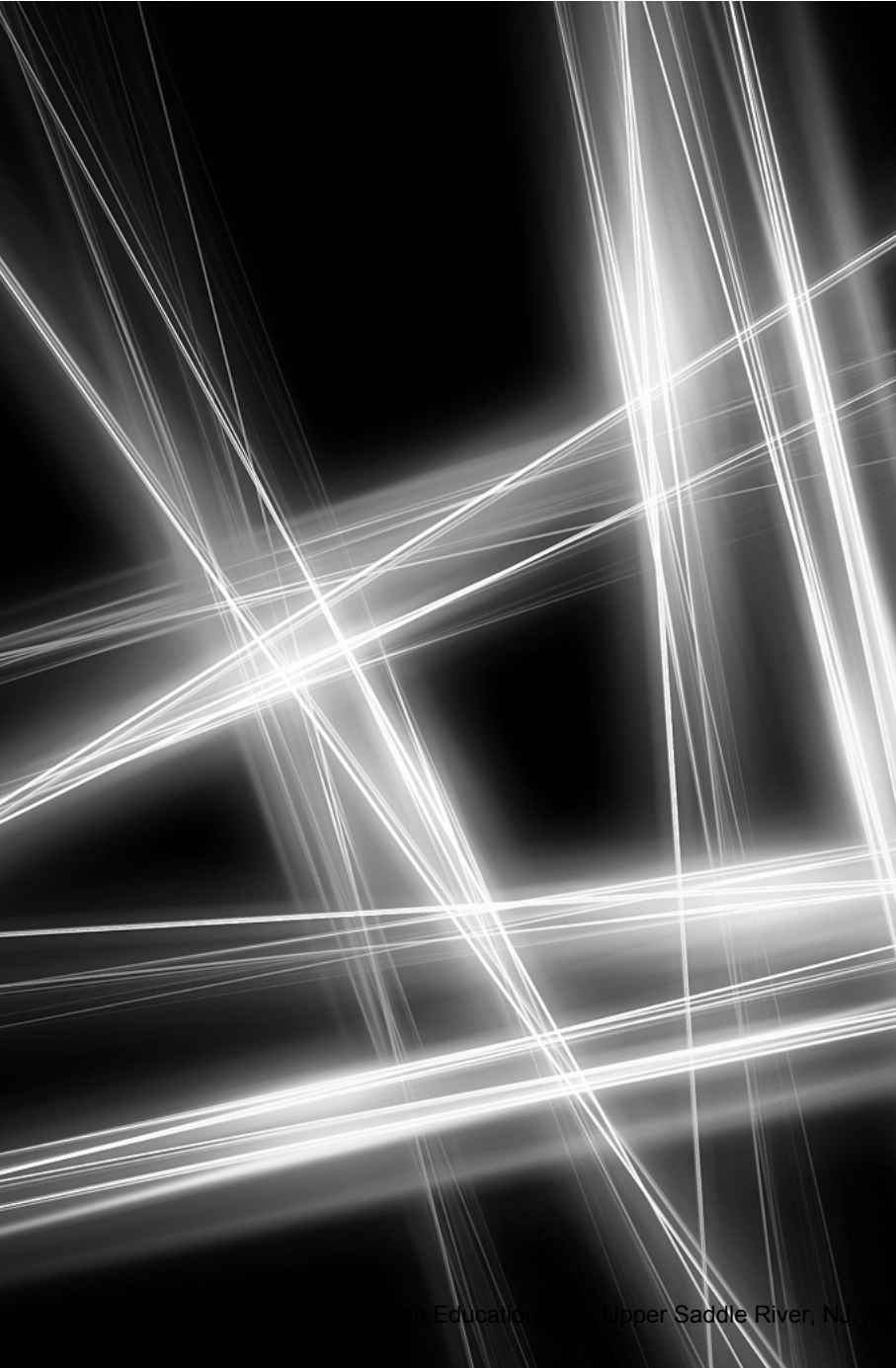


Windows 8 Graphics and Multimedia

26



Objectives

In this chapter you'll:

- Draw `Lines`, `Rectangles`, `Ellipses`, `PolyLines` and `Polygons` on `Canvas` controls.
- Use `SolidColorBrushes`, `ImageBrushes` and `LinearGradientBrushes` to customize the `Fill`, `Foreground` or `Background` of a control.
- Use transforms to reposition or reorient GUI elements.
- Use `ControlTemplates` to customize the look of a control while maintaining its functionality.
- Use `VisualStates` and animations to specify how a control's appearance changes as the control's state changes.
- Apply a projection to a control to give it a perspective.

-
- 26.1 Introduction
 - 26.2 Basic Shapes
 - 26.2.1 Rectangles
 - 26.2.2 Line
 - 26.2.3 Ellipses
 - 26.3 PolyLines and Polygons
 - 26.4 SolidColorBrushes and ImageBrushes
 - 26.5 GradientBrushes
 - 26.5.1 Setting the Rectangle's Fill to a Gradient
 - 26.5.2 GradientStops
 - 26.5.3 Defining the Gradient in the IDE
 - 26.5.4 Code-Behind File
 - 26.6 Transforms
 - 26.6.1 MainPage Instance Variables and Constructor
 - 26.6.2 MainPage Method RotateAndDrawStars
 - 26.6.3 Applying a RotateTransform
 - 26.7 Windows 8 Customization: A Television GUI
 - 26.7.1 XAML for the TV GUI
 - 26.7.2 XAML for the Power CheckBox's ControlTemplate
 - 26.7.3 Creating a Basic ControlTemplate
 - 26.7.4 Default ControlTemplate Markup
 - 26.7.5 Defining the CheckBox's New Look-And-Feel
 - 26.7.6 Using Animation to Change a Control's Look-and-Feel in Response to State Changes
 - 26.7.7 XAML for the Play, Pause and Stop RadioButtons' ControlTemplate
 - 26.7.8 XAML for the GUI
 - 26.7.9 TV GUI Code-Behind File
 - 26.8 Wrap-Up
-

Exercises

26.1 Introduction

This chapter overviews Windows 8's graphics and multimedia capabilities, including *two-dimensional shapes, transformations and video*. Windows 8 (like WPF in Chapters 32 and 33) integrates graphics and multimedia features that were previously available only in special libraries (such as DirectX). This enables you to use one set of libraries that all use the same XAML and code-behind programming model, rather than several libraries with different programming models. The graphics system in Windows 8 uses your computer's graphics hardware to reduce the load on the CPU.

Windows 8 graphics use *resolution-independent* units of measurement, making apps more uniform and *portable* across devices. The size properties of graphic elements in Windows 8 are measured in **machine-independent pixels**, where one pixel typically represents 1/96 of an inch—however, this depends on the computer's resolution settings—some users set their screens to lower resolutions so that everything appears larger on the screen and some set their screens to higher resolutions to have more screen space to work with. Windows 8 determines the correct size of each graphical element based on the screen size and resolution.

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 Windows 8's vector-based system, so they look good at many screen resolutions.

The basic 2-D shapes are Lines, Rectangles and Ellipses. Windows 8 also has controls that can be used to create custom shapes or curves. *Brushes* can be used to fill an ele-

ment with *solid colors, images and gradients*, allowing for unique and interesting visual experiences. Windows 8's robust transform capabilities allow you to further customize GUIs—*transforms* reposition and reorient controls and shapes.

26.2 Basic Shapes

Windows 8 UI has several built-in shapes (namespace `Windows.UI.Xaml.Shapes`). The `BasicShapes` example (Fig. 26.1) displays several `Rectangles`, a `Line` and several `Ellipses`. Each of the apps in this chapter uses the **Blank App** template introduced in Chapter 25. Apps with a white background use the `Light` theme (Section 25.2.3). We generally reformat XAML for clarity.

```

1  <!-- Fig. 26.1: MainPage.xaml -->
2  <!-- Specifying basic shapes in XAML. -->
3  <Page
4      x:Class="BasicShapes.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:BasicShapes"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <Canvas
13         Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
14         <!-- Rectangle with fill but no stroke -->
15         <Rectangle Canvas.Left="30" Canvas.Top="30" Width="300" Height="180"
16             Fill="Red" />
17
18         <!-- Rectangle with stroke but no fill -->
19         <Rectangle Canvas.Left="350" Canvas.Top="30" Width="300"
20             Height="180" Stroke="Black" StrokeThickness="10"/>
21
22         <!-- Rectangle with stroke and fill -->
23         <Rectangle Canvas.Left="670" Canvas.Top="30" Width="300"
24             Height="180" Stroke="Black" Fill="Red" StrokeThickness="10"/>
25
26         <!-- Line to separate rows of shapes -->
27         <Line X1="30" Y1="225" X2="970" Y2="225" Stroke="Blue"
28             StrokeThickness="10" />
29
30         <!-- Ellipse with fill and no stroke -->
31         <Ellipse Canvas.Left="30" Canvas.Top="240" Width="300" Height="180"
32             Fill="Red" />
33
34         <!-- Ellipse with stroke and no fill -->
35         <Ellipse Canvas.Left="350" Canvas.Top="240" Width="300" Height="180"
36             Stroke="Black" StrokeThickness="10"/>
37

```

Fig. 26.1 | Specifying basic shapes in XAML. (Part 1 of 2.)

```

38      <!-- Ellipse with stroke and fill -->
39      <Ellipse Canvas.Left="670" Canvas.Top="240" Width="300" Height="180"
40             Stroke="Black" Fill="Red" StrokeThickness="10"/>
41  </Canvas>
42 </Page>

```

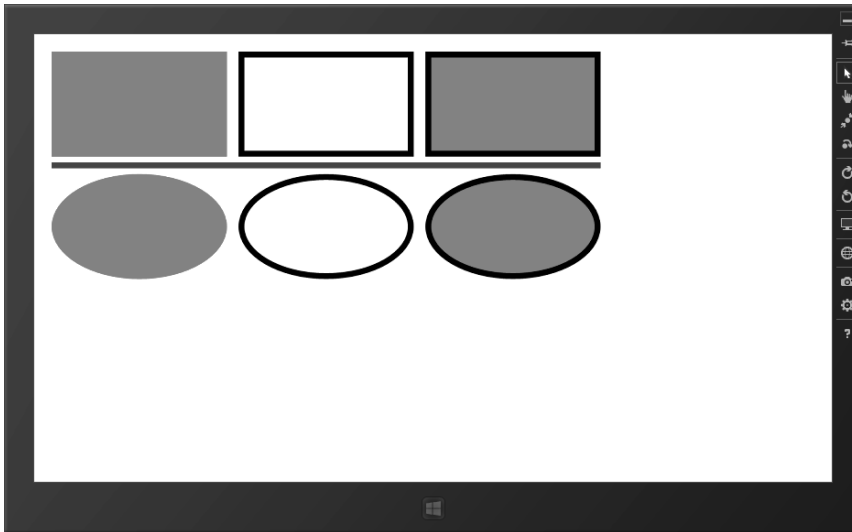


Fig. 26.1 | Specifying basic shapes in XAML. (Part 2 of 2.)

Using the Toolbox vs. Editing the XAML

By default, the **Toolbox** contains only the `Rectangle` and `Ellipse` shapes, which you can drag onto the GUI designer, as you did with Windows 8 UI controls in Chapter 25. There are various other shapes—and other less-commonly used controls—that you can add to the **Toolbox**. To do so:

1. Right click in the **General** section at the bottom of the **Toolbox**, then select **Choose Items...** to display the **Choose Toolbox Items** dialog. In the **Windows XAML Components** tab, the items that are *checked* are already displayed in the **Toolbox**.
2. To add any other shape or control, *check* its checkbox, then click **OK**. The new item will appear in the **Toolbox**'s **General** section.

If you'd like the items you add to appear in alphabetical order within the **Toolbox**, you can rearrange the **Toolbox** items by dragging them where you'd like them to appear in the **Toolbox**. You can also edit the XAML directly to create new shapes. Like any other control, you can select a shape in the designer or click its element in the XAML to edit the shape's properties in the **Properties** window.

Changing the Layout from Grid to Canvas

For this app, we used the IDE's **Document Outline** window to change the `Page`'s layout from a `Grid` (the default) to a `Canvas`. Recall from Section 25.3.1, that a `Canvas` allows

you to specify the exact position of a shape. The **Document Outline** window shows the *nested structure* of a page's layouts and controls and makes it easier to select specific controls to customize them with the **Properties** window. To change the layout:

1. Open the **Document Outline** window by selecting **VIEW > Other Windows > Document Outline**.
2. In the **Document Outline** window, right click the [Grid] node, which represents the Grid layout, and select **Change Layout Type > Canvas** from the popup menu.

26.2.1 Rectangles

The first shape (Fig. 26.1, lines 15–16) uses a **Rectangle** object to create a *filled* rectangle in the window. To specify the Rectangle's location, we set the attached properties `Canvas.Left` and `Canvas.Top` to 30. These properties appear as **Left** and **Top** in the **Properties** window's **Layout** section when the Rectangle is selected. We set the **Width** and **Height** properties—**Width** and **Height** in the **Properties** window's **Layout** section—to 300 and 180, respectively, to specify the size. To define the color, we set the **Fill** property to Red using **Fill** in the **Properties** window's **Brush** section. You can assign any solid **Color** or **Brush** (Section 26.4) to this property. Rectangles also have a **Stroke** property, which defines the color of the *shape's outline* (lines 20 and 24). If either the **Fill** or the **Stroke** is *not* specified, that property will be rendered *transparently*. For this reason, the first Rectangle in the window has no outline, while the second 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 one pixel—we used 10 for all Strokes in this app.

26.2.2 Line

A **Line** (lines 27–28) is defined by its two *endpoints*—`X1`, `Y1` and `X2`, `Y2`. Lines have a **Stroke** property that defines the Line's *color*. In this example, the Line's **Stroke** is set to Blue and its **StrokeThickness** to 10.

26.2.3 Ellipses

To draw a *circle* or *ellipse*, you can use the **Ellipse** control. An **Ellipse**'s location and size is defined like a **Rectangle**—with attached properties `Canvas.Left` and `Canvas.Top` for the *upper-left* corner, and properties **Width** and **Height** for the size (line 31). Together, the `Canvas.Left`, `Canvas.Top`, **Width** and **Height** of an **Ellipse** define an invisible *bounding rectangle* in which the **Ellipse** touches the *center* of each side. To draw a circle, provide the *same* value for the **Width** and **Height**. As with Rectangles, having an *unspecified* **Fill** property for an **Ellipse** makes the shape's fill *transparent* (lines 35–36).

26.3 PolyLines and Polygons

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. 26.2) displays one **Polyline** and two **Polygons**.

```

1  <!-- Fig. 26.2: MainPage.xaml -->
2  <!-- Defining Polylines and Polygons in XAML. -->
3  <Page
4      x:Class="DrawPolygons.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:DrawPolygons"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <Canvas
13         Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
14         <Polyline Stroke="Black" StrokeThickness="10"
15             Points="50,50 150,700 425,300 200,250"/>
16
17         <Polygon Stroke="Black" StrokeThickness="10"
18             Points="450,50 550,700 825,300 600,250"/>
19
20         <Polygon Fill="Red" Stroke="Black" StrokeThickness="10"
21             Points="850,50 950,700 1225,300 1000,250"/>
22     </Canvas>
23 </Page>

```

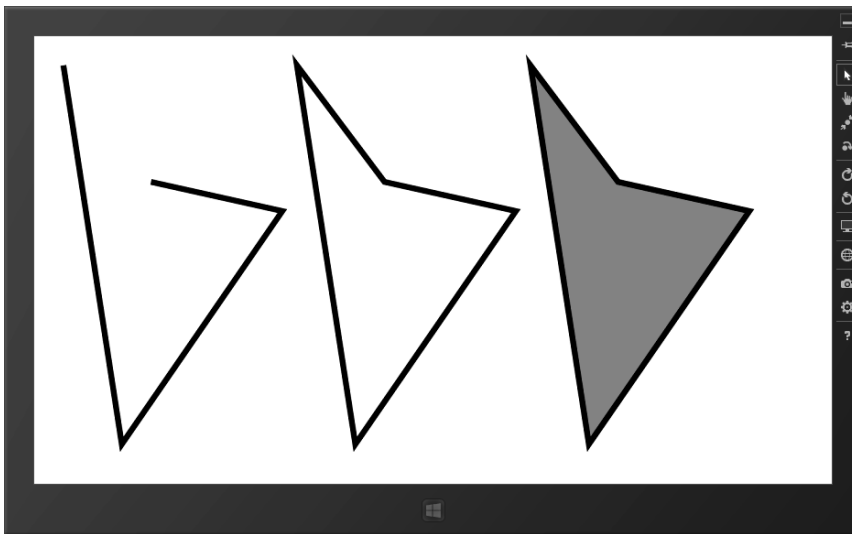


Fig. 26.2 | Defining Polyline and Polygons in XAML.

Embedded in the Canvas are a `Polyline` (lines 14–15) and two `Polygon` objects—one without a `Fill` (lines 17–18) and one with a `Fill` (lines 20–21). As you can see, `Polyline` and `Polygon` objects have `Stroke` and `StrokeThickness` properties like the simple shapes we discussed earlier. `Polygon` objects also have the `Fill` property.

Each `Polyline` and `Polygon` object has a **Points property** that we defined directly in the XAML. This property consists of a series of points that are defined as *x-y* coordinates

on the Canvas. Each point's x and y values are separated by a comma, and each point is separated from the next with a space. The `Points` property is an object of class `PointCollection` (namespace `Windows.UI.Xaml.Media`) that stores individual `Point` objects.

26.4 SolidColorBrushes and ImageBrushes

The `UsingBrushes` app (Fig. 26.3) uses `SolidColorBrushes` and `ImageBrushes` to change an element's graphic properties, such as the `Fill`, `Stroke` or `Background`. You'll learn about `GradientBrushes` in the next section. This app applies brushes to `TextBlocks` and `Ellipses`. The image used is provided in the `ImagesVideo` folder with this chapter's examples. We added it to the project by dragging it from Windows Explorer onto the project's `Assets` folder in the `Solution Explorer` window.

```

1  <!-- Fig. 26.3: MainWindow.xaml -->
2  <!-- Applying brushes. -->
3  <Page
4      x:Class="UsingBrushes.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:UsingBrushes"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <Grid
13         Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
14         <Grid.RowDefinitions>
15             <RowDefinition Height="Auto" />
16             <RowDefinition />
17         </Grid.RowDefinitions>
18
19         <Grid.ColumnDefinitions>
20             <ColumnDefinition />
21             <ColumnDefinition />
22         </Grid.ColumnDefinitions>
23
24         <!-- TextBlock with a SolidColorBrush -->
25         <TextBlock TextWrapping="Wrap" Text="Color" FontSize="200"
26             FontWeight="ExtraBold" TextAlignment="Center"
27             VerticalAlignment="Center" HorizontalAlignment="Center"
28             Foreground="Yellow"/>
29
30         <!-- Ellipse with a SolidColorBrush (just a Fill) -->
31         <Ellipse Fill="Yellow" Grid.Row="1" Margin="20"/>
32
33         <!-- TextBlock with an ImageBrush -->
34         <TextBlock TextWrapping="Wrap" Text="Image" FontSize="200"
35             FontWeight="ExtraBold" HorizontalAlignment="Center"
36             VerticalAlignment="Center" Grid.Column="1">

```

Fig. 26.3 | Applying brushes. (Part 1 of 2.)

```

37         <TextBlock.Foreground>
38             <!-- Flower image as an ImageBrush -->
39             <ImageBrush ImageSource="Assets/flowers.jpg"
40                 Stretch="UniformToFill"/>
41         </TextBlock.Foreground>
42     </TextBlock>
43
44     <!-- Ellipse with an ImageBrush -->
45     <Ellipse Grid.Row="1" Grid.Column="1" Margin="20">
46         <Ellipse.Fill>
47             <!-- Flower image as an ImageBrush -->
48             <ImageBrush ImageSource="Assets/flowers.jpg"
49                 Stretch="UniformToFill"/>
50         </Ellipse.Fill>
51     </Ellipse>
52 </Grid>
53 </Page>

```

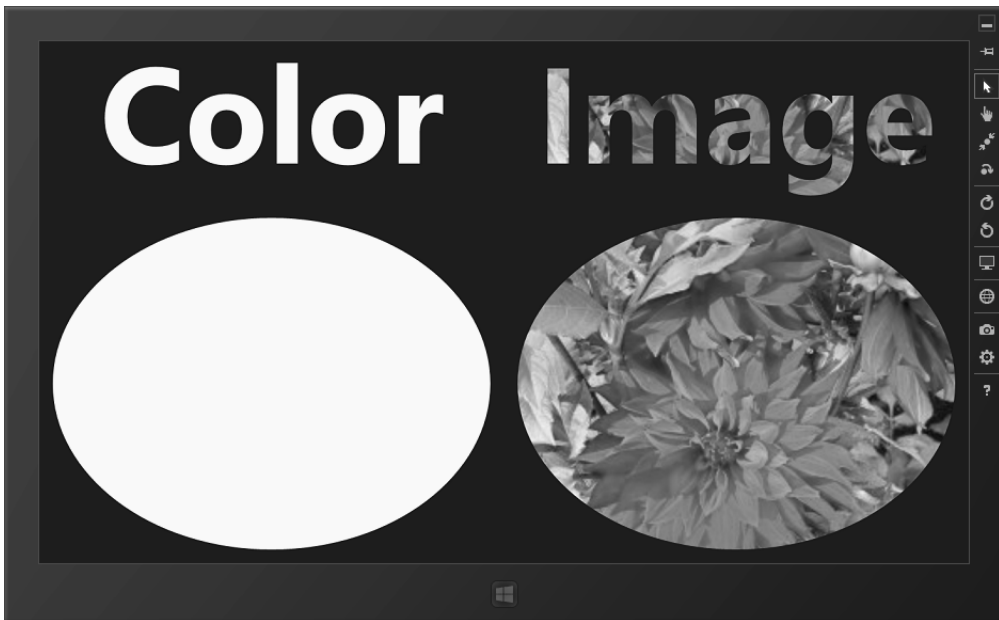



Fig. 26.3 | Applying brushes. (Part 2 of 2.)

SolidColorBrush

A **SolidColorBrush** specifies a solid color to apply to a property. All of the colors specified for the Strokes and Fills in this chapter's previous examples were actually SolidColorBrushs. For the TextBlock at lines 25–28, the **Foreground** property specifies the text color as a *color name* (Yellow). For the Ellipse at line 31, this is specified with the **Fill** property. If you're editing the XAML directly, the IDE's *IntelliSense* feature will display a drop-down list of *predefined* color names that you can use.

You may also specify any solid color you like via the **Properties** window's **Brush** section. To do so, click the  tab below the **Foreground** property. You can use any *custom*


color created from a combination of *alpha* (transparency), *red*, *green* and *blue* components called **ARGB values**—each is an integer in the range 0–255 that defines the amount of alpha, red, green and blue in the color, respectively. Custom colors are defined in *hexadecimal (base 16) format* (see Appendix D) in XAML, so the ARGB components are converted to hexadecimal values in the range 00–FF. An alpha value of 0 is *completely transparent* and an alpha value of 255 is *completely opaque*. In the IDE, the alpha is specified as a percentage from 0 to 100, which the IDE then translates into 00–FF.

If you know the hexadecimal value for the exact color you need, you can type it in the textbox that shows the current color’s hexadecimal value. Use the format #AARRGGBB. You can see the complete list of predefined colors and their hexadecimal values at

msdn.microsoft.com/en-us/library/system.windows.media.colors.aspx

ImageBrush

An **ImageBrush** paints an image into the property it’s assigned to. For instance, the TextBlock with the text “Image” and the Ellipse below it are both filled with the same flower picture. To fill the text, assign the ImageBrush to the Foreground property of the TextBlock. For brushes like ImageBrushes that have more complex settings, nested XAML elements are used to define the brushes and specify the property to which the brushes are assigned, as shown in lines 37–41 for the TextBlock’s Foreground property and lines 46–50 for the Ellipse’s Fill property.

To specify an ImageBrush for the TextBlock, click the  tab below the **Foreground** property, then specify values for the **Stretch** and **ImageSource** properties. In this app, the Stretch value UniformToFill indicates that the image should fill the element in which it’s displayed and 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—that is, you’ll see only a portion of the image. Other possibilities for Stretch are:

- **None**—Uses the image at its *original* size. Depending on the image size, it’s possible that the image will not fill the area or that only a small portion of the image will be displayed.
- **Fill**—Resizes the image to the width and height of the area it’s filling. This typically changes the image’s *aspect ratio*.
- **Uniform**—Preserves the image’s aspect ratio but resizes it to fit the *width or height* of the area it’s filling. Some of the area may *not* be filled by the image.

The ImageSource is set to an image file in the project. All of the project’s images are displayed in this property’s combobox, so you can conveniently select from them.

26.5 GradientBrushes

A **gradient** is a *gradual transition* through two or more colors. Gradients can be applied as the background or fill for various elements. A **LinearGradientBrush** transitions through colors along a *straight* path. The UsingGradients example (Figs. 26.4 and 26.6) displays a gradient across the window. This was created by applying a LinearGradientBrush to a Rectangle’s Fill. The gradient starts with white and transitions linearly to black from *left to right*. This app allows you to set the ARGB values of the start and end colors to change

the look of the gradient. The values entered in the TextBoxes must be in the range 0–255 for the app to run properly. If you set either color's *alpha* value to less than 255 (as in Fig. 26.4(c)), 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. 26.4.

TextBlock That's Initially Hidden Behind the Gradient

Lines 15–17 of Fig. 26.4 define a TextBlock containing the text "Transparency test", which is not visible initially because it's behind a Rectangle containing the gradient, which is completely *opaque*.

```

1  <!-- Fig. 26.4: MainWindow.xaml -->
2  <!-- Defining gradients in XAML. -->
3  <Page
4      x:Class="UsingGradients.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:UsingGradients"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <Grid
13         Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
14         <!-- TextBlock in the background to show transparency -->
15         <TextBlock TextWrapping="Wrap" Text="Transparency test"
16             FontSize="200" HorizontalAlignment="Center"
17             VerticalAlignment="Center" TextAlignment="Center"/>
18
19         <!-- sample rectangle with linear gradient fill -->
20         <Rectangle>
21             <Rectangle.Fill>
22                 <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
23                     <GradientStop x:Name="firstStop" Color="White"/>
24                     <GradientStop x:Name="secondStop" Color="Black"
25                         Offset="1"/>
26                 </LinearGradientBrush>
27             </Rectangle.Fill>
28         </Rectangle>
29
30         <!-- Controls for selecting color and transparency -->
31         <StackPanel VerticalAlignment="Top" HorizontalAlignment="Center">
32             <!-- shows which TextBox corresponds with which ARGB value-->
33             <StackPanel Orientation="Horizontal">
34                 <TextBlock TextWrapping="Wrap" Text="Alpha:"
35                     Width="75" Margin="8" FontSize="20"/>
36                 <TextBlock TextWrapping="Wrap" Text="Red:"
37                     Width="75" Margin="8" FontSize="20"/>
38                 <TextBlock TextWrapping="Wrap" Text="Green:"
39                     Width="75" Margin="8" FontSize="20"/>

```

Fig. 26.4 | Defining gradients in XAML. (Part I of 3.)

```

40         <TextBlock TextWrapping="Wrap" Text="Blue:"
41             Width="75" Margin="8" FontSize="20"/>
42     </StackPanel>
43
44     <!-- GUI to select the color of the first GradientStop -->
45     <StackPanel Grid.Row="2" Orientation="Horizontal">
46         <TextBox Name="fromAlpha" TextWrapping="Wrap" Text="255"
47             Width="75" Margin="8" TextAlignment="Center"
48             FontSize="20"/>
49         <TextBox Name="fromRed" TextWrapping="Wrap" Text="255"
50             Width="75" Margin="8" TextAlignment="Center"
51             FontSize="20"/>
52         <TextBox Name="fromGreen" TextWrapping="Wrap" Text="255"
53             Width="75" Margin="8" TextAlignment="Center"
54             FontSize="20"/>
55         <TextBox Name="fromBlue" TextWrapping="Wrap" Text="255"
56             Width="75" Margin="8" TextAlignment="Center"
57             FontSize="20"/>
58         <Button Name="startColorButton" Content="Set Start Color"
59             Width="150" Margin="8" Click="startColorButton_Click"/>
60     </StackPanel>
61
62     <!-- GUI to select the color of second GradientStop -->
63     <StackPanel Grid.Row="3" Orientation="Horizontal">
64         <TextBox Name="toAlpha" TextWrapping="Wrap" Text="255"
65             Width="75" Margin="8" TextAlignment="Center"
66             FontSize="20"/>
67         <TextBox Name="toRed" TextWrapping="Wrap" Text="0"
68             Width="75" Margin="8" TextAlignment="Center"
69             FontSize="20"/>
70         <TextBox Name="toGreen" TextWrapping="Wrap" Text="0"
71             Width="75" Margin="8" TextAlignment="Center"
72             FontSize="20"/>
73         <TextBox Name="toBlue" TextWrapping="Wrap" Text="0"
74             Width="75" Margin="8" TextAlignment="Center"
75             FontSize="20"/>
76         <Button Name="endColorButton" Content="Set End Color"
77             Width="150" Margin="8" Click="endColorButton_Click"/>
78     </StackPanel>
79 </StackPanel>
80 </Grid>
81 </Page>

```

a) Controls for setting the start and end colors

Alpha:	Red:	Green:	Blue:	
255	255	255	255	Set Start Color
255	0	0	0	Set End Color

Fig. 26.4 | Defining gradients in XAML. (Part 2 of 3.)

b) The application immediately after it's loaded



c) The application after changing the start and end colors

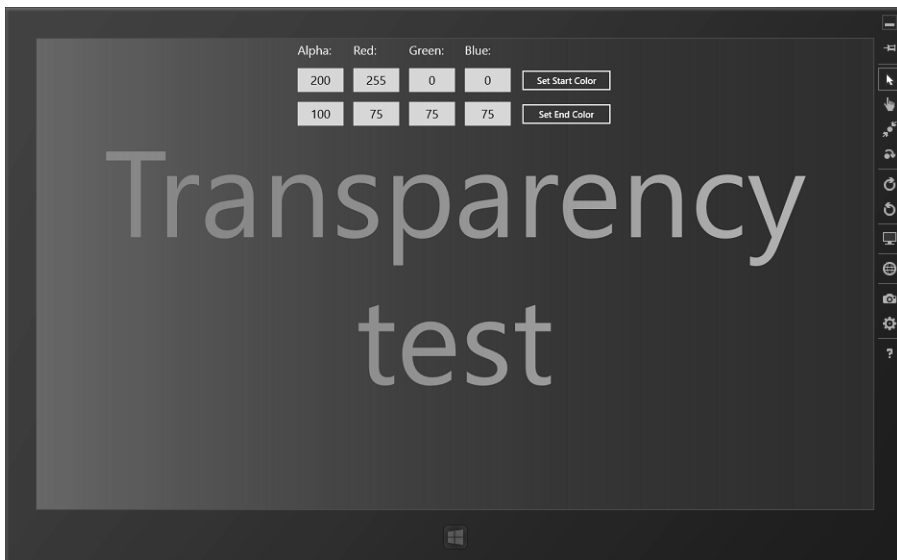


Fig. 26.4 | Defining gradients in XAML. (Part 3 of 3.)

26.5.1 Setting the Rectangle's Fill to a Gradient

The `Rectangle` with a `LinearGradientBrush` applied to its `Fill` is defined in lines 20–28. Line 22 defines the gradient's **StartPoint** and **EndPoint**. You must assign these properties **logical points** that are defined as x - and y -coordinates with values between 0.0 and 1.0,

inclusive. *Logical points* reference locations in the control independent of the actual size. The point (0,0) represents the control's *top-left corner* while the point (1,1) represents the *bottom-right corner*. The point (1,0) used in this example represents the top-right corner. The gradient will transition *linearly* from the start point to the end point.

26.5.2 GradientStops

A gradient's colors are defined by **GradientStops** (lines 23–25)—each specifies a single color along the gradient. You can define many stops. A **GradientStop**'s **Color** property defines the color you want at that **GradientStop**'s location—lines 23 and 24 indicate that the gradient transitions from white to black. Set the colors of the two **GradientStops** as shown in Fig. 26.5. A **GradientStop**'s **Offset** property defines where along the linear transition you want the color to appear. In the example we use the default value (0.0) for the first **GradientStop**'s **Offset** and 1.0 for the other **GradientStop**'s **Offset** (Fig. 26.4, line 25), indicating that these colors appear at the start and end of the gradient, respectively. You can change the **GradientStops**' positions by dragging the *thumbs* that represent them on the sample gradient in the **Properties** window (Fig. 26.5). Both **GradientStops** in this example have `x:Name` properties (that we added directly in the XAML) so we can interact with them *programmatically* to change their colors.

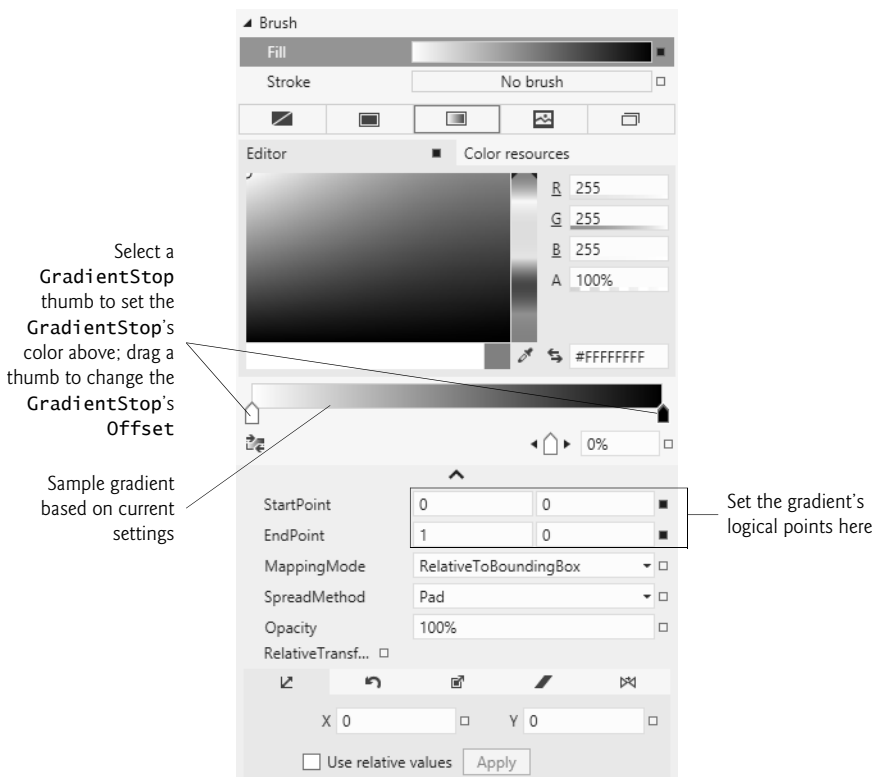



Fig. 26.5 | Brush section of **Properties** window for a **Rectangle**.

26.5.3 Defining the Gradient in the IDE

To define the gradient, click the Rectangle's Fill property's value in the **Properties** window's **Brush** section, then click the  tab below the **Fill** and **Stroke** properties. Set the colors of the GradientStops and the **StartPoint** and **EndPoint** as shown in Fig. 26.5.

26.5.4 Code-Behind File

The code-behind file (Fig. 26.6) responds to the user setting the Colors of the two GradientStops. When the user presses **Set Start Color**, we use the Text properties of the corresponding TextBoxes to obtain the ARGB values and create a new color that we then assign to firstStop's Color property (lines 23–27). Each TextBox's value is converted to a byte by Convert method ToByte—we assume in this example that the user enters a value in the range 0–255 for each color component. Color method FromArgb receives four bytes that specify the alpha, red, green and blue components of a color. Each byte can represent values in the range 0–255. When the user presses **Set End Color**, we do the same for secondStop's Color (lines 35–39).

```

1  // Fig. 26.6: MainPage.xaml.cs
2  // Customizing gradients.
3  using System;
4  using Windows.UI;
5  using Windows.UI.Xaml;
6  using Windows.UI.Xaml.Controls;
7
8  namespace UsingGradients
9  {
10     public sealed partial class MainPage : Page
11     {
12         // constructor
13         public MainPage()
14         {
15             InitializeComponent();
16         } // end constructor
17
18         // change the starting color of the gradient when the user clicks
19         private void startColorButton_Click(object sender,
20             RoutedEventArgs e)
21         {
22             // change the color to use the ARGB values specified by user
23             firstStop.Color = Color.FromArgb(
24                 Convert.ToByte(fromAlpha.Text),
25                 Convert.ToByte(fromRed.Text),
26                 Convert.ToByte(fromGreen.Text),
27                 Convert.ToByte(fromBlue.Text));
28         } // end method startColorButton_Click
29
30         // change the ending color of the gradient when the user clicks
31         private void endColorButton_Click(object sender,
32             RoutedEventArgs e)
33         {

```

Fig. 26.6 | Customizing gradients. (Part I of 2.)

```

34         // change the color to use the ARGB values specified by user
35         secondStop.Color = Color.FromArgb(
36             Convert.ToByte(toAlpha.Text),
37             Convert.ToByte(toRed.Text),
38             Convert.ToByte(toGreen.Text),
39             Convert.ToByte(toBlue.Text));
40     } // end method endColorButton_Click
41 } // end class MainPage
42 } // end namespace UsingGradients

```

Fig. 26.6 | 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. In this section, we demonstrate the **RotateTransform**, which *rotates* an object around a point by a specified *rotation angle*. Other transforms include:

- **ScaleTransform**—*scales* the object along the *x*-axis and *y*-axis.
- **SkewTransform**—*skews* (or *shears*) the object.
- **TranslateTransform**—*moves* an object to a new location.

You can apply multiple transforms to a control by grouping them in a **TransformGroup**.

The DrawStars example (Figs. 26.7–26.8) creates a circle of randomly colored stars. The app starts by creating a single Polygon *programmatically*, then makes 18 copies of the star and uses **RotateTransforms** to place each star around the circle at 20 degree intervals. Figure 26.7 shows the XAML code and a sample output. Lines 13–15 define a Canvas in which the shapes will be displayed. When the Canvas's **PointerPressed** event occurs, it's event handler recreates the stars with different randomly chosen colors. The code-behind file is shown in Fig. 26.8.

```

1  <!-- Fig. 26.7: MainWindow.xaml -->
2  <!-- Transforming a star polygon to display a circle of stars. -->
3  <Page
4      x:Class="DrawStars.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:DrawStars"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <!-- Canvas that displays the stars -->
13     <Canvas Name="mainCanvas"
14         Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
15         PointerPressed="mainCanvas_PointerPressed"/>
16 </Page>

```

Fig. 26.7 | Transforming a star polygon to display a circle of stars. (Part 1 of 2.)



Fig. 26.7 | Transforming a star polygon to display a circle of stars. (Part 2 of 2.)

```

1 // Fig. 26.8: MainPage.xaml.cs
2 // Applying transforms to a Polygon.
3 using System;
4 using Windows.Foundation;
5 using Windows.UI;
6 using Windows.UI.Xaml;
7 using Windows.UI.Xaml.Controls;
8 using Windows.UI.Xaml.Input;
9 using Windows.UI.Xaml.Media;
10 using Windows.UI.Xaml.Shapes;
11
12 namespace DrawStars
13 {
14     public sealed partial class MainPage : Page
15     {
16         Random random = new Random(); // for random color values
17         Polygon star = new Polygon(); // used to define star
18
19         // constructor
20         public MainPage()
21         {
22             this.InitializeComponent();
23
24             // determine horizontal center of screen
25             int centerX =
26                 Convert.ToInt32(Window.Current.Bounds.Width / 2);
27
28             // set initial star points
29             star.Points.Add(new Point(centerX, 50));

```

Fig. 26.8 | Applying transforms to a Polygon. (Part 1 of 2.)


```

30         star.Points.Add(new Point(centerX + 12, 86));
31         star.Points.Add(new Point(centerX + 44, 86));
32         star.Points.Add(new Point(centerX + 18, 104));
33         star.Points.Add(new Point(centerX + 28, 146));
34         star.Points.Add(new Point(centerX, 122));
35         star.Points.Add(new Point(centerX - 28, 146));
36         star.Points.Add(new Point(centerX - 18, 104));
37         star.Points.Add(new Point(centerX - 44, 86));
38         star.Points.Add(new Point(centerX - 12, 86));
39
40         RotateAndDrawStars(); // draw circle of stars
41     } // end constructor
42
43     // draw circle of stars in random colors
44     private void RotateAndDrawStars()
45     {
46         mainCanvas.Children.Clear(); // remove previous stars
47
48         // create 18 stars
49         for (int count = 0; count < 18; ++count)
50         {
51             Polygon newStar = new Polygon(); // create a polygon object
52
53             // copy star.Points collection into newStar.Points
54             foreach (var point in star.Points)
55                 newStar.Points.Add(point);
56
57             byte[] colorValues = new byte[3]; // create a Byte array
58             random.NextBytes(colorValues); // create three random values
59
60             // creates a random color brush
61             newStar.Fill = new SolidColorBrush(Color.FromArgb(
62                 255, colorValues[0], colorValues[1], colorValues[2]));
63
64             // apply a rotation to the shape
65             // used to rotate the star
66             RotateTransform rotate = new RotateTransform();
67             rotate.CenterX = Window.Current.Bounds.Width / 2;
68             rotate.CenterY = Window.Current.Bounds.Height / 2;
69             rotate.Angle = count * 20;
70             newStar.RenderTransform = rotate;
71             mainCanvas.Children.Add(newStar);
72         } // end for
73     } // end method RotateAndDrawStars
74
75     // redraws stars in new colors each time user touches the canvas
76     private void mainCanvas_PointerPressed(object sender,
77         PointerRoutedEventArgs e)
78     {
79         RotateAndDrawStars();
80     } // end method mainCanvas_PointerPressed
81 } // end class MainPage
82 } // end namespace DrawStars

```

Fig. 26.8 | Applying transforms to a Polygon. (Part 2 of 2.)

26.6.1 MainPage Instance Variables and Constructor

In the code-behind file, line 17 creates the `Polygon` that's used to define the initial star shape. The `MainPage` constructor determines the screen's horizontal center *x*-coordinate (lines 25–26), then adds `Point` objects (lines 29–38) to `Polygon` star's `Points` property—this is the programmatic equivalent of setting the `Points` property in XAML as we did in Fig. 26.2. Each `Point`'s *x*-coordinate is calculated based on the *x*-coordinate of the screen's horizontal center. These points define the star that appears at the top-center of the screen.

26.6.2 MainPage Method RotateAndDrawStars

Method `RotateAndDrawStars` (lines 44–73) is called from the constructor and the `mainCanvas_PointerPressed` event handler. The method replicates the star object 18 times and applies a different `RotateTransform` to each to get the circle of `Polygons` shown in the screen capture of Fig. 26.7. Each iteration of the loop creates a new `Polygon` with the *same* set of points (Fig. 26.8, lines 51–55). To generate the random colors for each star, we use the `Random` class's `NextBytes` method, which assigns a random value in the range 0–255 to each element in its byte-array argument. Lines 57–58 define a four-element `Byte` array and pass it to `NextBytes`. We then set the `newStar`'s `Fill` to a new `SolidColorBrush` with 255 for the *alpha* and the three randomly generated bytes as its *red*, *green* and *blue* values (lines 61–62).

26.6.3 Applying a RotateTransform

To apply a rotation to each new `Polygon`, we set its `RenderTransform` property to a new `RotateTransform` object (lines 66–70). Lines 67–68 set the `RotateTransform`'s `CenterX` and `CenterY` properties, which represent the coordinates of the *point of rotation*—that is, the point around which the shape will rotate. We used the screen's center *x-y* coordinates. Next, line 69 sets the `Angle` property, which indicates the number of *degrees to rotate*. A *positive* value rotates *clockwise*. Each iteration of the loop in `RotateAndDrawStars` assigns a new rotation-angle value by multiplying the control variable by 20. After setting `newStar`'s `RenderTransform` property (line 70), line 71 adds `newStar` as a child element of `mainCanvas` so it can be rendered on screen.

26.7 Windows 8 Customization: A Television GUI

In Chapter 25, we introduced styles for customizing the appearance of Windows 8 controls. Now that you have a basic understanding of how to create and manipulate graphics in Windows 8, you'll use **ControlTemplates** to define a control's appearance and to create an app with a graphically sophisticated multimedia GUI (Fig. 26.9). This app models a *television* screen in perspective. The TV can be turned on or off. When it's on, the user can *play*, *pause* and *stop* the TV's video. The video we used (`media.mp4`) is located with the chapter's examples in the `ImagesVideo` folder. You should create a `Video` subfolder in the project's `Assets` folder, then drag the video file from Windows Explorer into the `Video` subfolder in the **Solution Explorer** window. We downloaded this public-domain NASA video from

www.nasa.gov/multimedia/videogallery



Fig. 26.9 | GUI representing a television with video playing.

26.7.1 XAML for the TV GUI

The TV GUI may appear complex, but it's actually just a basic Windows 8 GUI built using standard controls that you've learned previously. Figures 26.10–26.12 present the XAML markup. When the app first loads, the power `CheckBox` (⏻) at the GUI's bottom-left is colored red to indicate that the power is *off*, and the *play* (▶), *pause* (⏸) and *stop* (■) `RadioButtons` at the GUI's bottom-right are gray to indicate that they're *disabled*. When you check the power `CheckBox`, it becomes green to indicate that the power is *on* and the *play*, *pause* and *stop* `RadioButtons` turn red to indicate that they're *unchecked*. As you select each `RadioButton`, it turns green to indicate that it's *checked*. As always, only *one* `RadioButton` in the group can be selected at a time.

When we built this app's GUI, we first defined the elements in Fig. 26.12, then implemented the `ControlTemplates` in Figs. 26.10–26.11 to customize the `CheckBox` and `RadioButton` controls.

26.7.2 XAML for the Power `CheckBox`'s `ControlTemplate`

Figure 26.10 begins `MainPage`'s XAML markup. Within the `Page.Resources` element (starting at line 12 and spanning Figs. 26.10–26.11) are two `ControlTemplates`. The one in Fig. 26.10 specifies the look-and-feel of the app's `powerCheckBox`.

```

1  <!-- Fig. 26.10: MainPage.xaml -->
2  <!-- TV GUI showing the versatility of Windows 8 customization. -->
3  <Page
4      x:Class="TV.MainPage"
5      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
6      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
7      xmlns:local="using:TV"
8      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
9      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
10     mc:Ignorable="d">
11
12     <Page.Resources>
13         <!-- define ControlTemplate for power CheckBox -->
14         <ControlTemplate x:Key="CheckBoxControlTemplate"
15             TargetType="CheckBox">
16             <Grid>
17                 <!-- CheckBox checked/unchecked appearance changes -->
18                 <VisualStateManager.VisualStateGroups>
19                     <VisualStateGroup x:Name="CheckStates">
20                         <VisualState x:Name="Checked">
21                             <Storyboard>
22                                 <ObjectAnimationUsingKeyFrames
23                                     Storyboard.TargetProperty="Color"
24                                     Storyboard.TargetName="startColor">
25                                     <DiscreteObjectKeyFrame KeyTime="0"
26                                         Value="LimeGreen"/>
27                                 </ObjectAnimationUsingKeyFrames>
28                             </Storyboard>
29                         </VisualState>
30                         <VisualState x:Name="Unchecked" />
31                     </VisualStateGroup>
32                 </VisualStateManager.VisualStateGroups>
33
34                 <!-- Red circle background for CheckBox -->
35                 <Ellipse Stroke="Black" Width="50" Height="50">
36                     <Ellipse.Fill>
37                         <LinearGradientBrush StartPoint="0,0" EndPoint="2,0">
38                             <GradientStop x:Name="startColor" Color="Red"/>
39                             <GradientStop Color="Black" Offset="1"/>
40                         </LinearGradientBrush>
41                     </Ellipse.Fill>
42                 </Ellipse>
43
44                 <!-- display button image -->
45                 <ContentPresenter Content="{TemplateBinding Content}" />
46             </Grid>
47         </ControlTemplate>
48

```

Fig. 26.10 | TV GUI XAML—CheckBoxControlTemplate that specifies the powerCheckBox's custom look-and-feel.

26.7.3 Creating a Basic ControlTemplate

You can create a ControlTemplate one of several ways:

- You can create a `Page.Resources` element in the XAML and edit the XAML directly.
- You can edit a copy of a control's default `Style`, which contains the control's default ControlTemplate. To do this, in **Design** view you'd right-click the control you wish to customize in your GUI, then select **Edit Template > Edit a Copy....** In the dialog that appears, you can specify the ControlTemplate's name (known as its *key*) and where the ControlTemplate should be defined. For example, you can define it in the Page's resources to share it among many elements on the Page, or in the app's resources to share it across pages in the app.
- You can define a new ControlTemplate then edit its content. We used this approach. To do this, right-click the control you wish to customize in your GUI, then select **Edit Template > Create Empty....** In the dialog that appears, specify the name `CheckBoxControlTemplate`. Leave the **This document** radio button selected and ensure that **Page** is selected in the drop-down list. This places the XAML for a basic ControlTemplate in your Page's `Page.Resources` element.

26.7.4 Default ControlTemplate Markup

The default ControlTemplate markup that the IDE generated contains:

- An **x:Key** property (line 14)—the ControlTemplate name you specified. This name is used in the `CheckBox` control's definition in Fig. 26.12 to specify which ControlTemplate should be applied. As you'll see shortly, because you right-clicked the `CheckBox` to create this ControlTemplate, the IDE uses it to change the `CheckBox`'s look and feel.
- A **TargetType** property (line 15)—the type of control that the template can be applied to.
- A nested `Grid` element (starting at line 16)—used to define the control's look and feel.
- A nested **VisualStateManager.VisualGroups** element within the `Grid`—used to specify changes to the control's look and feel due to various events and user interactions, such as when the `CheckBox` is *checked* or *unchecked*. We kept only the items in this element that we needed to customize the `CheckBox` appropriately for this app.

When you place controls in a ControlTemplate and select them, you can edit their properties in the **Properties** window or directly in the XAML.

26.7.5 Defining the CheckBox's New Look-And-Feel

The template defines a one-cell `Grid` containing a 50-by-50 `Ellipse` with a red-to-black gradient `Fill` (lines 35–42) on which we display the `CheckBox`'s `Content` property value. Normally this is text that appears next to the checkbox, but this app uses an image instead. Unless otherwise specified, the elements in a one-cell `Grid` are centered horizontally *and* vertically within the `Grid`, so the `CheckBox`'s `Content` value appears centered over the `El-`

Tipse—recall that the element defined later in the XAML is placed on top. Together, the `Ellipse` and image *completely change* the default look-and-feel of a standard `CheckBox`, but the control maintains its ability to be *checked* or *unchecked*.

The `ControlTemplate`'s **ContentPresenter** element (line 45) specifies where the `CheckBox`'s Content value should be displayed. The `ContentPresenter`'s Content property is set to "{TemplateBinding Content}", which indicates that the `CheckBox`'s Content value should appear at the `ContentPresenter`'s location in the template.

26.7.6 Using Animation to Change a Control's Look-and-Feel in Response to State Changes

Lines 18–32 specify how the control's appearance should change as the control's state changes—for example, from *checked* to *unchecked*. The `CheckBox` in this app is *unchecked* by default, so it will initially appear as defined by lines 35–42 within the `Grid`. The `VisualStateManager.VisualGroups` element may contain one or more `VisualStateGroup` elements that define various control states. For this `CheckBox`, we have one `VisualStateGroup` that defines two `VisualState` elements—one with the `x:Name` "Checked" (lines 20–29) and one with "Unchecked" (line 30). These are just two of many `VisualStates` that a `CheckBox` can have—we deleted the others for simplicity in this app. You must use the pre-defined state names (which you can see in the basic `ControlTemplate` when it's first generated) so that the app can look up and apply the correct `VisualState` each time the control's state changes.

The empty `VisualState` element at line 30

```
<VisualState x:Name="Unchecked" />
```

indicates that when the `CheckBox` is *unchecked*, it should return to the default appearance defined by the `ControlTemplate`.

The `VisualState` element at lines 20–29

```
<VisualState x:Name="Checked">
  <Storyboard>
    <ObjectAnimationUsingKeyFrames
      Storyboard.TargetProperty="Color"
      Storyboard.TargetName="startColor">
      <DiscreteObjectKeyFrame KeyTime="0"
        Value="LimeGreen"/>
    </ObjectAnimationUsingKeyFrames>
  </Storyboard>
</VisualState>
```

indicates that when the `CheckBox` is *checked*, the `GradientStop` named `startColor` should have its `Color` property changed to `LimeGreen`. Property value changes are specified in the nested `Storyboard` element, which is used to perform *animations*. There are many animation capabilities in Windows 8 (`bit.ly/Win8Storyboard`). Here, we use an **ObjectAnimationUsingKeyFrames** animation, which allows you to change a property of an object to a new value. `Storyboard.TargetName` specifies the object that will be modified and `Storyboard.TargetProperty` specifies the property to change. Within the `ObjectAnimationUsingKeyFrames` element, the nested `DiscreteObjectKeyFrame` specifies the new color value (`LimeGreen`) and that the color change should be performed im-

mediately (KeyTime is set to 0). You can also specify animations that are performed over a period of time. Some other animation types include animating Points so that you can move an object, animating transitions from one color to another over time and animating numeric values over time—this could be used to animate an object's size changes.

26.7.7 XAML for the Play, Pause and Stop RadioButtons' ControlTemplate

Figure 26.11 presents the ControlTemplate that specifies the look and feel of the app's playRadioButton, pauseRadioButton and stopRadioButton. Most of this markup is similar or identical to that of the CheckBoxControlTemplate in Fig. 26.10. The important differences are in lines 54–81 where we define the VisualStates for the RadioButtons' Normal, Disabled, Checked and Unchecked states using the same animation capabilities as in the CheckBoxControlTemplate.

```

49      <!-- define template for play, pause and stop RadioButtons -->
50      <ControlTemplate x:Key="RadioButtonControlTemplate"
51          TargetType="RadioButton">
52          <Grid>
53              <!-- RadioButton appearance changes -->
54              <VisualStateManager.VisualStateGroups>
55                  <VisualStateGroup x:Name="CommonStates">
56                      <VisualState x:Name="Normal" />
57                      <VisualState x:Name="Disabled">
58                          <Storyboard>
59                              <ObjectAnimationUsingKeyFrames
60                                  Storyboard.TargetProperty="Color"
61                                  Storyboard.TargetName="startColor">
62                                  <DiscreteObjectKeyFrame KeyTime="0"
63                                      Value="LightGray"/>
64                                  </ObjectAnimationUsingKeyFrames>
65                              </Storyboard>
66                          </VisualState>
67                  </VisualStateGroup>
68                  <VisualStateGroup x:Name="CheckStates">
69                      <VisualState x:Name="Checked">
70                          <Storyboard>
71                              <ObjectAnimationUsingKeyFrames
72                                  Storyboard.TargetProperty="Color"
73                                  Storyboard.TargetName="startColor">
74                                  <DiscreteObjectKeyFrame KeyTime="0"
75                                      Value="LimeGreen"/>
76                                  </ObjectAnimationUsingKeyFrames>
77                              </Storyboard>
78                          </VisualState>
79                      <VisualState x:Name="Unchecked" />
80                  </VisualStateGroup>
81              </VisualStateManager.VisualStateGroups>
82          </Grid>

```

Fig. 26.11 | TV GUI XAML—RadioButtonControlTemplate that specifies the RadioButtons' custom look-and-feel. (Part I of 2.)

```

83         <!-- Red circle background for RadioButtons -->
84         <Ellipse Name="backgroundEllipse" Stroke="Black"
85             Width="50" Height="50">
86             <Ellipse.Fill>
87                 <LinearGradientBrush StartPoint="0,0" EndPoint="2,0">
88                     <GradientStop x:Name="startColor" Color="Red"/>
89                     <GradientStop Color="Black" Offset="1"/>
90                 </LinearGradientBrush>
91             </Ellipse.Fill>
92         </Ellipse>
93
94         <!-- display button image -->
95         <ContentPresenter Content="{TemplateBinding Content}" />
96     </Grid>
97 </ControlTemplate>
98 </Page.Resources>
99

```

Fig. 26.11 | TV GUI XAML—RadioButtonControlTemplate that specifies the RadioButtons' custom look-and-feel. (Part 2 of 2.)

26.7.8 XAML for the GUI

Figure 26.12 presents the XAML for the app's controls. The main Grid contains a Border object (lines 104–165) that surrounds the entire GUI and defines the TV's size. To make the TV appear as if it's in perspective from a point in the distance, we selected the Border and used the Properties windows Transform section to specify a Projection, which gives an object a 3-D look that's known as a **perspective projection**. The projection is applied to *everything* nested in the Border element. You can also apply such projections to individual controls. For this app, we set the Projection's Y value to 20 which created the Border.Projection element in lines 109–111. Try setting the X and Z values individually in this app's project to see how the *x*- and *z*-axis projections differ from the *y*-axis projection we used here.

```

100 <!-- Grid that defines the GUI -->
101 <Grid
102     Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
103     <!-- define the "TV" -->
104     <Border x:Name="tvBorder" Width="800" Height="540"
105         HorizontalAlignment="Center" VerticalAlignment="Center"
106         Background="Silver">
107
108         <!-- Make the TV look like its in perspective -->
109         <Border.Projection>
110             <PlaneProjection RotationY="20"/>
111         </Border.Projection>
112
113         <!-- Screen within the Border's bounds -->
114         <Grid>
115             <Grid.RowDefinitions>

```

Fig. 26.12 | TV GUI XAML—Grid containing the GUI. (Part 1 of 2.)


```

116         <RowDefinition />
117         <RowDefinition Height="Auto" />
118     </Grid.RowDefinitions>
119
120     <!-- define the screen -->
121     <Border Margin="10" Background="Black"
122         BorderThickness="2" BorderBrush="Silver">
123         <MediaElement Name="videoMediaElement" AutoPlay="False" />
124     </Border>
125
126     <!-- define the play, pause, and stop buttons -->
127     <StackPanel Grid.Row="1" HorizontalAlignment="Right"
128         Orientation="Horizontal">
129         <RadioButton Name="playRadioButton" IsEnabled="False"
130             Margin="15" Checked="playRadioButton_Checked"
131             Template="{StaticResource RadioButtonControlTemplate}">
132             <!-- displayed by ControlTemplate's ContentPresenter -->
133             <Image Width="40" Height="40"
134                 Source="Assets/Images/play.png" Stretch="Uniform" />
135         </RadioButton>
136
137         <RadioButton Name="pauseRadioButton" IsEnabled="False"
138             Margin="15" Checked="pauseRadioButton_Checked"
139             Template="{StaticResource RadioButtonControlTemplate}">
140             <!-- displayed by ControlTemplate's ContentPresenter -->
141             <Image Width="40" Height="40"
142                 Source="Assets/Images/pause.png" Stretch="Uniform" />
143         </RadioButton>
144
145         <RadioButton Name="stopRadioButton" IsEnabled="False"
146             Margin="15" Checked="stopRadioButton_Checked"
147             Template="{StaticResource RadioButtonControlTemplate}">
148             <!-- displayed by ControlTemplate's ContentPresenter -->
149             <Image Width="40" Height="40"
150                 Source="Assets/Images/stop.png" Stretch="Uniform" />
151         </RadioButton>
152     </StackPanel>
153
154     <!-- define the power button -->
155     <CheckBox Name="powerCheckBox" Grid.Row="1"
156         HorizontalAlignment="Left"
157         Margin="15" Checked="powerCheckBox_Checked"
158         Unchecked="powerCheckBox_Unchecked"
159         Template="{StaticResource CheckBoxControlTemplate}">
160         <!-- displayed by ControlTemplate's ContentPresenter -->
161         <Image Source="Assets/Images/power.png"
162             Width="40" Height="40" />
163     </CheckBox>
164 </Grid>
165 </Border>
166 </Grid>
167 </Page>

```

Fig. 26.12 | TV GUI XAML—Grid containing the GUI. (Part 2 of 2.)

Within the `Border` is a two-row `Grid`. Lines 121–124 define the first row, which contains a `Border` with a nested **MediaElement**—used to add audio or video to a Windows 8 app. Before using an audio or video file in your app, add it to your project's `Assets` folder. In this app, we'll set the `MediaElement`'s **Source** property *programmatically* in response to user interactions.

Lines 127–152 define a horizontal `StackPanel` containing the *play*, *pause* and *stop* `RadioButtons` that appear at the right side of the `Grid`'s second row. Each `RadioButton`'s `Template` property (lines 131, 139 and 147) binds the `RadioButton` to the `RadioButtonControlTemplate` from Fig. 26.11. Nested in each `RadioButton` element is an `Image` element (e.g., lines 133–134) that specifies the `RadioButton`'s `Content` property value—the `RadioButtonControlTemplate`'s `ContentPresenter` uses this `Image` on the `RadioButton`.

Lines 155–163 define the power `CheckBox` that appears at the left side of the `Grid`'s second row. Line 159 specifies the `CheckBox`'s `ControlTemplate`.

26.7.9 TV GUI Code-Behind File

Figure 26.13 presents the code-behind file for the TV app. Line 11 declares a `Uri` that specifies the video played by the `MediaElement`. The `Uri` is initialized in the `MainPage` constructor at line 19. The inherited `Page` property `BaseUri` specifies this app's location. The second argument specifies the relative path to the video file within the project's `Assets` folder.

```

1  // Fig. 26.13: MainPage.xaml.cs
2  // Code-behind file for the TV GUI.
3  using System;
4  using Windows.UI.Xaml;
5  using Windows.UI.Xaml.Controls;
6
7  namespace TV
8  {
9      public sealed partial class MainPage : Page
10     {
11         Uri uri; // Uri for location of video in this project
12
13         // constructor
14         public MainPage()
15         {
16             this.InitializeComponent();
17
18             // set Uri for location of video in this project
19             uri = new Uri(BaseUri, @"Assets\Video\media.mp4");
20         } // end constructor
21
22         // turn "on" the TV
23         private void powerCheckBox_Checked( object sender,
24             RoutedEventArgs e )
25         {
26             // set the videoMediaElement's Source
27             videoMediaElement.Source = uri;

```

Fig. 26.13 | Code-behind file for the TV GUI. (Part I of 2.)

```

28
29         // enable play, pause, and stop buttons
30         playRadioButton.IsEnabled = true;
31         pauseRadioButton.IsEnabled = true;
32         stopRadioButton.IsEnabled = true;
33     } // end method powerCheckBox_Checked
34
35     // turn "off" the TV
36     private void powerCheckBox_Unchecked( object sender,
37         RoutedEventArgs e )
38     {
39         // remove the videoMediaElement's source
40         videoMediaElement.Source = null;
41
42         // disable the play, pause, and stop buttons
43         playRadioButton.IsChecked = false;
44         pauseRadioButton.IsChecked = false;
45         stopRadioButton.IsChecked = false;
46         playRadioButton.IsEnabled = false;
47         pauseRadioButton.IsEnabled = false;
48         stopRadioButton.IsEnabled = false;
49     } // end method powerCheckBox_Unchecked
50
51     // play the video
52     private void playRadioButton_Checked( object sender,
53         RoutedEventArgs e )
54     {
55         videoMediaElement.Play();
56     } // end method playRadioButton_Checked
57
58     // pause the video
59     private void pauseRadioButton_Checked( object sender,
60         RoutedEventArgs e )
61     {
62         videoMediaElement.Pause();
63     } // end method pauseRadioButton_Checked
64
65     // stop the video
66     private void stopRadioButton_Checked( object sender,
67         RoutedEventArgs e )
68     {
69         videoMediaElement.Stop();
70     } // end method stopRadioButton_Checked
71 } // end class MainPage
72 } // end namespace TV

```

Fig. 26.13 | Code-behind file for the TV GUI. (Part 2 of 2.)

When the user turns on the TV (i.e., *checks* the `powerCheckBox`), the event handling method `powerCheckBox_Checked` (lines 23–33) sets the `MediaElement`'s `Source` property to the `Uri` representing the video, then enables the *play*, *pause* and *stop* `RadioButtons`. When the user turns the TV off (i.e., *unchecks* the `powerCheckBox`), the event handling method `powerCheckBox_Unchecked` (lines 36–49) sets the `MediaElement`'s `Source` prop-

erty to null (which causes the picture to be removed from the screen), then *unchecks* and *disables* the *play*, *pause* and *stop* RadioButtons.

Whenever one of the playback-option RadioButtons is selected, the corresponding event handler (lines 52–70) calls the MediaElement’s Play, Pause or Stop method. The methods that execute these tasks are built into the MediaElement control.

26.8 Wrap-Up

This chapter introduced several Windows 8 graphics and multimedia capabilities, including two-dimensional shapes, transformations and video. You learned that Windows 8 graphics uses machine-independent pixels—making apps more portable across devices—and that graphic elements are rendered on screen using a vector-based system in which calculations determine how to size and scale each element so that graphics appear smooth at any scale.

You learned how to create basic shapes such as Lines, Rectangles and Ellipses, and set their Fill and Stroke properties.

Next, you created an app that displayed a Polyline and Polygons. These controls allowed you to specify sets of Points in PointCollections. You specified these Points by declaring them in the shapes’ XAML markup.

We discussed several types of brushes for customizing an object’s Fill, Foreground and Background. We demonstrated SolidColorBrush for a single color, ImageBrush for an image and the LinearGradientBrush for a gradient that transitions between multiple colors.

Next, you programmatically created a Polygon and added Points to it, then applied a RotateTransform to reposition and reorient the polygon around a point on the screen.

Finally, in the television GUI app, you used ControlTemplates to completely customize the look of a CheckBox and RadioButtons while maintaining their normal functionality. You defined VisualStates that used animation to specify how a control’s look and feel changes for various states. You also used a projection to apply a perspective effect to the TV GUI.

Exercises

26.1 (*Enhanced UsingGradients app*) Modify the example from Section 26.5 to remove the GUI and specify four GradientStops. You can add GradientStops in the XAML by copying and pasting the existing ones, then changing their x:Name attributes. Position the gradient stops at 0, 0.33, 0.67 and 1.0.

26.2 (*Snake PolyLine app*) Using a Polyline object, create an app that acts like a snake following the mouse cursor (or the user’s finger on a touch screen) around the window. Once the Polyline reaches 20 Points, the app should remove the first Point in the Points property each time it adds another Point. When the user stops dragging the mouse (or stops touching the screen) then starts again, the app should clear the Polyline’s Points and start over. See Section 25.3.3 for the details of mouse/touch event handling.

26.3 (*Drawing app*) Create an app that allows the user to select whether to draw a Line, Rectangle or Ellipse and to specify the shape’s Stroke and Fill colors. Then allow the user to draw

that shape by dragging the mouse (or dragging a finger across a touch screen). Use mouse/touch event handling (Section 25.3.3) to allow the user to specify the shape's location and size. For a `Line`, this is determined by where the *press* and *release* events occur—these are the `Line`'s starting and ending coordinates, respectively. The ending coordinate will change while the user is dragging. For a `Rectangle` or `Ellipse`, determine which *x*-coordinate and which *y*-coordinate represent the shape's top-left corner, then calculate the shape's width and height as the absolute value of the difference between the two *x*-coordinates and the two *y*-coordinates, respectively.

