WPF Graphics and Multimedia

25

Nowadays people's visual imagination is so much more sophisticated, so much more developed, particularly in young people, that now you can make an image which just slightly suggests something, they can make of it what they will.
—Robert Doisneau

In shape, it is perfectly elliptical. In texture, it is smooth and lustrous. In color, it ranges from pale alabaster to warm terra cotta.
—Sydney J Harris, "Tribute to an Egg"

Objectives

In this chapter you'll learn:

- To manipulate fonts.
- To draw basic WPF shapes.
- To use WPF brushes to customize the `Fill` or `Background` of an object.
- To use WPF transforms to reposition or reorient GUI elements.
- To completely customize the look of a control while maintaining its functionality.
- To animate the properties of a GUI element.
- To transform and animate 3-D objects.
- To use speech synthesis and recognition.

Outline

25.1 Introduction

This chapter overviews WPF's graphics and multimedia capabilities, including two-dimensional and three-dimensional shapes, fonts, transformations, animations, audio and video. WPF integrates drawing and animation features that were previously available only in special libraries (such as DirectX). 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 applications 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.

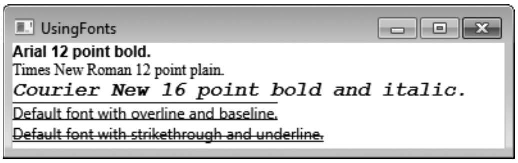
The basic 2-D shapes are `Lines`, `Rectangles` and `Ellipses`. WPF also has controls that can be used to create custom shapes or curves. `Brushes` can be used to fill an element with solid colors, complex patterns, gradients, images or videos, allowing for unique and interesting visual experiences. WPF's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

WPF also includes 3-D modeling and rendering capabilities. In addition, 2-D manipulations can be applied to 3-D objects as well. You can find more information on WPF in our WPF Resource Center at `www.deitel.com/wpf/`. The chapter ends with an introduction to speech synthesis and recognition.

25.2 Controlling Fonts

This section introduces how to control fonts by modifying the font properties of a **`TextBlock`** control in the XAML code. Figure 25.1 shows how to use `TextBlocks` and how to change the properties to control the appearance of the displayed text.

```
 1  <!-- Fig. 25.1: 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 property -->
10        <TextBlock FontFamily="Arial" FontSize="12" FontWeight="Bold">
11           Arial 12 point bold.</TextBlock>
12
13        <!-- if no font size is specified, default is 12 -->
14        <TextBlock FontFamily="Times New Roman">
15           Times New Roman 12 point plain.</TextBlock>
16
17        <!-- specifying a different font size and using FontStyle -->
18        <TextBlock FontFamily="Courier New" FontSize="16"
19           FontStyle="Italic" FontWeight="Bold">
20           Courier New 16 point bold and italic.
21        </TextBlock>
22
23        <!-- using Overline and Baseline TextDecorations -->
24        <TextBlock>
25           <TextBlock.TextDecorations>
26              <TextDecoration Location="OverLine" />
27              <TextDecoration Location="Baseline" />
28           </TextBlock.TextDecorations>
29           Default font with overline and baseline.
30        </TextBlock>
31
32        <!-- using Strikethrough and Underline TextDecorations -->
33        <TextBlock>
34           <TextBlock.TextDecorations>
35              <TextDecoration Location="Strikethrough" />
36              <TextDecoration Location="Underline" />
37           </TextBlock.TextDecorations>
38           Default font with strikethrough and underline.
39        </TextBlock>
40     </StackPanel>
41  </Window>
```



```
UsingFonts
Arial 12 point bold.
Times New Roman 12 point plain.
Courier New 16 point bold and italic.
Default font with overline and baseline.
Default font with strikethrough and underline.
```

Fig. 25.1 | Formatting fonts in XAML code.

The text that you want to display in the `TextBlock` is placed between the `TextBlock` tags. The **`FontFamily`** property defines the font of the displayed text. This property can be set to any font. Lines 10, 14 and 18 define the separate `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 for Windows Vista/Windows 7, is used (lines 24 and 33).

The **FontSize** property defines the text size measured in 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 10 and 18. In lines 14, 24 and 33, the FontSize is not defined so the default is used.

TextBlocks have various properties that can further modify the font. Lines 10 and 19 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—to define the thickness of the text. You can use the **FontStyle** property to make the text either Italic or Oblique—which is simply a more emphasized italic. Line 19 sets the FontStyle property to Italic.

You can also define **TextDecorations** for a TextBlock to draw a horizontal line through the text. **Overline** and **Baseline**—shown in the fourth TextBlock of Fig. 25.1—create lines above the text and at the base of the text, respectively (lines 26–27). **Strikethrough** and **Underline**—shown in the fifth TextBlock—create lines through the middle of the text and under the text, respectively (lines 35–36). 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.

25.3 Basic Shapes

WPF has several built-in shapes. The BasicShapes example (Fig. 25.2) shows you how to display Lines, Rectangles and Ellipses.

```
<!-- Fig. 25.2: MainWindow.xaml -->
<!-- Drawing basic shapes in XAML. -->
<Window x:Class="BasicShapes.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BasicShapes" Height="200" Width="500">
   <Canvas>
      <!-- Rectangle with fill but no stroke -->
      <Rectangle Canvas.Left="90" Canvas.Top="30" Width="150" Height="90"
         Fill="Blue" />

      <!-- Lines defined by starting points and ending points-->
      <Line X1="90" Y1="30" X2="110" Y2="40" Stroke="Black" />
      <Line X1="90" Y1="120" X2="110" Y2="130" Stroke="Black" />
      <Line X1="240" Y1="30" X2="260" Y2="40" Stroke="Black" />
      <Line X1="240" Y1="120" X2="260" Y2="130" Stroke="Black" />

      <!-- Rectangle with stroke but no fill -->
      <Rectangle Canvas.Left="110" Canvas.Top="40" Width="150"
         Height="90" Stroke="Black" />

      <!-- Ellipse with fill and no stroke -->
      <Ellipse Canvas.Left="280" Canvas.Top="75" Width="100" Height="50"
         Fill="Red" />
```

Fig. 25.2 | Drawing basic shapes in XAML. (Part 1 of 2.)

```
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>
```

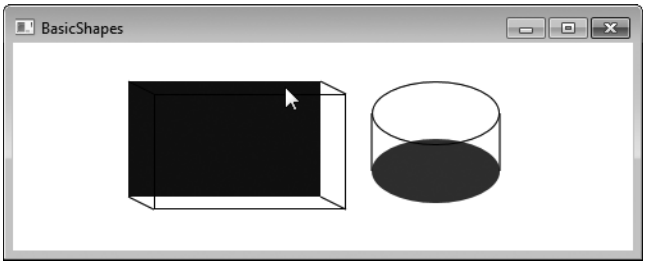


Fig. 25.2 | Drawing basic shapes in XAML. (Part 2 of 2.)

The first shape drawn uses the **`Rectangle`** object to create a filled rectangle in the window. Notice that the layout control is a `Canvas` allowing us to use coordinates to position the shapes. To specify the upper-left corner of the `Rectangle` at lines 9–10, we set the `Canvas.Left` and `Canvas.Top` properties to 90 and 30, respectively. We then set the `Width` and `Height` properties to 150 and 90, respectively, to specify the size. To define the `Rectangle`'s color, we use the **`Fill`** property (line 10). You can assign any `Color` or `Brush` to this property. `Rectangles` also have a **`Stroke`** property, which defines the color of the outline of the shape (line 20). If either the `Fill` or the `Stroke` is not specified, that property will be rendered transparently. For this reason, the blue `Rectangle` in the window has no outline, while the second `Rectangle` drawn has only an outline (with a transparent center). Shape objects have a **`StrokeThickness`** property which defines the thickness of the outline. The default value for `StrokeThickness` is 1 pixel.

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

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

25.4 Polygons and Polylines

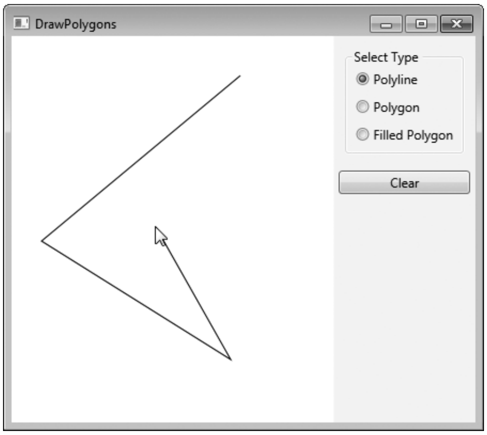
There are two shape controls for drawing multisided shapes—**`Polyline`** and **`Polygon`**. `Polyline` draws a series of connected lines defined by a set of points, while `Polygon` does the same but connects the start and end points to make a closed figure. The application `DrawPolygons` (Fig. 25.3) 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.

```
<!-- Fig. 25.3: MainWindow.xaml -->
<!-- Defining Polylines and Polygons in XAML. -->
<Window x:Class="DrawPolygons.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="DrawPolygons" Height="400" Width="450" Name="mainWindow">
   <Grid>
      <Grid.ColumnDefinitions>
         <ColumnDefinition />
         <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>

      <!-- Canvas contains two polygons and a polyline -->
      <!-- only the shape selected by the radio button is visible -->
      <Canvas Name="drawCanvas" Grid.Column="0" Background="White"
         MouseDown="drawCanvas_MouseDown">
         <Polyline Name="polyLine" Stroke="Black"
            Visibility="Collapsed" />
         <Polygon Name="polygon" Stroke="Black" Visibility="Collapsed" />
         <Polygon Name="filledPolygon" Fill="DarkBlue"
            Visibility="Collapsed" />
      </Canvas>

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

               <!-- unfilled Polygon option -->
               <RadioButton Name="polygonRadio" Margin="5"
                  Checked="polygonRadio_Checked">Polygon</RadioButton>

               <!-- filled Polygon option -->
               <RadioButton Name="filledPolygonRadio" Margin="5"
                  Checked="filledPolygonRadio_Checked">
                  Filled Polygon</RadioButton>
            </StackPanel>
         </GroupBox>

         <!-- Button clears the shape from the canvas -->
         <Button Name="clearButton" Click="clearButton_Click"
            Margin="5">Clear</Button>
```

Fig. 25.3 | Defining Polylines and Polygons in XAML. (Part 1 of 2.)

```
47          </StackPanel>
48       </Grid>
49    </Window>
```

a) Application with the Polyline option selected



b) Application with the Filled Polygon option selected

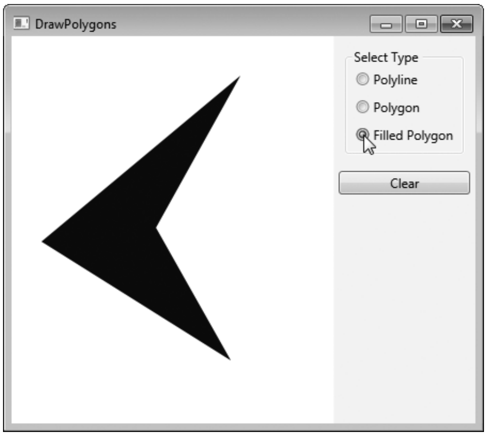


Fig. 25.3 | Defining Polylines and Polygons in XAML. (Part 2 of 2.)

The code 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 application is shown in Fig. 25.4.

```
 1   // Fig. 25.4: 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      {
```

Fig. 25.4 | Drawing `Polylines` and `Polygons`. (Part 1 of 3.)

```
      // stores the collection of points for the multisided shapes
      private PointCollection points = new PointCollection();

      // initialize the points of the shapes
      public MainWindow()
      {
         InitializeComponent();

         polyLine.Points = points; // assign Polyline points
         polygon.Points = points; // assign Polygon points
         filledPolygon.Points = points; // assign filled Polygon points
      } // end constructor

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

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

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

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

      // when the user selects the filled Polygon
      private void filledPolygonRadio_Checked( object sender,
         RoutedEventArgs e )
      {
         // filled Polygon is visible, the other two are not
         polyLine.Visibility = Visibility.Collapsed;
         polygon.Visibility = Visibility.Collapsed;
```

Fig. 25.4 | Drawing `Polylines` and `Polygons`. (Part 2 of 3.)

```
64             filledPolygon.Visibility = Visibility.Visible;
65          } // end method filledPolygonRadio_Checked
66       } // end class MainWindow
67    } // end namespace DrawPolygons
```

Fig. 25.4 | Drawing `Polylines` and `Polygons`. (Part 3 of 3.)

To allow the user to specify a variable number of points, line 12 declares a **`PointCollection`**, which is a collection that stores `Point` objects. This keeps track of each mouse-click location. The collection's **`Add`** method adds new points to the end of the collection. When the application 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.

25.5 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` application (Fig. 25.5) to see `Brushes` applied to `TextBlocks` and `Ellipses`.

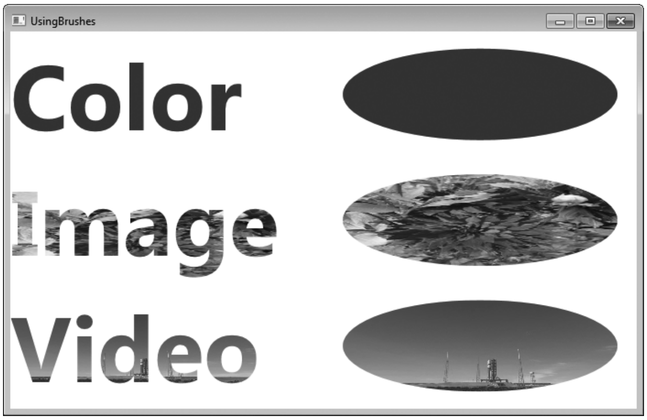
```
<!-- Fig. 25.5: MainWindow.xaml -->
<!-- Applying brushes to various XAML elements. -->
<Window x:Class="UsingBrushes.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="UsingBrushes" Height="450" Width="700">
   <Grid>
      <Grid.RowDefinitions>
         <RowDefinition />
         <RowDefinition />
         <RowDefinition />
      </Grid.RowDefinitions>

```

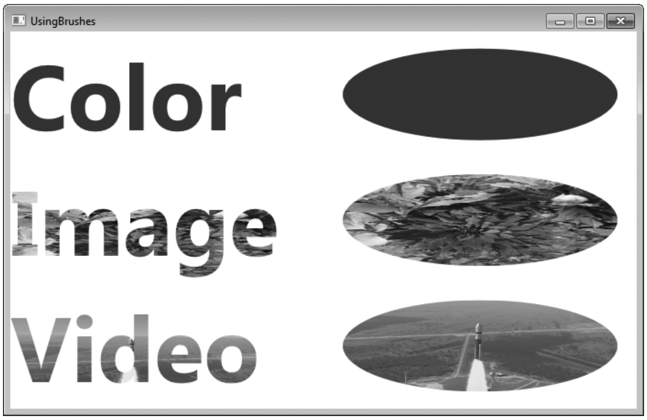
Fig. 25.5 | Applying brushes to various XAML elements. (Part 1 of 3.)

```
      <Grid.ColumnDefinitions>
         <ColumnDefinition />
         <ColumnDefinition />
      </Grid.ColumnDefinitions>

      <!-- TextBlock with a SolidColorBrush -->
      <TextBlock FontSize="100" FontWeight="999">
         <TextBlock.Foreground>
            <SolidColorBrush Color="#5F2CAE" />
         </TextBlock.Foreground>
         Color
      </TextBlock>

      <!-- Ellipse with a SolidColorBrush (just a Fill) -->
      <Ellipse Grid.Column="1" Height="100" Width="300" Fill="#5F2CAE" />

      <!-- TextBlock with an ImageBrush -->
      <TextBlock Grid.Row="1" FontSize="100" FontWeight="999">
         <TextBlock.Foreground>
            <!-- Flower image as an ImageBrush -->
            <ImageBrush ImageSource="flowers.jpg" />
         </TextBlock.Foreground>
         Image
      </TextBlock>

      <!-- Ellipse with an ImageBrush -->
      <Ellipse Grid.Row="1" Grid.Column="1" Height="100" Width="300">
         <Ellipse.Fill>
            <ImageBrush ImageSource="flowers.jpg" />
         </Ellipse.Fill>
      </Ellipse>

      <!-- TextBlock with a MediaElement as a VisualBrush -->
      <TextBlock Grid.Row="2" FontSize="100" FontWeight="999">
         <TextBlock.Foreground>
            <!-- VisualBrush with an embedded MediaElement-->
            <VisualBrush>
               <VisualBrush.Visual>
                  <MediaElement Source="nasa.wmv" />
               </VisualBrush.Visual>
            </VisualBrush>
         </TextBlock.Foreground>
         Video
      </TextBlock>

      <!-- Ellipse with a MediaElement as a VisualBrush -->
      <Ellipse Grid.Row="2" Grid.Column="1" Height="100" Width="300">
         <Ellipse.Fill>
            <VisualBrush>
               <VisualBrush.Visual>
                  <MediaElement Source="nasa.wmv" IsMuted="True"/>
               </VisualBrush.Visual>
            </VisualBrush>
```

Fig. 25.5 | Applying brushes to various XAML elements. (Part 2 of 3.)

```
            </Ellipse.Fill>
         </Ellipse>
      </Grid>
   </Window>
```

a)


b)


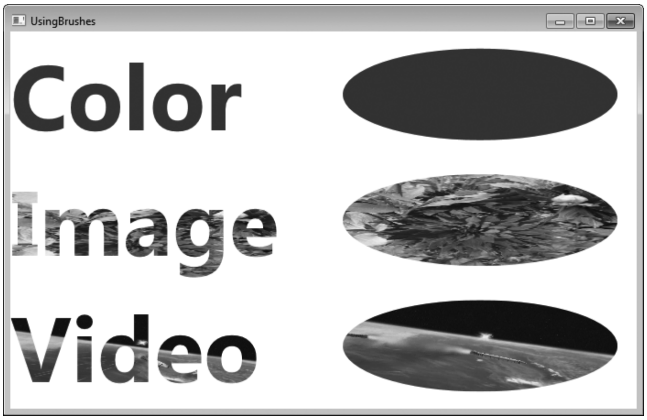
c)


Fig. 25.5 | Applying brushes to various XAML elements. (Part 3 of 3.)

ImageBrush

An **ImageBrush** paints an image into the property it is 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. Notice in lines 32–35 we apply the ImageBrush with its **ImageSource** set to the file we want to display (the image file must be included in the project). We can also assign the brush to the Fill of the Ellipse (lines 41–43) to display the image inside the shape.

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 application, you use the **MediaElement** control. Before using a video file in your application, add it to your Visual Studio project by first selecting the **Add Existing Item...** option in the **Project** menu. In the file dialog that appears, find and select the video you want to use. In the drop-down menu next to the **File Name** TextBox, you must change the selection to **All Files (*.*)** to be able to find your file. Once you have selected your file, click **Add**. 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** property of your MediaElement to the video. In the UsingBrushes application, we use nasa.wmv (line 52 and 64).

We use the **VisualBrush** element to display a video in the desired controls. Lines 50–54 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 48–55) and the Fill of the Ellipse (lines 61–67) to play the video inside the controls. Notice that the Fill of the third Row's elements is different in each screen capture in Fig. 25.5. This is because the video is playing inside the two elements.

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, which displays a gradient across the window. This was created by applying a LinearGradientBrush to a Rectangle's Fill. The gradient starts white and transitions linearly to black from left to right. You can set the RGBA values of the start and end colors to change the look of the gradient. The values entered in the TextBoxes must be in the range 0–255 for the application 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 application is shown in Fig. 25.6.

The GUI for this application contains a single Rectangle with a LinearGradientBrush applied to its Fill (lines 20–30). We define the **StartPoint** and **EndPoint** of the gradient in line 22. You must assign **logical points** to these properties, meaning the *x*- and

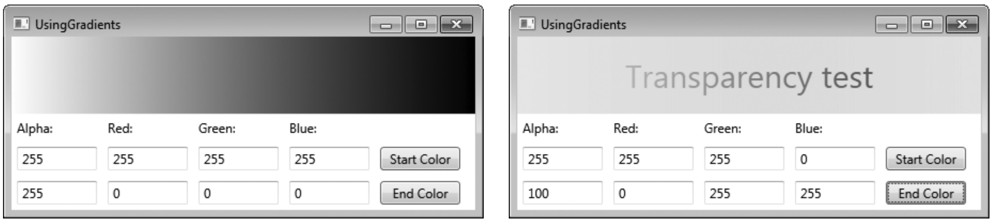
```
<!-- Fig. 25.6: MainWindow.xaml -->
<!-- Defining gradients in XAML. -->
<Window x:Class="UsingGradients.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="UsingGradients" Height="200" Width="450">
   <Grid>
      <Grid.RowDefinitions>
         <RowDefinition />
         <RowDefinition Height="Auto" />
         <RowDefinition Height="Auto" />
         <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <!-- TextBlock in the background to show transparency -->
      <TextBlock FontSize="30" HorizontalAlignment="Center"
         VerticalAlignment="Center">Transparency test</TextBlock>

      <!-- sample rectangle with linear gradient fill -->
      <Rectangle>
         <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
               <!-- gradient stop can define a color at any offset -->
               <GradientStop x:Name="startGradient" Offset="0.0"
                  Color="White" />
               <GradientStop x:Name="stopGradient" Offset="1.0"
                  Color="Black" />
            </LinearGradientBrush>
         </Rectangle.Fill>
      </Rectangle>

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

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

      <!-- GUI to select the color of second GradientStop -->
      <StackPanel Grid.Row="3" Orientation="Horizontal">
         <TextBox Name="toAlpha" Width="75" Margin="5">255</TextBox>
         <TextBox Name="toRed" Width="75" Margin="5">0</TextBox>
```

Fig. 25.6 | Defining gradients in XAML. (Part 1 of 2.)

```
54             <TextBox Name="toGreen" Width="75" Margin="5">0</TextBox>
55             <TextBox Name="toBlue" Width="75" Margin="5">0</TextBox>
56             <Button Name="toButton" Width="75" Margin="5"
57                Click="toButton_Click">End Color</Button>
58          </StackPanel>
59       </Grid>
60    </Window>
```

a) The application immediately after it is loaded

b) The application after changing the start and end colors



a)

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

b)

Alpha:	Red:	Green:	Blue:	
255	255	255	0	Start Color
100	0	255	255	End Color

Fig. 25.6 | Defining gradients in XAML. (Part 2 of 2.)

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.

A gradient is defined using `GradientStop` controls. 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 and 27 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 24 and 26), indicating that these colors appear at the start and end of the gradient (which were defined in line 22), respectively. The code in Fig. 25.7 allows the user to set the `Color`s of the two stops.

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

```
1  // Fig. 25.7: MainWindow.xaml.cs
2  // Customizing gradients.
3  using System;
4  using System.Windows;
5  using System.Windows.Media;
6
```

Fig. 25.7 | Customizing gradients. (Part 1 of 2.)

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

      // change the starting color of the gradient when the user clicks
      private void fromButton_Click( object sender, RoutedEventArgs e )
      {
         // change the color to use the ARGB values specified by user
         startGradient.Color = Color.FromArgb(
            Convert.ToByte( fromAlpha.Text ),
            Convert.ToByte( fromRed.Text ),
            Convert.ToByte( fromGreen.Text ),
            Convert.ToByte( fromBlue.Text ) );
      } // end method fromButton_Click

      // change the ending color of the gradient when the user clicks
      private void toButton_Click( object sender, RoutedEventArgs e )
      {
         // change the color to use the ARGB values specified by user
         stopGradient.Color = Color.FromArgb(
            Convert.ToByte( toAlpha.Text ),
            Convert.ToByte( toRed.Text ),
            Convert.ToByte( toGreen.Text ),
            Convert.ToByte( toBlue.Text ) );
      } // end method toButton_Click
   } // end class MainWindow
} // end namespace UsingGradients
```

Fig. 25.7 | Customizing gradients. (Part 2 of 2.)

25.6 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 25.7 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 25.8 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. 25.9.

```
<!-- Fig. 25.8: 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 application -->

      <!-- 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>
```

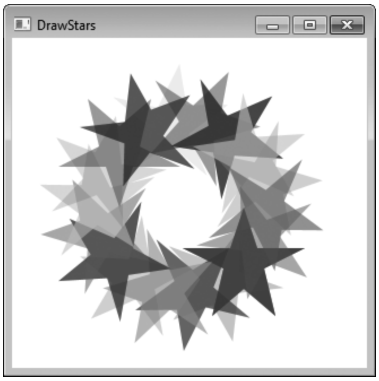


Fig. 25.8 | Defining a `Polygon` representing a star in XAML.

```
// Fig. 25.9: 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. 25.9 | 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. 25.9 | Applying transforms to a `Polygon`. (Part 2 of 2.)

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. 25.8. Each iteration of the loop duplicates `star` by creating a new `Polygon` with the same set of points (lines 22–23). 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 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.

25.7 WPF Customization: A Television GUI

In Chapter 24, 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 is 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. 25.10).

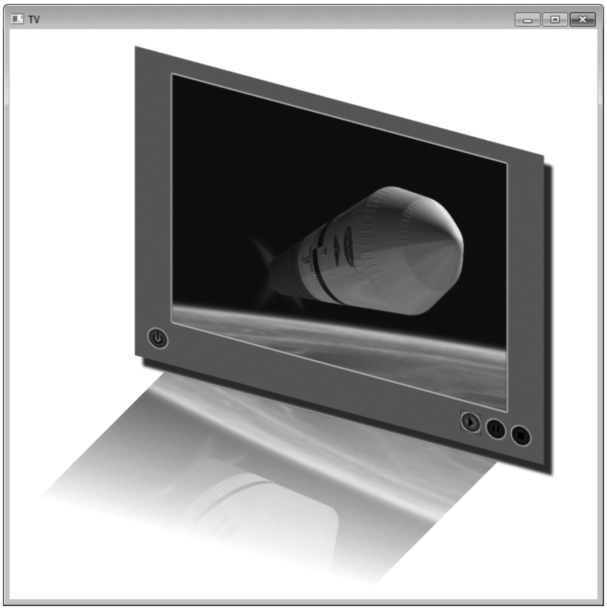


Fig. 25.10 | GUI representing a television.

The TV GUI may appear overwhelmingly complex, but it's actually just a basic WPF GUI built using controls with modified appearances. This example demonstrates the use of **WPF bitmap effects** to apply simple visual effects to some of the GUI elements. In addition, it introduces **opacity masks**, which can be used to hide parts of an element. Other than these two new concepts, the TV application is created using only the WPF elements and concepts that you've already learned. Figure 25.11 presents the XAML markup and a screen capture of the application when it first loads. The video used in this case study is a public-domain NASA video entitled *Animation: To the Moon* and can be downloaded from the NASA website (`www.nasa.gov/multimedia/hd/index.html`).

```
<!-- Fig. 25.11: MainWindow.xaml -->
<!-- TV GUI showing the versatility of WPF customization. -->
<Window x:Class="TV.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="TV" Height="720" Width="720">
   <Window.Resources>
      <!-- define template for play, pause and stop buttons -->
      <ControlTemplate x:Key="RadioButtonTemplate"
         TargetType="RadioButton">
         <Grid>
            <!-- create a circular border -->
            <Ellipse Width="25" Height="25" Fill="Silver" />
```

Fig. 25.11 | TV GUI showing the versatility of WPF customization (XAML). (Part 1 of 5.)

```

            <!-- create an "illuminated" background -->
            <Ellipse Name="backgroundEllipse" Width="22" Height="22">
               <Ellipse.Fill> <!-- enabled and unchecked state -->
                  <RadialGradientBrush> <!-- red "light" -->
                     <GradientStop Offset="0" Color="Red" />
                     <GradientStop Offset="1.25" Color="Black" />
                  </RadialGradientBrush>
               </Ellipse.Fill>
            </Ellipse>

            <!-- display button image -->
            <ContentPresenter Content="{TemplateBinding Content}" />
         </Grid>

         <!-- change appearance when state changes -->
         <ControlTemplate.Triggers>
            <!-- disabled state -->
            <Trigger Property="RadioButton.IsEnabled" Value="False">
               <Setter TargetName="backgroundEllipse" Property="Fill">
                  <Setter.Value>
                     <RadialGradientBrush> <!-- dim "light" -->
                        <GradientStop Offset="0" Color="LightGray" />
                        <GradientStop Offset="1.25" Color="Black" />
                     </RadialGradientBrush>
                  </Setter.Value>
               </Setter>
            </Trigger>

            <!-- checked state -->
            <Trigger Property="RadioButton.IsChecked" Value="True">
               <Setter TargetName="backgroundEllipse" Property="Fill">
                  <Setter.Value>
                     <RadialGradientBrush> <!-- green "light" -->
                        <GradientStop Offset="0" Color="LimeGreen" />
                        <GradientStop Offset="1.25" Color="Black" />
                     </RadialGradientBrush>
                  </Setter.Value>
               </Setter>
            </Trigger>
         </ControlTemplate.Triggers>
      </ControlTemplate>
   </Window.Resources>

   <!-- define the GUI -->
   <Canvas>
      <!-- define the "TV" -->
      <Border Canvas.Left="150" Height="370" Width="490"
         Canvas.Top="20" Background="DimGray">
         <Grid>
            <Grid.RowDefinitions>
               <RowDefinition />
```

Fig. 25.11 | TV GUI showing the versatility of WPF customization (XAML). (Part 2 of 5.)

```
            <RowDefinition Height="Auto" />
         </Grid.RowDefinitions>

         <!-- define the screen -->
         <Border Margin="0,20,0,10" Background="Black"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            BorderThickness="2" BorderBrush="Silver" CornerRadius="2">
            <MediaElement Height="300" Width="400"
               Name="videoMediaElement" Source="Video/future_nasa.wmv"
               LoadedBehavior="Manual" Stretch="Fill" />
         </Border>

         <!-- define the play, pause, and stop buttons -->
         <StackPanel Grid.Row="1" HorizontalAlignment="Right"
            Orientation="Horizontal">
            <RadioButton Name="playRadioButton" IsEnabled="False"
               Margin="0,0,5,15"
               Template="{StaticResource RadioButtonTemplate}"
               Checked="playRadioButton_Checked">
               <Image Height="20" Width="20"
                  Source="Images/play.png" Stretch="Uniform" />
            </RadioButton>
            <RadioButton Name="pauseRadioButton" IsEnabled="False"
               Margin="0,0,5,15"
               Template="{StaticResource RadioButtonTemplate}"
               Checked="pauseRadioButton_Checked">
               <Image Height="20" Width="20"
                  Source="Images/pause.png" Stretch="Uniform" />
            </RadioButton>
            <RadioButton Name="stopRadioButton" IsEnabled="False"
               Margin="0,0,15,15"
               Template="{StaticResource RadioButtonTemplate}"
               Checked="stopRadioButton_Checked">
               <Image Height="20" Width="20"
                  Source="Images/stop.png" Stretch="Uniform" />
            </RadioButton>
         </StackPanel>

         <!-- define the power button -->
         <CheckBox Name="powerCheckBox" Grid.Row="1" Width="25"
            Height="25" HorizontalAlignment="Left"
            Margin="15,0,0,15" Checked="powerCheckBox_Checked"
            Unchecked="powerCheckBox_Unchecked">
            <CheckBox.Template> <!-- set the template -->
               <ControlTemplate TargetType="CheckBox">
                  <Grid>
                     <!-- create a circular border -->
                     <Ellipse Width="25" Height="25"
                        Fill="Silver" />

                     <!-- create an "illuminated" background -->
                     <Ellipse Name="backgroundEllipse" Width="22"
                        Height="22">
```

Fig. 25.11 | TV GUI showing the versatility of WPF customization (XAML). (Part 3 of 5.)

```
                              <Ellipse.Fill> <!-- unchecked state -->
                                 <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">
```

Fig. 25.11 | TV GUI showing the versatility of WPF customization (XAML). (Part 4 of 5.)

```
               <Rectangle.Fill>
                  <!-- 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>
```

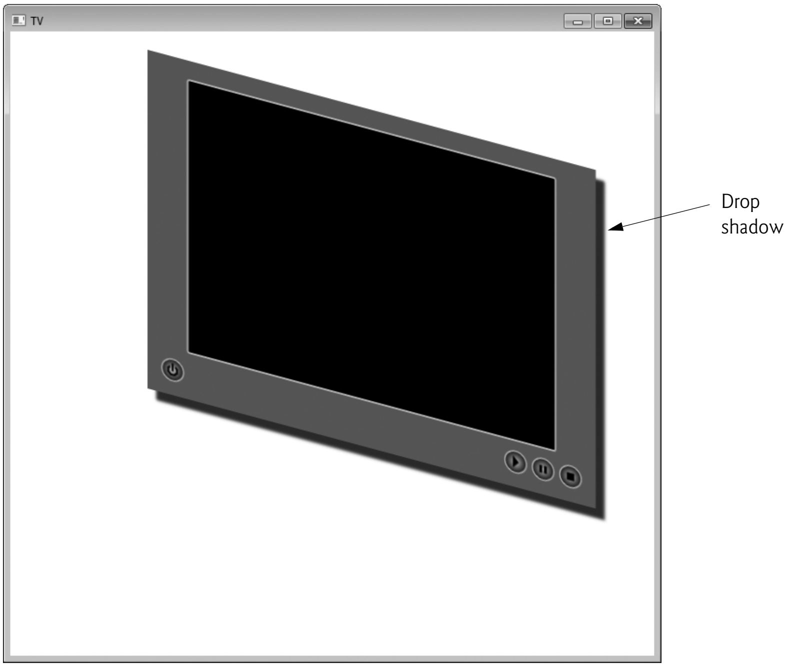


Fig. 25.11 | TV GUI showing the versatility of WPF customization (XAML). (Part 5 of 5.)

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. 25.11, 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` acts as a border (line 13). The smaller `Ellipse` 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 (lines 16–23). 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 application 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 bottom-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 is 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 is 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 vertically by 15 degrees clockwise from the *x*-axis and horizontally by 45 degrees clockwise from the *y*-axis (lines 193–195). 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. 25.12. 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. In Section 25.9, we present a 3-D object in perspective.

Examining the Code-Behind Class

Figure 25.13 presents the code-behind class that provides the functionality for the TV application. 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 these tasks are built into the `MediaElement` control. Playback can be modified programmatically only if the `LoadedBehavior` is `Manual` (line 75 in Fig. 25.11).

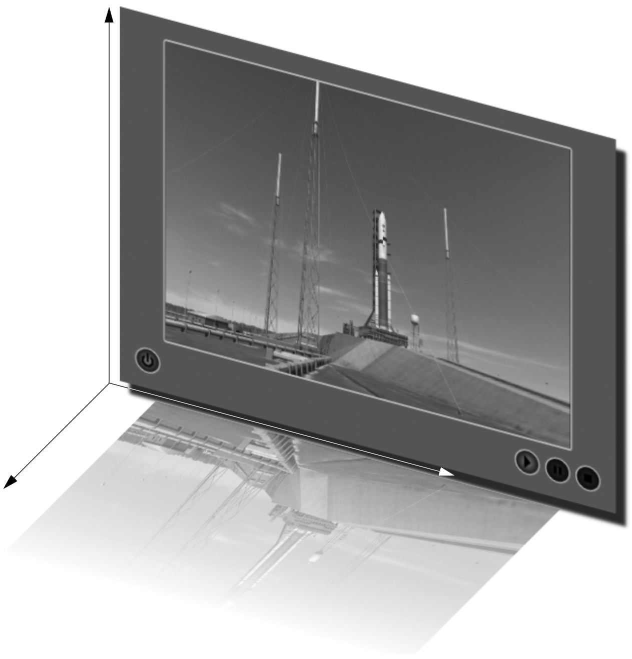


Fig. 25.12 | The effect of skewing the TV application's GUI components.

```
// Fig. 25.13: MainWindow.xaml.cs
// TV GUI showing the versatility of WPF customization (code-behind).
using System.Windows;

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

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

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

Fig. 25.13 | TV GUI showing the versatility of WPF customization (code-behind). (Part 1 of 2.)

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

         // turns "off" the TV
         private void powerCheckBox_Unchecked( object sender,
            RoutedEventArgs e )
         {
            // shut down the screen
            videoMediaElement.Close();

            // hide the reflection
            reflectionRectangle.Visibility = Visibility.Hidden;

            // disable the play, pause, and stop buttons
            playRadioButton.IsChecked = false;
            pauseRadioButton.IsChecked = false;
            stopRadioButton.IsChecked = false;
            playRadioButton.IsEnabled = false;
            pauseRadioButton.IsEnabled = false;
            stopRadioButton.IsEnabled = false;
         } // end method powerCheckBox_Unchecked

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

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

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

Fig. 25.13 | TV GUI showing the versatility of WPF customization (code-behind). (Part 2 of 2.)

25.8 Animations

An animation in WPF applications 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. 25.14) 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.

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 `Tar-getName` (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.

```
 1  <!-- Fig. 25.14: 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" Stretch="Fill"
14           Source="newfractal.wmv" /> <!-- Animated video -->
15
16        <StackPanel Grid.Column="1">
17           <!-- TextBox will contain the new Width for the video -->
18           <TextBlock Margin="5,0,0,0">Width:</TextBlock>
19           <TextBox Name="widthValue" Width="75" Margin="5">100</TextBox>
20
21           <!-- TextBox will contain the new Height for the video -->
22           <TextBlock Margin="5,0,0,0">Height:</TextBlock>
23           <TextBox Name="heightValue" Width="75" Margin="5">100</TextBox>
24
25           <!-- When clicked, rectangle animates to the input values -->
26           <Button Width="75" Margin="5">Animate
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
```

Fig. 25.14 | Animating the width and height of a video. (Part 1 of 2.)

```
                        <!-- Animates the Height -->
                        <DoubleAnimation Duration="0:0:2"
                           Storyboard.TargetProperty="Height"
                           To="{Binding ElementName=heightValue,
                           Path=Text}" />
                     </Storyboard>
                  </BeginStoryboard>
               </EventTrigger>
            </Button.Triggers>
         </Button>
      </StackPanel>
   </Grid>
</Window>
```

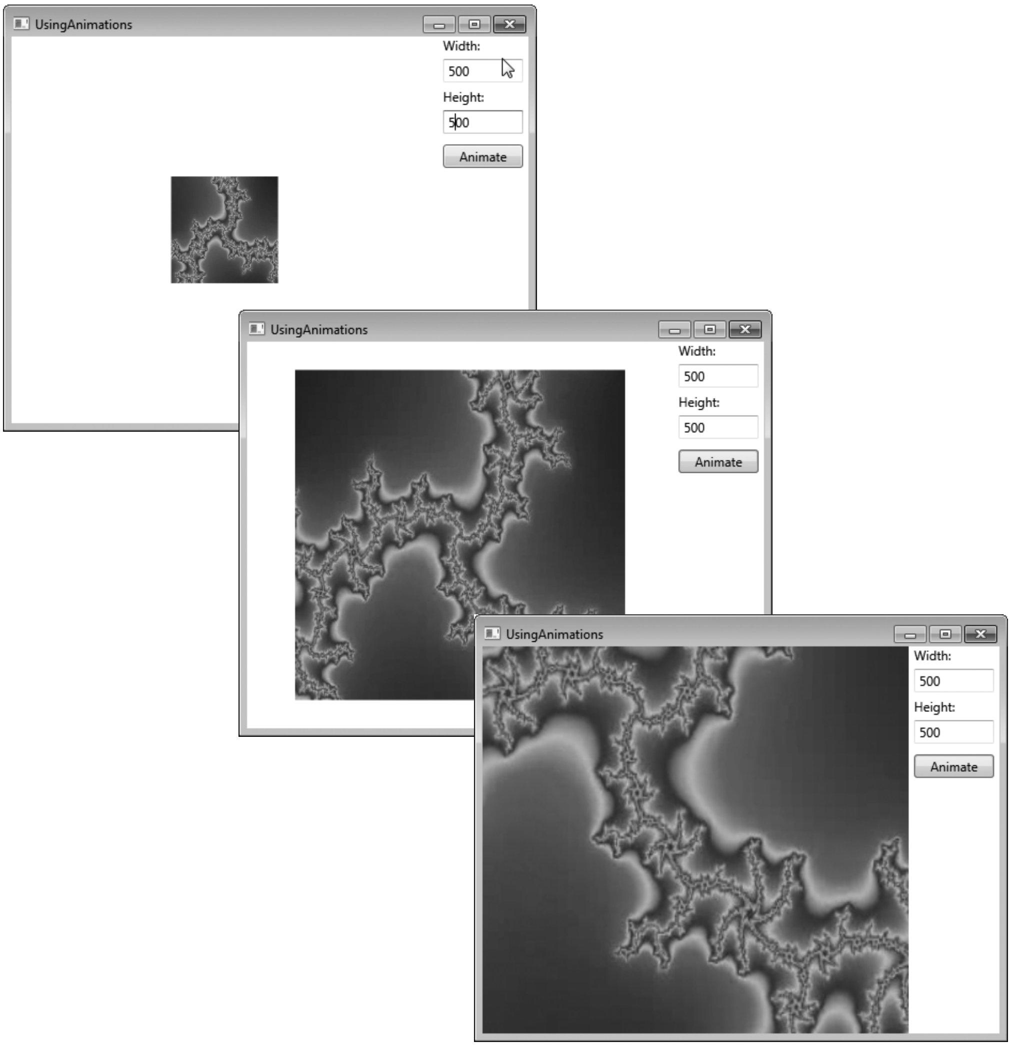


Fig. 25.14 | Animating the width and height of a video. (Part 2 of 2.)

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 is not always displayed at its original width and height. In line 13, 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**. A video's aspect ratio is the proportion between its width and height. Keeping this ratio at its original value ensures that the video does not look "stretched." `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.

25.9 (Optional) 3-D Objects and Transforms

WPF has substantial three-dimensional graphics capabilities. Once a 3-D shape is created, it can be manipulated using 3-D transforms and animations. This section requires an understanding of 3-D analytical geometry. Readers without a strong background in these geometric concepts can still enjoy this section. We overview several advanced WPF 3-D capabilities.

The next example creates a rotating pyramid. The user can change the axis of rotation to see all sides of the object. The XAML code for this application is shown in Fig. 25.15.

The first step in creating a 3-D object is to create a **`Viewport3D`** control (lines 29–76). The viewport represents the 2-D view the user sees when the application executes. This control defines a rendering surface for the content and contains content that represents the 3-D objects to render.

```
1  <!-- Fig. 25.15: MainWindow.xaml -->
2  <!-- Animating a 3-D object. -->
3  <Window x:Class="Application3D.MainWindow"
4      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
5      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
6      Title="Application3D" Height="300" Width="300">
7     <Grid>
```

Fig. 25.15 | Animating a 3-D object. (Part 1 of 3.)

```
      <Grid.RowDefinitions>
         <RowDefinition />
         <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <Grid.Triggers>
         <!-- when the window has loaded, begin the animation -->
         <EventTrigger RoutedEvent="Grid.Loaded">
            <BeginStoryboard>
               <Storyboard Storyboard.TargetName="rotation"
                  RepeatBehavior="Forever">

                  <!-- rotate the object 360 degrees -->
                  <DoubleAnimation Storyboard.TargetProperty="Angle"
                     To="360" Duration="0:0:3" />
               </Storyboard>
            </BeginStoryboard>
         </EventTrigger>
      </Grid.Triggers>

      <!-- viewport window for viewing the 3D object -->
      <Viewport3D>
         <Viewport3D.Camera>
            <!-- camera represents what user sees -->
            <PerspectiveCamera x:Name="camera" Position="6,0,1"
               LookDirection="-1,0,0" UpDirection="0,0,1" />
         </Viewport3D.Camera>

         <!-- defines the 3-D content in the viewport -->
         <ModelVisual3D>
            <ModelVisual3D.Content>
               <Model3DGroup>

                  <!-- two light sources to illuminate the objects-->
                  <DirectionalLight Color="White" Direction="-1,0,0" />
                  <DirectionalLight Color="White" Direction="0,0,-1" />

                  <GeometryModel3D>
                     <!-- rotate the geometry about the z-axis -->
                     <GeometryModel3D.Transform>
                        <RotateTransform3D>
                           <RotateTransform3D.Rotation>
                              <AxisAngleRotation3D x:Name="rotation"
                                 Angle="0" Axis="0,0,1" />
                           </RotateTransform3D.Rotation>
                        </RotateTransform3D>
                     </GeometryModel3D.Transform>

                     <!-- defines the pyramid -->
                     <GeometryModel3D.Geometry>
                        <MeshGeometry3D Positions="1,1,0 1,-1,0 -1,1,0
                           -1,-1,0 0,0,2" TriangleIndices="0,4,1 2,4,0
```

Fig. 25.15 | Animating a 3-D object. (Part 2 of 3.)

```
60                            3,4,2 3,1,4 2,0,1 3,2,1"
61                            TextureCoordinates="0,0 1,0 0,1 1,1 0,0" />
62                      </GeometryModel3D.Geometry>
63
64                      <!-- defines the surface of the object -->
65                      <GeometryModel3D.Material>
66                         <DiffuseMaterial>
67                            <DiffuseMaterial.Brush>
68                               <ImageBrush ImageSource="cover.png" />
69                            </DiffuseMaterial.Brush>
70                         </DiffuseMaterial>
71                      </GeometryModel3D.Material>
72                   </GeometryModel3D>
73                </Model3DGroup>
74             </ModelVisual3D.Content>
75          </ModelVisual3D>
76       </Viewport3D>
77
78       <!-- RadioButtons to change the axis of rotation -->
79       <GroupBox Grid.Row="1" Header="Axis of rotation">
80          <StackPanel Orientation="Horizontal"
81             HorizontalAlignment="Center">
82             <RadioButton Name="xRadio" Margin="5"
83                Checked="xRadio_Checked">x-axis</RadioButton>
84             <RadioButton Name="yRadio" Margin="5"
85                Checked="yRadio_Checked">y-axis</RadioButton>
86             <RadioButton Name="zRadio" Margin="5"
87                Checked="zRadio_Checked">z-axis</RadioButton>
88          </StackPanel>
89       </GroupBox>
90    </Grid>
91 </Window>
```

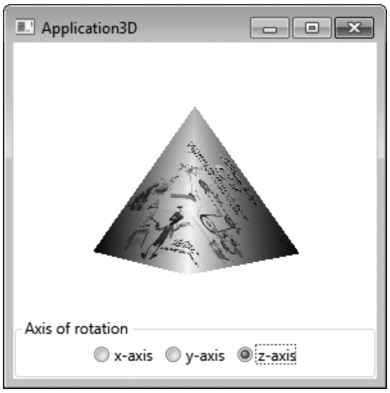


Fig. 25.15 | Animating a 3-D object. (Part 3 of 3.)

Create a **ModelVisual3D** object (lines 37–75) to define a 3-D object in a Viewport3D control. ModelVisual3D's **Content** property contains the shapes you wish to define in your space. To add multiple objects to the Content, embed them in a **Model3DGroup** element.

Creating the 3-D Object

3-D objects in WPF are modeled as sets of triangles, because you need a minimum of three points to make a flat surface. Every surface must be created or approximated as a collection of triangles. For this reason, shapes with flat surfaces (like cubes) are relatively simple to create, while curved surfaces (like spheres) are extremely complex. To make more complicated 3-D elements, you can use 3-D application development tools such as Electric Rain's ZAM 3D (`erain.com/products/zam3d/DefaultPDC.asp`), which generates the XAML markup.

Use the **`GeometryModel3D`** element to define a shape (lines 45–72). This control creates and textures your 3-D model. First we discuss this control's **`Geometry`** property (lines 57–62). Use the **`MeshGeometry3D`** control (lines 58–61) to specify the exact shape of the object you want to create in the `Geometry` property. To create the object, you need two collections—one is a set of points to represent the vertices, and the other uses those vertices to specify the triangles that define the shape. These collections are assigned to the **`Positions`** and **`TriangleIndices`** properties of `MeshGeometry3D`, respectively. The points that we assigned to the `Positions` attribute (lines 58–59) are shown in a 3-D space in Fig. 25.16. The view in the figure does not directly correspond to the view of the pyramid shown in the application. In the application, if you change the camera's `Position` (as you'll soon learn) to `"5,5,5"`, `LookDirection` to `"-1,-1,-1"` and `UpDirection` to `"0,1,0"`, you'll see the pyramid in the same orientation as in Fig. 25.16.

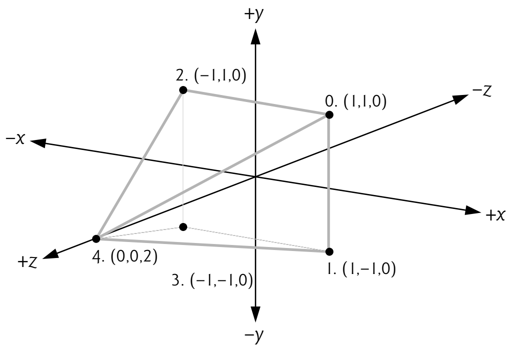


Fig. 25.16 | 3-D points making up a pyramid with a square base.

The points are labeled in the order they're defined in the `Positions` collection. For instance, the text `0. (1,1,0)` in the diagram refers to the first defined point, which has an index of 0 in the collection. Points in 3-D are defined with the notation "(*x*-coordinate, *y*-coordinate, *z*-coordinate)." With these points, we can define the triangles that we use to model the 3-D shape. The `TriangleIndices` property specifies the three corners of each individual triangle in the collection. The first element in the collection defined in line 59 is `(0,4,1)`. This indicates that we want to create a triangle with corners at points 0, 4 and 1 defined in the `Positions` collection. You can see this triangle in Fig. 25.16 (the front-most triangle in the picture). We can define all the sides of the pyramid by defining the rest of the triangles. Note also that while the pyramid has five flat surfaces, there are six triangles defined, because we need two triangles to create the pyramid's square base.

The order in which you define the triangle's corners dictates which side is considered the "front" versus the "back." Suppose you want to create a flat square in your viewport.

This can be done using two triangles, as shown in Fig. 25.17. If you want the surface facing toward you to be the "front," you must define the corners in counterclockwise order. So, to define the lower-left triangle, you need to define the triangle as `"0,1,3"`. The upper-right triangle needs to be `"1,2,3"`. By default, the "front" of the triangle is drawn with your defined `Material` (described in the next section) while the "back" is made transparent. Therefore, the order in which you define the triangle's vertices is significant.

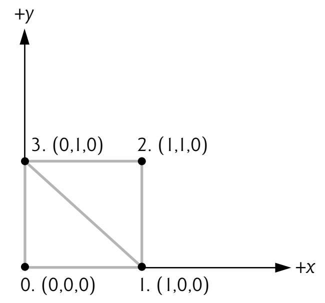


Fig. 25.17 | Defining two triangles to create a square in 3-D space.

Using a Brush on the Surface of a 3-D Object

By defining the **`Material`** property of the `GeometryModel3D`, we can specify what type of brush to use when painting each surface of the 3-D object. There are several different controls you can use to set the `Material` property. Each control gives a different "look" to the surface. Figure 25.18 describes the available controls.

3-D material controls	
`DiffuseMaterial`	Creates a "flat" surface that reflects light evenly in all directions.
`SpecularMaterial`	Creates a glossy-looking material. It creates a surface similar to that of metal or glass.
`EmissiveMaterial`	Creates a glowing surface that generates its own light (this light does not act as a light source for other objects).
`MaterialGroup`	Allows you to combine multiple materials, which are layered in the order they're added to the group.

Fig. 25.18 | 3-D material controls.

In the example, we use the **`DiffuseMaterial`** control. We can assign the brushes described in Section 25.5 to the material's **`Brush`** property to define how to paint the 3-D object's surface. We use an `ImageBrush` with `cover.png` as its source (line 68) to draw an image on the pyramid.

Notice in line 61 of Fig. 25.15 that we define the **`TextureCoordinates`** of the 3-D object. This property takes a `PointCollection` and determines how the `Material` is mapped onto the object's surfaces. If this property is not defined, the brush may not render correctly on the surface. The `TextureCoordinates` property defines which point on the image is mapped onto which vertex—an intersection of two or more edges—of the object.

Notice we assigned the `String` `"0,0 1,0 0,1 1,1 0,0"` to the `TextureCoordinates` property. This `String` is translated into a `PointCollection` containing `Points` `(0,0)`, `(1,0)`, `(0,1)`, `(1,1)` and `(0,0)`. These points are logical points—as described in Section 25.5—on the image. The five points defined here correspond directly to the five points defined in the `Positions` collection. The image's top-left corner `(0,0)`—defined first in `TextureCoordinates`—is mapped onto the first point in the `Positions` collection `(1,1,0)`. The bottom-right corner `(1,1)` of the image—defined fourth in `TextureCoordinates`—is mapped onto the fourth point in the `Positions` collection `(-1,-1,0)`. The other two corners are also mapped accordingly to the second and third points. This makes the image fully appear on the bottom surface of the pyramid, since that face is rectangular.

If a point is shared by two adjacent sides, you may not want to map the same point of the image to that particular vertex for the two different sides. To have complete control over how the brush is mapped onto the surfaces of the object, you may need to define a vertex more than once in the `Positions` collection.

Defining a Camera and a Light Source

The **`Camera`** property of `Viewport3D` (lines 30–34) defines a virtual camera for viewing the defined 3-D space. In this example, we use a **`PerspectiveCamera`** to define what the user sees. We must set the camera's **`Position`**, **`LookDirection`** and **`UpDirection`** (lines 32–33). The `Position` property requires a **`Point3D`** object which defines a 3-D point, while the `LookDirection` and `UpDirection` require **`Vector3D`** objects which define vectors in 3-D space. 3-D vectors are defined by an *x*-, a *y*- and a *z*-component (defined in that order in the XAML markup). For instance, the vector applied to the `UpDirection` is written as `"0,0,1"` (line 33) and represents a vector with an *x*- and *y*-component of 0, and a *z*-component of 1. This vector points in the positive direction of the *z*-axis.

The `Position` defines the location of the camera in the 3-D space. The `LookDirection` defines the direction in which the camera is pointed. The `UpDirection` defines the orientation of the camera by specifying the upward direction in the viewport. If the `UpDirection` in this example were set to `"0,0,-1"` then the pyramid would appear "upside-down" in the viewport.

Unlike 2-D objects, a 3-D object needs a virtual light source so the camera can actually "see" the 3-D scene. In the `Model3DGroup`, which groups all of the `ModelVisual3D`'s objects, we define two **`DirectionalLight`** objects (lines 42–43) to illuminate the pyramid. This control creates uniform rays of light pointing in the direction specified by the **`Direction`** property. This property receives a vector that points in the direction of the light. You can also define the **`Color`** property to change the light's color.

Animating the 3-D Object

As with 2-D animations, there is a set of 3-D animations that can be applied to 3-D objects. Lines 47–54 define the **`Transform`** property of the `GeometryModel3D` element that models a pyramid. We use the **`RotateTransform3D`** control to implement a rotation of the pyramid. We then use the **`AxisAngleRotation3D`** to strictly define the transform's rotation (lines 50–51). The **`Angle`** and **`Axis`** properties can be modified to customize the transform. The `Angle` is initially set to 0 (that is, not rotated) and the `Axis` of rotation to the *z*-axis, represented by the vector defined as `"0,0,1"` (line 51).

To animate the rotation, we created a `Storyboard` that modifies the `Angle` property of the `AxisAngleRotation3D` (lines 17–23). Notice we set the **`RepeatBehavior`** of the `Sto-`

ryboard to **Forever** (line 18), indicating that the animation repeats continuously while the window is open. This Storyboard is set to begin when the page loads (line 15).

The application contains RadioButtons at the bottom of the window that change the axis of rotation. The code-behind for this functionality appears in Fig. 25.19.

With each RadioButton's Checked event, we change the Axis of rotation to the appropriate Vector3D. We also change the Position of the PerspectiveCamera to give a better view of the rotating object. For instance, when xButton is clicked, we change the axis of rotation to the *x*-axis (line 19) and the camera's position to give a better view (line 20).

```
 1  // Fig. 25.19: MainWindow.xaml.cs
 2  // Changing the axis of rotation for a 3-D animation.
 3  using System.Windows;
 4  using System.Windows.Media.Media3D;
 5
 6  namespace Application3D
 7  {
 8     public partial class MainWindow : Window
 9     {
10        // constructor
11        public MainWindow()
12        {
13           InitializeComponent();
14        } // end constructor
15
16        // when user selects xRadio, set axis of rotation
17        private void xRadio_Checked( object sender, RoutedEventArgs e )
18        {
19           rotation.Axis = new Vector3D( 1, 0, 0 ); // set rotation axis
20           camera.Position = new Point3D( 6, 0, 0 ); // set camera position
21        } // end method xRadio_Checked
22
23        // when user selects yRadio, set axis of rotation
24        private void yRadio_Checked( object sender, RoutedEventArgs e )
25        {
26           rotation.Axis = new Vector3D( 0, 1, 0 ); // set rotation axis
27           camera.Position = new Point3D( 6, 0, 0 ); // set camera position
28        } // end method yRadio_Checked
29
30        // when user selects zRadio, set axis of rotation
31        private void zRadio_Checked( object sender, RoutedEventArgs e )
32        {
33           rotation.Axis = new Vector3D( 0, 0, 1 ); // set rotation axis
34           camera.Position = new Point3D( 6, 0, 1 ); // set camera position
35        } // end method zRadio_Checked
36     } // end class MainWindow
37  } // end namespace Application3D
```

Fig. 25.19 | Changing the axis of rotation for a 3-D animation.

25.10 Speech Synthesis and Speech Recognition

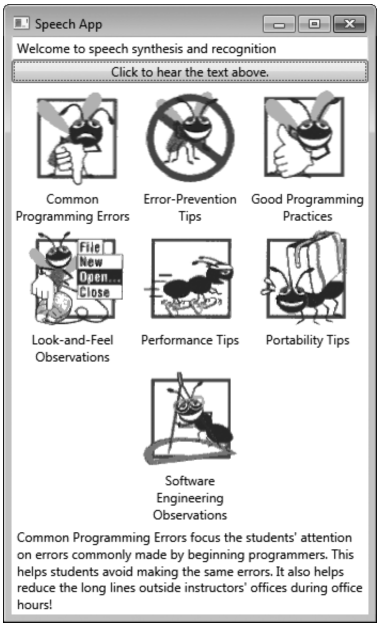
Speech-based interfaces make computers easier to use for people with disabilities (and others). **Speech synthesizers**, or **text-to-speech (TTS) systems**, read text out loud and are an ideal method for communicating information to sight-impaired individuals. **Speech recognizers**, or **speech-to-text (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 4.0 provides powerful tools for working with speech synthesis and recognition. The program shown in Figs. 25.20–25.21 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 application's GUI (Fig. 25.20) consists of a vertical `StackPanel` containing a `TextBox`, a `Button` and a series of horizontal `StackPanel`s containing `Image`s and `TextBlock`s that label those `Image`s.

```xml
<!-- Fig. 25.20: 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>
```

Fig. 25.20 | Text-To-Speech and Speech-To-Text (XAML). (Part 1 of 2.)

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


Speech App
Welcome to speech synthesis and recognition
Click to hear the text above.
Common Programming Errors
Error-Prevention Tips
Good Programming Practices
File
New
Open...
Close
Look-and-Feel Observations
Performance Tips
Portability Tips
Software Engineering Observations

Common Programming Errors focus the students' attention on errors commonly made by beginning programmers. This helps students avoid making the same errors. It also helps reduce the long lines outside instructors' offices during office hours!

Fig. 25.20 | Text-To-Speech and Speech-To-Text (XAML). (Part 2 of 2.)

Figure 25.21 provides the speech application'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. On the .NET tab of the **Add Reference** dialog, 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. 25.21: MainWindow.xaml.cs
 2  // Text-To-Speech and Speech-To-Text
 3  using System;
 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
```

Fig. 25.21 | Text-To-Speech and Speech-To-Text code-behind. (Part 1 of 3.)

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

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

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

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

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

```

Fig. 25.21 | Text-To-Speech and Speech-To-Text code-behind. (Part 2 of 3.)

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

      private void Image_MouseDown( object sender,
         System.Windows.Input.MouseButtonEventArgs e )
      {
         // use the image-to-description dictionary to get the
         // appropriate description for the clicked image
         displayString = imageDescriptions[ (Image) sender ];
         DisplaySpeak();
      } // end method Image_MouseDown

      // when the listener recognizes a phrase from the grammar, set the
      // display string and call DisplaySpeak
      void myGrammar_SpeechRecognized(
         object sender, RecognitionEventArgs e )
      {
         // Use the phrase-to-description dictionary to get the
         // appropriate description for the spoken phrase
         displayString = phraseDescriptions[ e.Result.Text ];

         // Use the dispatcher to call DisplaySpeak
         this.Dispatcher.BeginInvoke(
            new Action( DisplaySpeak ) );
      } // end method myGrammar_SpeechRecognized

      // Set the appropriate text block to the display string
      // and order the synthesizer to speak it
      void DisplaySpeak()
      {
         InfoBlock.Text = displayString;
         talker.SpeakAsync( displayString );
      } // end method DisplaySpeak
   } // end class MainWindow
} // end namespace SpeechApp
```

Fig. 25.21 | Text-To-Speech and Speech-To-Text code-behind. (Part 3 of 3.)

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 Images and strings, and the Dictionary phraseDescriptions contains pairs of strings and strings. 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 application 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. SpeechSynthesizers 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.

25.11 Wrap-Up

In this chapter you learned how to manipulate graphic elements in your WPF application. 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 application 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.

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 application 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 application 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.

You learned how to create a 3-D space using a Viewport3D control. You saw how to model 3-D objects as sets of triangles using the MeshGeometry3D control. The ImageBrush, which was previously applied to a 2-D object, was used to display a book-cover image on the surface of the 3-D pyramid using GeometryModel3D's mapping techniques. We discussed how to include lighting and camera objects in your Viewport3D to modify the view shown in the application. We showed how similar transforms and animations are in 2-D and 3-D.

Finally, we introduced the speech synthesis and speech recognition APIs. You learned how to make the computer speak text and how to receive voice input. You also learned how to create a Grammar of phrases that the user can speak to control the program. In Chapter 26, we discuss XML and LINQ to XML.

Summary

Section 25.1 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 25.2 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 can also define the `TextDecorations` property of a `TextBlock` to give the text any of four `TextDecorations`: `Underline`, `Baseline`, `Strikethrough` and `Overline`.

Section 25.3 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 25.4 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 25.5 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 is assigned to (such as a `Background`).
- A `VisualBrush` can display a fully customized video into the property it is assigned to.
- To use audio and video in a WPF application, 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 25.6 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 25.7 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 25.8 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 25.9 (Optional) 3-D Objects and Transforms

- WPF has substantial three-dimensional graphics capabilities which require an understanding of 3-D analytical geometry.
- The `Viewport3D` represents the 2-D view of the 3-D space the user sees.
- Create a `ModelVisual3D` object to define 3-D objects in a `Viewport3D` control.
- 3-D objects in WPF are modeled as sets of triangles.
- Use the `GeometryModel3D` element to define a shape.

- Use the MeshGeometry3D control to specify the exact shape of the object you want to create in the Geometry property of GeometryModel3D. The Positions and TriangleIndices properties of MeshGeometry3D are used to define the triangles modeling the shape.
- Corners of the triangles in the TriangleIndices property of MeshGeometry3D must be defined in counterclockwise order to be viewed.
- Specify a brush in a GeometryModel3D's Material property to paint the surface of the 3-D object.
- The TextureCoordinates property of MeshGeometry3D determines how a Material is mapped onto the 3-D object's surface. You may need to define a vertex more than once in the Positions collection to get proper mapping.
- The Camera property of Viewport3D defines a virtual camera for viewing the defined 3-D space.
- A 3-D object needs a virtual light source so the camera can actually "see" the 3-D scene.
- A DirectionalLight control creates uniform rays of light pointing in a specified direction.
- Use the AxisAngleRotation3D to strictly define the RotateTransform3D's rotation.
- The RepeatBehavior of a Storyboard or animation indicates whether it repeats when the animation is complete.

Section 25.10 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
Angle property of AxisAngleRotation3D control
aspect ratio
Axis property of AxisAngleRotation3D control
AxisAngleRotation3D control
Background property of TextBlock control
BlurEffect
Brush property of DiffuseMaterial control
Camera property of Viewport3D control
CenterX property of ScaleTransform control
CenterY property of ScaleTransform control
Clear method of class PointCollection
Color property of DirectionalLight control
Color property of GradientStop control
Content property of ModelVisual3D control
cropping
Dictionary class
DiffuseMaterial control
Direction property of DirectionalLight control
DirectionalLight control
Dispatcher

DoubleAnimation control
DropShadowEffect
Duration property of DoubleAnimation control
Ellipse control
EndPoint property of LinearGradientBrush control
Fill property of a shape control
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 control
Geometry property of GeometryModel3D control
GeometryModel3D control
gradient
GradientStop control
Grammar
ImageBrush control (WPF)
ImageSource property of ImageBrush control
Line control
LinearGradientBrush control
Location property of TextDecoration control
logical point
LookDirection property of PerspectiveCamera control
machine-independent pixel
Material property of GeometryModel3D control
MediaElement control
MeshGeometry3D control
Model3DGroup control
ModelVisual3D control
NextBytes method of class Random
None value for Stretch property
Offset property of GradientStop control
opacity mask
OpacityMask property of Rectangle control
orthographic projection
perspective projection
PerspectiveCamera control
Point3D class
PointCollection class
Points property of Polyline control
Polygon control
Polyline control
Position property of PerspectiveCamera control
Positions property of MeshGeometry3D control
RadialGradientBrush control
raster-based graphics
Rectangle control
RenderTransform property of a WPF UI element
RepeatBehavior property of Storyboard control
RotateTransform control
RotateTransform3D control
ScaleTransform control
ScaleX property of ScaleTransform control
ScaleY property of ScaleTransform control
SkewTransform control
Source property of MediaElement control
Speech recognizers
Speech synthesizers
SpeechRecognizer
SpeechSynthesizer
speech-to-text systems
StartPoint property of LinearGradientBrush control
Storyboard control
Stretch property of MediaElement control
Stroke property of a shape control
StrokeThickness property of a shape control
STT systems
System.Speech.Recognition
System.Speech.Synthesis
TargetName property of Storyboard control
TargetProperty property of Storyboard control
TextBlock control
TextDecoration control
TextDecorations property of TextBlock control
text-to-speech systems
TextureCoordinates property of MeshGeometry3D control
To property of DoubleAnimation control
transform
Transform property of ModelVisual3D control
TranslateTransform control
TriangleIndices property of MeshGeometry3D control
TTS systems
Uniform value for Stretch property
UniformToFill value for Stretch property
UpDirection property of PerspectiveCamera control
vector-based graphics
Vector3D class
Viewport3D control
Visibility property of a WPF UI element
Visual property of VisualBrush control
VisualBrush control
WPF bitmap effect

Self-Review Exercises

25.1 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 is 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 application.
j) A 3-D object can be manipulated much as a 2-D object would be manipulated.

25.2 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) ________ control 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 `Position` of a `PerspectiveCamera` is defined with a(n) ________ object while the `Direction` is defined with a(n) ________ object.
i) The three basic available shape controls are `Line`, ________ and ________.
j) A(n) ________ creates an opacity mapping from a brush and applies it to an element.
k) ________ 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

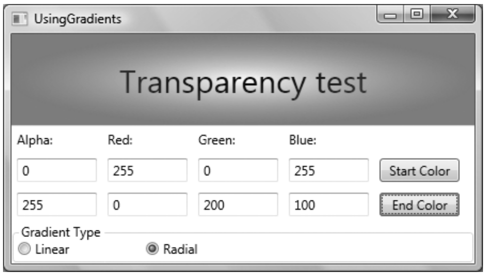
25.1 a) True. b) False. A `Line` is defined by a start point and an end point. c) False. When no `Fill` is defined, the object is transparent. d) True. e) False. A `Collapsed` object has a `Width` and `Height` of 0. f) True. g) False. You can define start and end points for the gradient to change the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

25.2 a) `TextBlock`. b) `TextDecoration`. c) `To`. d) `SkewTransform`, `ScaleTransform`. e) `Offset`. f) `TargetProperty`. g) `PointCollection`. h) `Point3D`, `Vector3D`. i) `Rectangle`, `Ellipse`. j) opacity mask. k) Logical.

Exercises

25.3 *(Enhanced **`UsingGradients`** application)* Modify the `UsingGradients` example from Section 25.5 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 `Radial-`

`Gradient` should have those start and stop colors as well when switched to. The GUI should appear as shown in Fig. 25.22.



Alpha:	Red:	Green:	Blue:	
0	255	0	255	Start Color
255	0	200	100	End Color

Gradient Type: Linear [✓ Radial]

Fig. 25.22 | `UsingGradients` example after `RadioButton` enhancement.

25.4 *(Enhanced `DrawStars` application)* Modify the `DrawStars` example in Section 25.6 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. 25.23, 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`.]

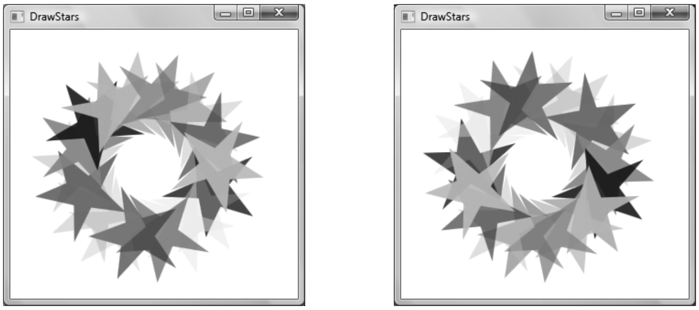


Fig. 25.23 | Animated `Polygons` rotating along the same circle.

25.5 *(Optional: Modified `Application3D` application)* Modify the `Application3D` example from Section 25.9 to create a 3-D die that can be animated along each axis. An image of all six faces of a die is in the `ExerciseImages` folder with this chapter's examples. You have to map the correct part of the image onto the sides of a cube. The application should look like the screen capture in Fig. 25.24.

25.6 *(`Image` reflector application)* Create an application that has the same GUI as shown in Fig. 25.25(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 25.25(b) shows one of the enlarged covers with a mouse over it.

25.7 *(Snake `PolyLine` application)* Create an application that creates a `Polyline` object that acts like a snake following your cursor around the window. The application, once the `Polyline` is created, should appear as shown in Fig. 25.26. 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).

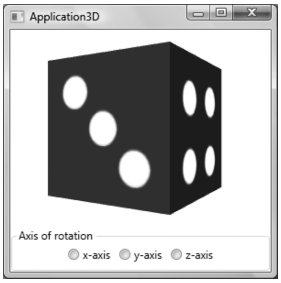


Fig. 25.24 | Creating a rotating 3-D die.

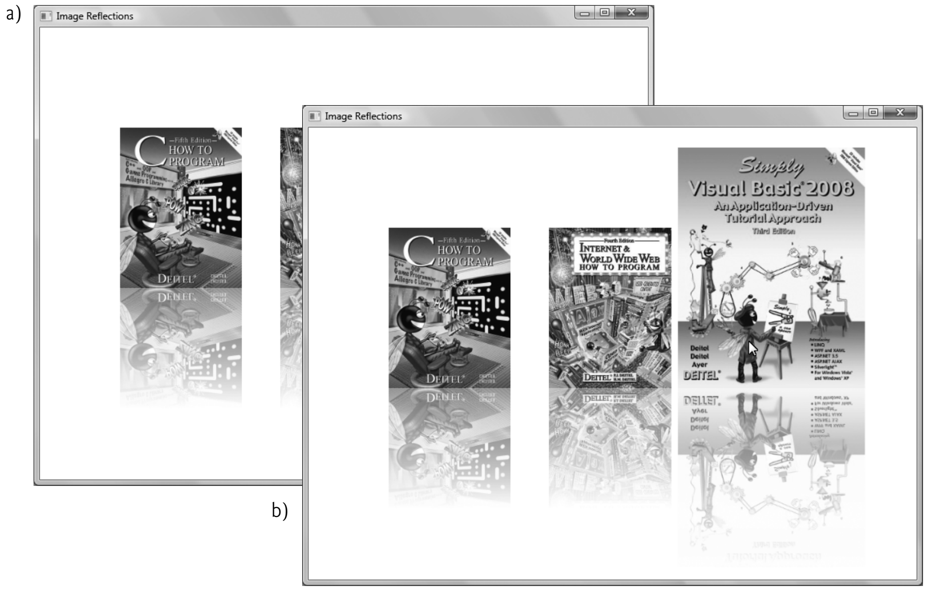


Fig. 25.25 | Cover images and their reflections. Each cover expands when the mouse hovers over it.

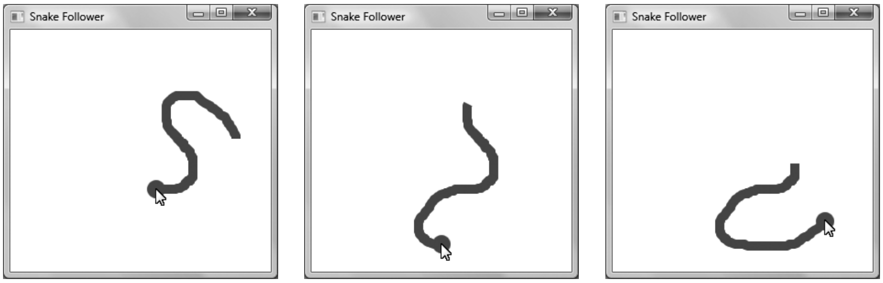


Fig. 25.26 | Snake follows the mouse cursor inside the window.

25.8 *(Project: Speech-Controlled Drawing Application)* Create a speech-controlled drawing application that is 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 application should support lines, rectangles and ellipses.