```
        </VisualBrush.Visual>
      </VisualBrush>
    </DiffuseMaterial.Brush>
  </DiffuseMaterial>
</GeometryModel3D.Material>
```

Figure 27-18 shows a snapshot of this example in action.

Figure 27-18. *Displaying video on several 3-D surfaces*

Interactivity and Animations

To get the full value out of your 3-D scene, you need to make it *dynamic*. In other words, you need to have some way to modify part of the scene, either automatically or in response to user actions. After all, if you don't need a dynamic 3-D scene, you'd be better off creating a 3-D image in your favorite illustration program and then exporting it as an ordinary XAML vector drawing. (Some 3-D modeling tools, such as ZAM 3D, provide exactly this option.)

In the following sections, you'll learn how to manipulate 3-D objects using transforms and how to add animation and move the camera. You'll also consider a separately released tool: a Trackball class that allows you to rotate a 3-D scene interactively. Finally, you'll learn how to perform hit testing in a 3-D scene and how to place interactive 2-D elements, such as buttons and text boxes, on a 3-D surface.

Transforms

As with 2-D content, the most powerful and flexible way to change an aspect of your 3-D scene is to use transforms. This is particularly the case with 3-D, as the classes you work with are relatively low-level. For example, if you want to scale a sphere, you need to construct the appropriate geometry and use the

ScaleTransform3D to animate it. If you had a 3-D sphere primitive to work with, this might not be necessary because you might be able to animate a higher-level property like Radius.

Transforms are obviously the answer to creating dynamic effects. However, before you can use transforms, you need to decide how you want to apply them. There are several possible approaches:

- Modify a transform that's applied to your Model3D. This allows you to change a single aspect of a single 3-D object. You can also use this technique on a Model3DGroup, as it derives from Model3D.
- Modify a transform that's applied to your ModelVisual3D. This allows you to change an entire scene.
- Modify a transform that's applied to your light. This allows you to change the lighting of your scene (for example, to create a "sunrise" effect).
- Modify a transform that's applied to your camera. This allows you to move the camera through your scene.

Transforms are so useful in 3-D drawing that it's a good idea to get into the habit of using a Transform3DGroup whenever you need a transform. That way, you can add additional transforms afterward without being forced to change your animation code. The ZAM 3D modeling program always adds a set of four placeholder transforms to every Model3DGroup, so that the object represented by that group can be manipulated in various ways:

```
<Model3DGroup.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetX="0" OffsetY="0" OffsetZ="0"/>
    <ScaleTransform3D ScaleX="1" ScaleY="1" ScaleZ="1"/>
    <RotateTransform3D>
      <RotateTransform3D.Rotation>
        <AxisAngleRotation3D Angle="0" Axis="0 1 0"/>
      </RotateTransform3D.Rotation>
    </RotateTransform3D>
    <TranslateTransform3D OffsetX="0" OffsetY="0" OffsetZ="0"/>
  </Transform3DGroup>
</Model3DGroup.Transform>
```

Notice that this set of transforms includes two TranslateTransform3D objects. That's because translating an object before it's been rotated produces a different result than translating it after it's been rotated, and you may want to use both effects.

Another handy technique is to name your transform objects in XAML using the x:Name attribute. Even though the transform objects don't have a name property, this creates a private member variable you can use to access them more easily without being forced to dig through a deep hierarchy of objects. This is particularly important because complex 3-D scenes often have multiple layers of Model3DGroup objects, as described earlier. Walking down this element tree from the top-level ModelVisual3D is awkward and error-prone.

Rotations

To get a taste of the ways you might use transforms, consider the following markup. It applies a RotateTransform3D, which allows you to rotate a 3-D object around an axis you specify. In this case, the axis of rotation is set to line up exactly with the Y axis in your coordinate system:

```
<ModelVisual3D.Transform>
  <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D x:Name="rotate" Axis="0 1 0" />
     </RotateTransform3D.Rotation>
  </RotateTransform3D>
</ModelVisual3D.Transform>
```

Using this named rotation, you can create a databound Slider that allows the user to spin the cube around its axis:

```
<Slider Grid.Row="1"  Minimum="0" Maximum="360" Orientation="Horizontal"
  Value="{Binding ElementName=rotate, Path=Angle}" ></Slider>
```

Just as easily, you can use this rotation in an animation. Here's an animation that spins a torus (a 3-D ring) simultaneously along two different axes. It all starts when a button is clicked:

```
<Button>
  <Button.Content>Rotate Torus</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <BeginStoryboard>
          <Storyboard RepeatBehavior="Forever">
            <DoubleAnimation Storyboard.TargetName="ring"
             Storyboard.TargetProperty="rotate1" To="360" Duration="0:0:2.5"/>
            <DoubleAnimation Storyboard.TargetName="ring"
             Storyboard.TargetProperty="rotate2" To="360" Duration="0:0:2.5"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

Figure 27-19 shows four snapshots of the torus in various stages of rotation.

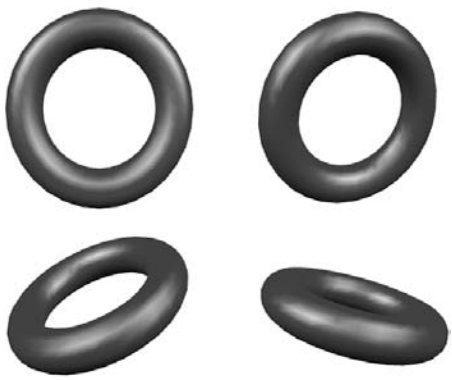

Figure 27-19. *A rotating 3-D shape*

A Fly Over

A common effect in 3-D scenes is to move the camera around the object. This task is conceptually quite easy in WPF. You simply need a TranslateTransform to move the camera. However, two considerations apply:

- Usually, you'll want to move the camera along a route rather than in a straight line from a start point to an end point. There are two ways to solve this challenge—you can use a path-based animation to follow a geometrically defined route, or you can use a key frame animation that defines several smaller segments.
- As the camera moves, it also needs to adjust the direction in which it's looking. You'll also need to animate the LookDirection property to keep focused on the object.

The following markup shows an animation that flies through the center of a torus, spins around its outer edge, and eventually drifts back to the starting point. To see this animation in action, check out the samples for this chapter:

```
<StackPanel Orientation="Horizontal">
  <Button>
    <Button.Content>Begin Fly-Through</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <BeginStoryboard>
          <Storyboard>
            <Point3DAnimationUsingKeyFrames
             Storyboard.TargetName="camera"
             Storyboard.TargetProperty="Position">
              <LinearPoint3DKeyFrame Value="0,0.2,-1" KeyTime="0:0:10"/>
              <LinearPoint3DKeyFrame Value="-0.5,0.2,-1" KeyTime="0:0:15"/>
              <LinearPoint3DKeyFrame Value="-0.5,0.5,0" KeyTime="0:0:20"/>
              <LinearPoint3DKeyFrame Value="0,0,2" KeyTime="0:0:23"/>
              </Point3DAnimationUsingKeyFrames>

            <Vector3DAnimationUsingKeyFrames
             Storyboard.TargetName="camera"
             Storyboard.TargetProperty="LookDirection">
              <LinearVector3DKeyFrame Value="-1,-1,-3" KeyTime="0:0:4"/>
              <LinearVector3DKeyFrame Value="-1,-1,3" KeyTime="0:0:10"/>
              <LinearVector3DKeyFrame Value="1,0,3" KeyTime="0:0:14"/>
              <LinearVector3DKeyFrame Value="0,0,-1" KeyTime="0:0:22"/>
            </Vector3DAnimationUsingKeyFrames>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Button.Triggers>
  </Button>
</StackPanel>
```

For a bit more fun, you can start both animations (the rotation shown earlier and the fly-over effect shown here), which will cause the camera to pass through the edge of the ring as it rotates. You can also animate the UpDirection property of the camera to wiggle it as it moves:

```
<Vector3DAnimation
  Storyboard.TargetName="camera" Storyboard.TargetProperty="UpDirection"
  From="0,0,-1" To="0,0.1,-1" Duration="0:0:0.5" AutoReverse="True"
  RepeatBehavior="Forever" />
```

3-D PERFORMANCE

Rendering a 3-D scene requires much more work than rendering a 2-D scene. When you animate a 3-D scene, WPF attempts to refresh the parts that have changed 60 times per second. Depending on the complexity of your scene, this can easily use up the memory resources on your video card, which will cause the frame rate to fall and the animation to become choppy.

There are a few basic techniques you can use to get better 3-D performance. Here are some strategies for tweaking the viewport to reduce the 3-D rendering overhead:

* If you don't need to crop content that extends beyond the bounds of your viewport, set Viewport3D.ClipToBounds to false.
* If you don't need to provide hit testing in your 3-D scene, set Viewport3D.IsHitTestVisible to false.
* If you don't mind lower quality—jagged edges on 3-D shapes—set the attached property RenderOptions.EdgeMode to Aliased on the Viewport3D.
* If your Viewport3D is larger than it needs to be, resize it to be smaller.

It's also important to ensure that your 3-D scene is as lightweight as possible. Here are a few critical tips for creating the most efficient meshes and models:

* Whenever possible, create a single complex mesh rather than several smaller meshes.
* If you need to use different materials for the same mesh, define the MeshGeometry object once (as a resource) and then reuse it to create multiple GeometryModel3D objects.
* Whenever possible, wrap a group of GeometryModel3D objects in a Model3DGroup, and place that group in a single ModelVisual3D object. Don't create a separate ModelVisual3D object for each GeometryModel3D.
* Don't define a back material (using GeometryModel3D.BackMaterial) unless the user will actually see the back of the object. Similarly, when defining meshes, consider leaving out triangles that won't be visible (for example, the bottom surface of a cube).
* Prefer solid brushes, gradient brushes, and the ImageBrush over the DrawingBrush and VisualBrush, both of which have more overhead. When using the DrawingBrush and VisualBrush to paint static content, you can cache the brush content to improve performance. To do so, use the attached property RenderOptions.CachingHint on the brush and set it to Cache.

If you keep these guidelines in mind, you'll be well on the way to ensuring the best possible 3-D drawing performance, and the highest possible frame rate for 3-D animation.

The Trackball

One of the most commonly requested behaviors in a 3-D scene is the ability to rotate an object using the mouse. One of the most common implementations is called a *virtual trackball*, and it's found in many 3-D graphics and 3-D design programs. Although WPF doesn't include a native implementation of a virtual trackball, the WPF 3-D team has released a free sample class that performs this function. This virtual trackball is a robust, extremely popular piece of code that finds its way into most of the 3-D demo applications that are provided by the WPF team.

The basic principle of the virtual trackball is that the user clicks somewhere on the 3-D object and drags it around an imaginary center axis. The amount of rotation depends on the distance the mouse is dragged. For example, if you click in the middle of the right side of a Viewport3D and drag the mouse to the left, the 3-D scene will appear to rotate around an imaginary vertical line. If you move the mouse all the way to the left side, the 3-D scene will be flipped 180 degrees to expose its back, as shown in Figure 27-20.

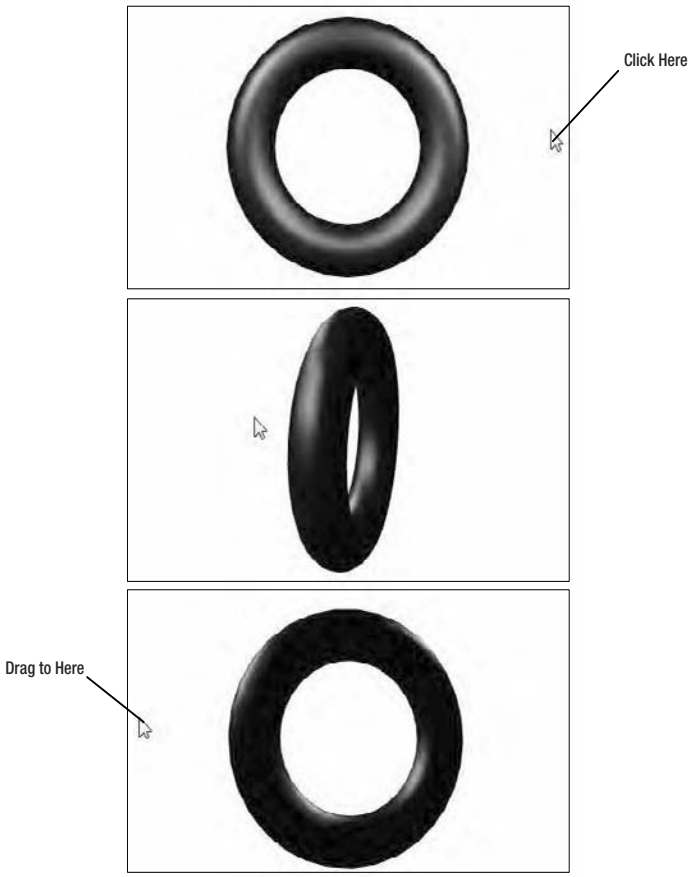

Figure 27-20. *Changing your viewpoint with the virtual trackball*

Although the virtual trackball appears to rotate the 3-D scene, it actually works by moving the camera. The camera always remains equally distant from the center point of the 3-D scene—essentially, the camera is moved along the contour of a big sphere that contains the entire scene. For a description of how the WPF virtual trackball works and the calculations that are involved, refer to `http://viewport3d.com/trackball.htm`. You can download the virtual trackball code with the 3-D tools projects described earlier at `http://www.codeplex.com/3DTools`.

■ **Note** Because the virtual trackball moves the camera, you shouldn't use it in conjunction with your own camera-moving animation. However, you can use it in conjunction with an animated 3-D scene (for example, a 3-D scene that contains a rotating torus like the one described earlier).

Using the virtual trackball is absurdly easy. All you need to do is wrap your Viewport3D in the TrackballDecorator class. The TrackballDecorator class is included with the 3-D tools project, so you'll need to begin by adding an XML alias for the namespace:

```
<Window xmlns:tools="clr-namespace:_3DTools;assembly=3DTools"    ...    >
```

Then you can easily add the TrackballDecorator to your markup:

```
<tools:TrackballDecorator>
  <Viewport3D>
    ...
  </Viewport3D>
</tools:TrackballDecorator>
```

Once you take this step, the virtual trackball functionality is automatically available—just click with the mouse and drag.

Hit Testing

Sooner or later, you'll want to create an interactive 3-D scene—one where the user can click 3-D shapes to perform different actions. The first step to implementing this design is *hit testing*, the process by which you intercept a mouse click and determine what region was clicked. Hit testing is easy in the 2-D world, but it's not quite as straightforward in a Viewport3D.

Fortunately, WPF provides sophisticated 3-D hit-testing support. You have three options for performing hit-testing in a 3-D scene:

- You can handle the mouse events of the viewport (such as MouseUp or MouseDown). Then you can call the VisualTreeHelper.HitTest() method to determine what object was hit. In the first version of WPF (released with .NET 3.0), this was the only possible approach.
- You can create your own 3-D control by deriving a custom class from the abstract UIElement3D class. This approach works, but it requires a lot of work. You need to implement all the UIElement-type plumbing on your own.

- You can replace one of your ModelVisual3D objects with a ModelUIElement3D object. The ModelUIElement3D class is derived from UIElement3D. It fuses the all-purpose 3-D model you've used so far with the interactive capabilities of a WPF element, including mouse handling.

To understand how 3-D hit testing works, it helps to consider a simple example. In the following section, you'll add hit testing to the familiar torus.

Hit Testing in the Viewport

To use the first approach to hit testing, you need to attach an event handler to one of the mouse events of the Viewport3D, such as MouseDown:

```
<Viewport3D MouseDown="viewport_MouseDown">
```

The MouseDown event handler uses hit-testing code at its simplest. It takes the current position of the mouse and returns a reference for the topmost ModelVisual3D that the point intercepts (if any):

```
private void viewport_MouseDown(object sender, MouseButtonEventArgs e)
{
    Viewport3D viewport = (Viewport3D)sender;
    Point location = e.GetPosition(viewport);
    HitTestResult hitResult = VisualTreeHelper.HitTest(viewport, location);

    if (hitResult != null && hitResult.VisualHit == ringVisual)
    {
        // The click hit the ring.
    }
}
```

Although this code works in simple examples, it's usually not sufficient. As you learned earlier, it's almost always better to combine multiple objects in the same ModelVisual3D. In many cases, all the objects in your entire scene will be placed in the same ModelVisual3D, so the hit doesn't provide enough information.

Fortunately, if the click intercepts a mesh, you can cast the HitTestResult to the more capable RayMeshGeometry3DHitTestResult object. You can find out which ModelVisual3D was hit using the RayMeshGeometry3DHitTestResult:

```
RayMeshGeometry3DHitTestResult meshHitResult =
  hitResult as RayMeshGeometry3DHitTestResult;
if (meshHitResult != null && meshHitResult.ModelHit == ringModel)
{
    // Hit the ring.
}
```

Or for even more fine-grained hit testing, you can use the MeshHit property to determine which specific mesh was hit. In the following example, the code determines whether the mesh representing the torus was hit. If it has been hit, the code creates and starts a new animation that rotates the torus. Here's the trick—the rotation axis is set so that it runs through the center of the torus, perpendicular to an imaginary line that connects the center of the torus to the location where the mouse was clicked. The effect makes it appear that the torus has been "hit" and is rebounding away from the click by twisting slightly away from the foreground and in the opposite direction.

Here's the code that implements that effect:

```
private void viewport_MouseDown(object sender, MouseButtonEventArgs e)
{
    Viewport3D viewport = (Viewport3D)sender;
    Point location = e.GetPosition(viewport);
    HitTestResult hitResult = VisualTreeHelper.HitTest(viewport, location);
    RayMeshGeometry3DHitTestResult meshHitResult =
      hitResult as RayMeshGeometry3DHitTestResult;

    if (meshHitResult != null && meshHitResult.MeshHit == ringMesh)
    {
       // Set the axis of rotation.
        axisRotation.Axis = new Vector3D(
          -meshHitResult.PointHit.Y, meshHitResult.PointHit.X, 0);

        // Start the animation.
        DoubleAnimation animation = new DoubleAnimation();
        animation.To = 40;
        animation.DecelerationRatio = 1;
        animation.Duration = TimeSpan.FromSeconds(0.15);
        animation.AutoReverse = true;
        axisRotation.BeginAnimation(AxisAngleRotation3D.AngleProperty, animation);
    }
}
```

This approach to hit testing works perfectly well. However, if you have a scene with a large number of 3-D objects and the interaction you require with these objects is straightforward (for example, you have a dozen buttons), this approach to hit testing makes for more work than necessary. In this situation, you're better off using the ModelUIElement3D class, which is introduced in the next section.

The ModelUIElement3D

The ModelUIElement3D is a type of Visual3D. Like all the Visual3D objects, it can be placed in a Viewport3D container.

Figure 27-21 shows the inheritance hierarchy for all the classes that derive from Visual3D. The three key classes that derive from Visual3D are ModelVisual3D (which you've used up to this point), UIElement3D (which defines the 3-D equivalent of the WPF element), and Viewport2DVisual3D (which allows you to place 2-D content in a 3-D scene, as described in the section "2-D Elements on 3-D Surfaces" later in this chapter).

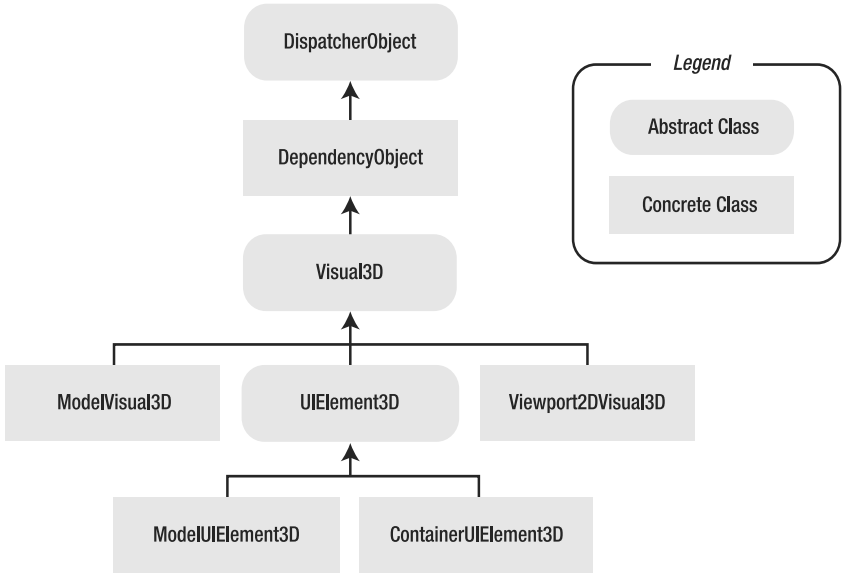

Figure 27-21. *The 3-D visual classes*

The UIElement3D class plays an analogous role to the UIElement class in the 2-D world, by adding support for mouse, keyboard, and stylus events, along with focus tracking. However, UIElement3D doesn't support any sort of layout system. The UIElement3D class, its descendants, and the Viewport2DVisual3D class are all new in WPF 3.5.

Although you can create a custom 3-D element by deriving from UIElement3D, it's far easier to use the ready-made classes that derive from UIElement3D: ModelUIElement3D and ContainerUIElement3D.

Using a ModelUIElement3D is not much different from using the ModelVisual3D class with which you're already familiar. The ModelUIElement3D class supports transforms (through the Transform property) and allows you to define its shape with a GeometryModel3D object (by setting the Model property, not the Content property as you do with ModelVisual3D).

Hit Testing with the ModelUIElement3D

Right now, the torus consists of a single ModelVisual3D, which contains a Model3DGroup. This group includes the torus geometry and the light sources that illuminate it. To change the torus example so that it uses the ModelUIElement3D, you simply need to replace the ModelVisual3D that represents the torus with a ModelUIElement3D:

```
<Viewport3D x:Name="viewport">
  <Viewport3D.Camera>...</Viewport3D.Camera>

  <ModelUIElement3D>
    <ModelUIElement3D.Model>
      <Model3DGroup>...<Model3DGroup>
    </ModelUIElement3D.Model>
  </ModelUIElement3D>
```

```
</Viewport3D>
```

Now you can perform hit testing directly with the ModelUIElement3D:

```
<ModelUIElement3D MouseDown="ringVisual_MouseDown">
```

The difference between this example and the previous one is that now the MouseDown event will fire only when the ring is clicked (rather than every time a point inside the viewport is clicked). However, the event-handling code still needs a bit of tweaking to get the result you want in this example.

The MouseDown event provides a standard MouseButtonEventArgs object to the event handler. This object provides the standard mouse event details, such as the exact time the event occurred, the state of the mouse buttons, and a GetPosition() method that allows you to determine the clicked coordinates relative to any element that implements IInputElement (such as the Viewport3D or the MouseUIElement3D). In many cases, these 2-D coordinates are exactly what you need. (For example, they are a requirement if you're using 2-D content on a 3-D surface, as described in the next section. In this case, any time you move, resize, or create elements, you're positioning them in 2-D space, which is then mapped to a 3-D surface based on a preexisting set of texture coordinates.)

However, in the current example it's important to get the 3-D coordinates on the torus mesh so that the appropriate animation can be created. That means you still need to use the VisualTreeHelper.HitTest() method, as shown here:

```
private void ringVisual_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Get the 2-D coordinates relative to the viewport.
    Point location = e.GetPosition(viewport);

    // Get the 3-D coordinates relative to the mesh.
    RayMeshGeometry3DHitTestResult meshHitResult =
      (RayMeshGeometry3DHitTestResult)VisualTreeHelper.HitTest(
        viewport, location);

   // Create the animation.
    axisRotation.Axis = new Vector3D(
      -meshHitResult.PointHit.Y, meshHitResult.PointHit.X, 0);
    DoubleAnimation animation = new DoubleAnimation();
    animation.To = 40;
    animation.DecelerationRatio = 1;
    animation.Duration = TimeSpan.FromSeconds(0.15);
    animation.AutoReverse = true;
    axisRotation.BeginAnimation(AxisAngleRotation3D.AngleProperty, animation);
}
```

Using this sort of realistic 3-D behavior, you could create a true 3-D "control," such as a button that deforms when you click it.

If you simply want to react to clicks on a 3-D object and you don't need to perform calculations that involve the mesh, you won't need to use the VisualTreeHelper at all. The fact that the MouseDown event fired tells you that the torus was clicked.

■ **Tip** In most cases, the ModelUIElement3D provides a simpler approach to hit testing than using the mouse events of the viewport. If you simply want to detect when a given shape is clicked (for example, you have a 3-D shape that represents a button and triggers an action), the ModelUIElement3D class is perfect. On the other hand, if you want to perform more complex calculations with the clicked coordinates or examine *all* the shapes that exist at a clicked location (not just the topmost one), you'll need more sophisticated hit testing code, and you'll probably want to respond to the mouse events of the viewport.

The ContainerUIElement3D

The ModelUIElement3D class is intended to represent a single control-like object. If you want to place more than one ModelUIElement3D in a 3-D scene and allow the user to interact with them independently, you need to create ModelUIElement3D objects and wrap them in a single ContainerUIElement3D. You can then add that ContainerUIElement3D to the viewport.

The ContainerUIElement3D has one other advantage. It supports any combination of objects that derive from Visual3D. That means it can hold ordinary ModelVisual3D objects, interactive ModelUIElement3D objects, and Viewport2DVisual3D objects, which represent 2-D elements that have been placed in 3-D space. You'll learn more about this trick in the next section.

2-D Elements on 3-D Surfaces

As you learned earlier in this chapter, you can use texture mapping to place 2-D brush content on a 3-D surface. You can use this to place images or videos in a 3-D scene. Using a VisualBrush, you can even take the visual appearance of an ordinary WPF element (such as a button), and place it in your 3-D scene.

However, the VisualBrush is inherently limited. As you already know, the VisualBrush can copy the visual appearance of an element, but it doesn't actually duplicate the element. If you use the VisualBrush to place the visual for a button in a 3-D scene, you'll end up with a 3-D picture of a button. In other words, you won't be able to click it.

The solution to this problem is the Viewport2DVisual3D class. The Viewport2DVisual3D class wraps another element and maps it to a 3-D surface using texture mapping. You can place the Viewport2DVisual3D directly in a Viewport3D, alongside other Visual3D objects (such as ModelVisual3D objects and ModelUIElement3D objects). However, the element inside the Viewport2DVisual3D retains its interactivity and has all the WPF features you're accustomed to, including layout, styling, templates, mouse events, drag-and-drop, and so on.

Figure 27-22 shows an example. A StackPanel containing a TextBlock, Button, and TextBox is placed on one of the faces of a 3-D cube. The user is in the process of typing text into the TextBox, and you can see the I-beam cursor that shows the insertion point.

Figure 27-22. *Interactive WPF elements in 3-D*

In your Viewport3D, you can place all the usual ModelVisual3D objects. In the example shown in Figure 27-22, there's a ModelVisual3D for the cube. To place your 2-D element content in the scene, you use a Viewport2DVisual3D object instead. The Viewport2DVisual3D class provides the properties listed in Table 27-5.

Table 27-5. *Properties of the InteractiveVisual3D*

Name	Description
Geometry	The mesh that defines the 3-D surface.
Visual	The 2-D element that will be placed on the 3-D surface. You can use only a single element, but it's perfectly legitimate to use a container panel to wrap multiple elements together. The example in Figure 27-22 uses a Border that contains a StackPanel with three child elements.
Material	The material that will be used to render the 2-D content. Usually, you'll use a DiffuseMaterial. You must set the attached Viewport2DVisual3D.IsVisualHostMaterial on the DiffuseMaterial to true so that the material is able to show element content.
Transform	A Transform3D or Transform3DGroup that determines how your mesh should be altered (rotated, scaled, skewed, and so on).

Using the 2-D on 3-D technique is relatively straightforward, provided you're already familiar with texture mapping (as described in the "Texture Mapping" section earlier in this chapter). Here's the markup that creates the WPF elements shown in Figure 27-22:

```
<Viewport2DVisual3D>
  <Viewport2DVisual3D.Geometry>
    <MeshGeometry3D
      Positions="0,0,0 0,0,10 0,10,0 0,10,10"
      TriangleIndices="0,1,2 2,1,3"
      TextureCoordinates="0,1 1,1 0,0 1,0"
    />
  </Viewport2DVisual3D.Geometry>

  <Viewport2DVisual3D.Material>
    <DiffuseMaterial Viewport2DVisual3D.IsVisualHostMaterial="True" />
  </Viewport2DVisual3D.Material>

  <Viewport2DVisual3D.Visual>
    <Border BorderBrush="Yellow" BorderThickness="1">
      <StackPanel Margin="10">
        <TextBlock Margin="3">This is 2D content on a 3D surface.</TextBlock>
        <Button Margin="3">Click Me</Button>
        <TextBox Margin="3">[Enter Text Here]</TextBox>
      </StackPanel>
    </Border>
  </Viewport2DVisual3D.Visual>

  <Viewport2DVisual3D.Transform>
    <RotateTransform3D>
      <RotateTransform3D.Rotation>
        <AxisAngleRotation3D
          Angle="{Binding ElementName=sliderRotate, Path=Value}"
          Axis="0 1 0" />
      </RotateTransform3D.Rotation>
    </RotateTransform3D>
  </Viewport2DVisual3D.Transform>
</Viewport2DVisual3D>
```

In this example, the Viewport2DVisual3D.Geometry property supplies a mesh that mirrors a single face of the cube. The TextureCoordinates of the mesh define how the 2-D content (the Border that wraps the StackPanel) should be mapped to the 3-D surface (the cube face). The texture mapping that you use with the Viewport2DVisual3D works in the same way as the texture mapping you used earlier with the ImageBrush and VisualBrush.

■ **Note** When defining the TextureCoordinates, it's important to make sure you have the element facing the camera. WPF does not render anything for the back surface of Viewport2DVisual3D, so if you flip it around and stare at its back, the element will disappear. (If this isn't the result you want, you can use another Viewport2DVisual3D to create content for the back side.)

This example also uses a RotateTransform3D to allow the user to turn the cube around using a slider underneath the Viewport3D. The ModelVisual3D that represents the cube includes the same RotateTransform3D, so the cube and 2-D element content move together.

Currently, this example doesn't use any event handling in the Viewport2DVisual3D content. However, it's easy enough to add an event handler:

```
<Button Margin="3" Click="cmd_Click">Click Me</Button>
```

WPF handles mouse events in a clever way. It uses texture mapping to translate the virtual 3-D coordinates (where the mouse is) to ordinary, non-texture-mapped 2-D coordinates. From the element's point of view, the mouse events are exactly the same in the 3-D world as they are in the 2-D world. This is part of the magic that holds the solution together.

■ **Tip** For a more elaborate example of 2-D content on a 3-D surface, refer to `http://tinyurl.com/3cnfxx`. You'll find a spinning globe example that lets you plant markers (with descriptive text) at arbitrary locations. All the content in this example consists of 2-D elements that are mapped to 3-D space.

The Last Word

3-D support is one of the gems of the WPF platform. Previous high-level development toolkits, such as Windows Forms, have avoided 3-D support altogether, leaving it to hard-core DirectX junkies. In fact, the most impressive part of WPF's 3-D features is their ease of use. Although it's possible to create complex code that creates and modifies 3-D meshes using intense math, it's just as possible to export 3-D models from a design tool and manipulate them using straightforward transformations. And key features such as a virtual trackball implementation and 2-D element interactivity are provided by high-level classes that take no expertise at all.

This chapter provided a tour of the core pillars of WPF's 3-D support and introduced some of the indispensable tools that have emerged since WPF 1.0 was released. However, 3-D programming is a detailed topic, and it's certainly possible to delve much more deeply into 3-D theory. If you want to brush up on the math that underlies 3-D development, you may want to consider the book *3D Math Primer for Graphics and Game Development* by Fletcher Dunn (Wordware Publishing, 2002). You're also certain to find complete books on 3-D programming with WPF.

The easiest way to continue your exploration into the world of 3-D is to head to the Web and check out the resources and sample code provided by the WPF team and other independent developers. Here's a short list of useful links, including some that have already been referenced in this chapter:

- `http://www.codeplex.com/3DTools` provides an essential library of tools for developers doing 3-D work in WPF, including the virtual trackball and the ScreenSpaceLines3D class discussed in this chapter.
- `http://blogs.msdn.com/mswanson/articles/WPFToolsAndControls.aspx` provides a list of WPF tools, including 3-D design programs that use XAML natively and export scripts that can transform other 3-D formats (including Maya, LightWave, Blender, and 3ds) to XAML.
- `http://www.therhogue.com/WinFX` includes samples that demonstrate several common 3-D effects (such as a carousel of images) and some more complex techniques (such as an animated mesh).

- http://blogs.msdn.com/danlehen/archive/2005/10/16/481597.aspx includes classes that wrap the meshes required for three common 3-D primitives: a cone, a sphere, and a cylinder.
- http://windowsclient.net/downloads/folders/wpfsamples/entry3743.aspx provides a SandBox3D project that allows you to load simple 3-D meshes and manipulate them with transforms.

If you're in no mood to type in lengthy links, or you want to find out if these addresses have changed, check out the link page for this book at http://www.prosetech.com.

CHAPTER 28

Documents

Using the WPF skills you've picked up so far, you can craft windows and pages that include a wide variety of elements. Displaying fixed text is easy—you simply need to add the TextBlock and Label elements to the mix.

However, the Label and TextBlock aren't a good solution if you need to display large volumes of text (such as a newspaper article or detailed instructions for online help). Large amounts of text are particularly problematic if you want your text to fit in a resizable window in the best possible way. For example, if you pile a large swath of text into a TextBlock and stretch it to fit a wide window, you'll end up with long lines that are difficult to read. Similarly, if you combine text and pictures using the ordinary TextBlock and Image elements, you'll find that they no longer line up correctly when the window changes size.

To deal with these issues, WPF includes a set of higher-level features that work with *documents*. These features allow you to display large amounts of content in a way that makes them easy to read regardless of the size of the containing window. For example, WPF can hyphenate words (if you only have a narrow space available) or place your text into multiple columns (if you have a wide space to work with).

In this chapter, you'll learn how to use *flow documents* to display content. You'll also learn how to let users edit flow document content with the RichTextBox control. Once you've mastered flow documents, you'll take a quick look at XPS, Microsoft's new technology for creating print-ready documents. Finally, you'll consider WPF's annotation feature, which allows users to add comments and other markers to documents and store them permanently.

Understanding Documents

WPF separates documents into two broad categories:

- **Fixed documents.** These are typeset, print-ready documents. The positioning of all content is fixed (for example, the way text is wrapped over multiple lines and hyphenated can't change). Although you might choose to read a fixed document on a computer monitor, fixed documents are intended for print output. Conceptually, they're equivalent to Adobe PDF files. WPF includes a single type of fixed document, which uses Microsoft's XPS (XML Paper Specification) standard.
- **Flow documents.** These are documents that are designed for viewing on a computer. Like fixed documents, flow documents support rich layout. However, WPF can optimize a flow document based on the way you want to view it. It can lay out the content dynamically based on details such as the size of the view window, the display resolution, and so on. Conceptually, flow documents are used for many of the same reasons as HTML documents, but they have more advanced text layout features.

Although flow documents are obviously more important from an application-building point of view, fixed documents are important for documents that need to be printed without alteration (such as forms and publications).

WPF provides support for both types of documents using different containers. The DocumentViewer allows you to show fixed documents in a WPF window. The FlowDocumentReader, FlowDocumentPageViewer, and FlowDocumentScrollViewer give you different ways to look at flow documents. All of these containers are read-only. However, WPF includes APIs for creating fixed documents programmatically, and you can use the RichTextBox to allow the user to edit flow content.

In this chapter, you'll spend most of your time exploring flow documents and the ways they can be used in a WPF application. Toward the end of this chapter, you'll take a look at fixed documents, which are more straightforward.

Flow Documents

In a flow document, the content adapts itself to fit the container. Flow content is ideal for onscreen viewing. In fact, it avoids many of the pitfalls of HTML.

Ordinary HTML content uses flow layout to fill the browser window. (This is the same way WPF organizes elements if you use a WrapPanel.) Although this approach is very flexible, it only gives a good result for a small range of window sizes. If you maximize a window on a high-resolution monitor (or, even worse, a widescreen display), you'll end up with long lines that are extremely difficult to read. Figure 28-1 shows this problem with a portion of a web page from Wikipedia.

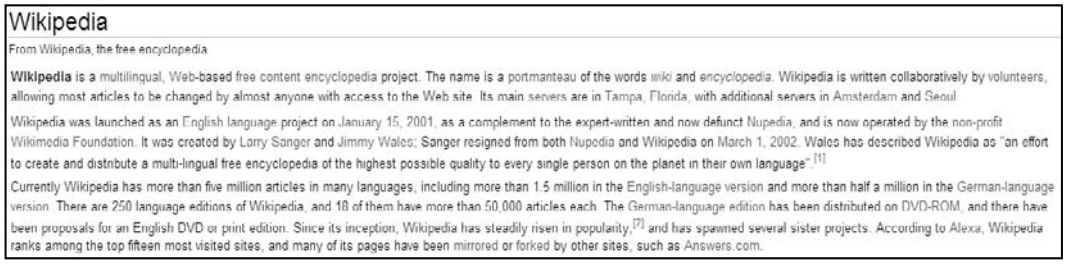

Figure 28-1*. Long lines in flow content*

Many websites avoid this problem by using some sort of fixed layout that forces content to fit a narrow column. (In WPF, you can create this sort of design by placing your content in a column inside a Grid container and setting the ColumnDefinition.MaxWidth property.) This prevents the readability problem, but it results in a fair bit of wasted screen space in large windows. Figure 28-2 shows this problem on a portion of a page from the New York Times website.

Figure 28-2. *Wasted space in flow content*

Flow document content in WPF improves upon these current-day approaches by incorporating better pagination, multicolumn display, sophisticated hyphenation and text flow algorithms, and user-adjustable viewing preferences. The end result is that WPF gives the user a much better experience when reading large amounts of content.

The Flow Elements

You build a WPF flow document using a combination of flow elements. Flow elements have an important difference from the elements you've seen so far. They don't inherit from the familiar UIElement and FrameworkElement classes. Instead, they form an entirely separate branch of classes that derive from ContentElement and FrameworkContentElement.

The content element classes are simpler than the non-content element classes that you've seen throughout this book. However, content elements support a similar set of basic events, including events for keyboard and mouse handling, drag-and-drop operations, tooltip display, and initialization. The key difference between content and non-content elements is that content elements do not handle their own rendering. Instead, they require a container that can render all its content elements. This deferred rendering allows the container to introduce various optimizations. For example, it allows the container to choose the best way to wrap lines of text in a paragraph, even though a paragraph is a single element.

■ **Note** Content elements can accept focus, but ordinarily they don't (because the Focusable property is set to false by default). You can make a content element focusable by setting Focusable to true on individual elements, by using an element type style that changes a whole group of elements, or by deriving your own custom element that sets Focusable to true. The Hyperlink is an example of a content element that sets its Focusable property to true.

Figure 28-3 shows the inheritance hierarchy of content elements.

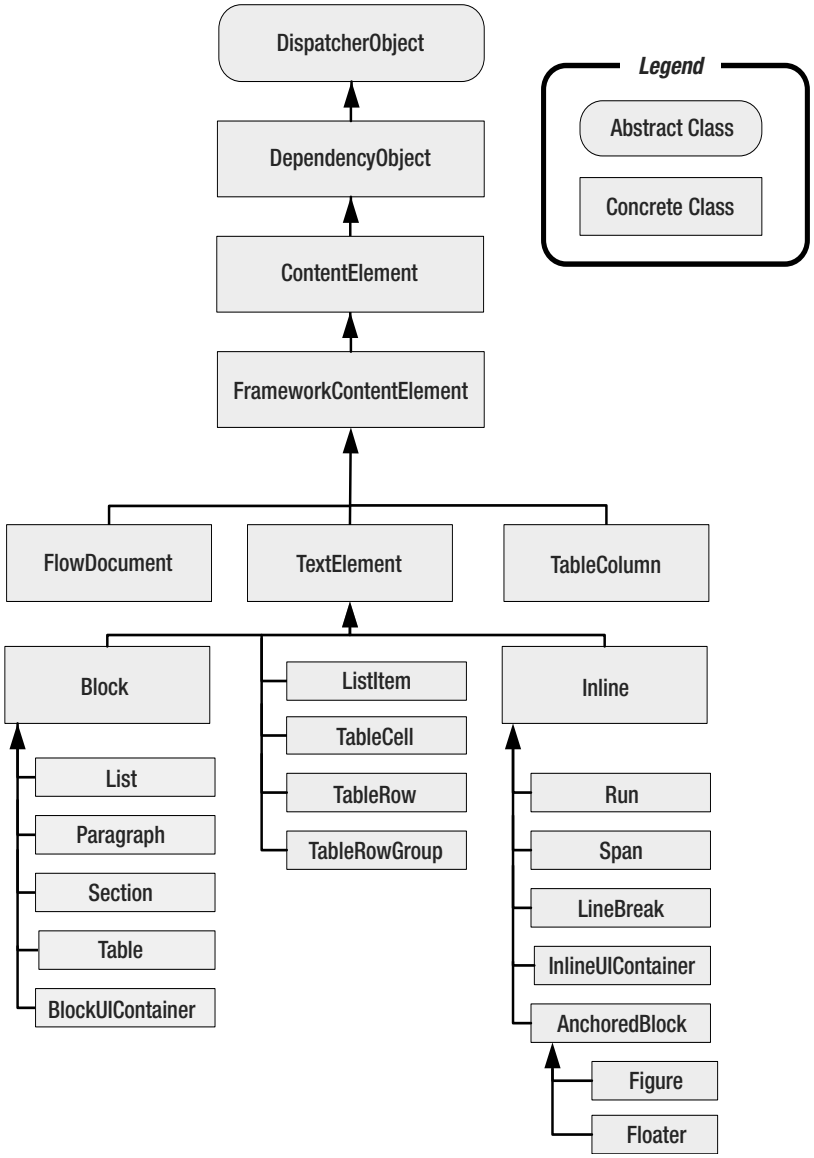

Figure 28-3. *Content elements*

There are two key branches of content elements: