CHAPTER 27

3-D Drawing

WPF includes an expansive 3-D model that allows you to build complex 3-D scenes out of straightforward markup. Helper classes provide hit-testing, mouse-based rotation, and other fundamental building blocks. And virtually any computer can display the 3-D content, thanks to WPF's ability to fall back on software rendering when video card support is lacking.

The most remarkable part of WPF's libraries for 3-D programming is that they are designed to be a clear, consistent extension of the WPF model you've already learned about. For example, you use the same set of brush classes to paint 3-D surfaces as you use to paint 2-D shapes. You use a similar transform model to rotate, skew, and move 3-D objects, and a similar geometry model to define their contours. More dramatically, you can use the same styling, data binding, and animation features on 3-D objects as you use with 2-D content. It's this support of high-level WPF features that makes WPF's 3-D graphics suitable for everything from eye-catching effects in simple games to charting and data visualization in a business application. (The one situation where WPF's 3-D model *isn't* sufficient is high-powered real-time games. If you're planning to build the next Halo, you're much better off with the raw power of DirectX.)

Even though WPF's model for 3-D drawing is surprisingly clear and consistent, creating rich 3-D interfaces is still difficult. In order to code 3-D animations by hand (or just understand the underlying concepts), you need to master more than a little math. And modeling anything but a trivial 3-D scene with handwritten XAML is a huge, error-prone chore—it's far more involved than the 2-D equivalent of creating a XAML vector image by hand. For that reason, you're much more likely to rely on a third-party tool to create 3-D objects, export them to XAML, and then add them to your WPF applications.

Entire books have been written about all these issues—3-D programming math, 3-D design tools, and the 3-D libraries in WPF. In this chapter, you'll learn enough to understand the WPF model for 3-D drawing, create basic 3-D shapes, design more-advanced 3-D scenes with a 3-D modeling tool, and use some of the useful code released by the WPF team and other third-party developers.

3-D Drawing Basics

A 3-D drawing in WPF involves four ingredients:

- A viewport, which hosts your 3-D content
- A 3-D object
- A light source that illuminates part or all of your 3-D scene
- A camera, which provides the vantage point from which you view the 3-D scene

Of course, more-complex 3-D scenes will feature multiple objects and may include multiple light sources. (It's also possible to create a 3-D object that doesn't require a light source, if the 3-D object itself gives off light.) However, these basic ingredients provide a good starting point.

Compared to 2-D graphics, it's the second and third points that really make a difference. Programmers who are new to 3-D programming sometimes assume that 3-D libraries are just a simpler way to create an object that has a 3-D appearance, such as a glowing cube or a spinning sphere. But if that's all you need, you're probably better off creating a 3-D drawing by using the 2-D drawing classes you've already learned about. After all, there's no reason that you can't use the shapes, transforms, and geometries you learned about in Chapter 12 and Chapter 13 to construct a shape that appears to be 3-D—in fact, it's usually easier than working with the 3-D libraries.

So what's the advantage of using the 3-D support in WPF? The first advantage is that you can create effects that would be extremely complex to calculate using a simulated 3-D model. One good example is light effects such as reflection, which become very involved when working with multiple light sources and different materials with different reflective properties. The other advantage to using a 3-D drawing model is that it allows you to interact with your drawing as a set of 3-D objects. This greatly extends what you can do programmatically. For example, after you build the 3-D scene you want, it becomes almost trivially easy to rotate your object or rotate the camera around your object. Doing the same work with 2-D programming would require an avalanche of code (and math).

Now that you know what you need, it's time to build an example that has all these pieces. This is the task you'll tackle in the following sections.

The Viewport

If you want to work with 3-D content, you need a container that can host it. This container is the Viewport3D class, which is found in the System.Windows.Controls namespace. Viewport3D derives from FrameworkElement, and so it can be placed anywhere you'd place a normal element. For example, you can use it as the content of a window or a page, or you can place it inside a more complex layout.

The Viewport3D class only hints at the complexity of 3-D programming. It adds just two properties—Camera, which defines your lookout onto the 3-D scene, and Children, which holds all the 3-D objects you want to place in the scene. Interestingly enough, the light source that illuminates your 3-D scene is itself an object in the viewport.

■ **Note** Among the inherited properties in the Viewport3D class, one is particularly significant: ClipToBounds. If set to true (the default), content that stretches beyond the bounds of the viewport is trimmed out. If set to false, this content appears on top of any adjacent elements. This is the same behavior you get from the ClipToBounds property of the Canvas. However, there's an important difference when using the Viewport3D: performance. Setting Videport3D.ClipToBounds to false can dramatically improve performance when rendering a complex, frequently refreshed 3-D scene.

3-D Objects

The viewport can host any 3-D object that derives from Visual3D (from the System.Windows.Media.Media3D namespace, where the vast majority of the 3-D classes live). However, you'll need to perform a bit more work than you might expect to create a 3-D visual. In version 1.0, the WPF library lacks a collection of 3-D shape primitives. If you want a cube, a cylinder, a torus, and so on, you'll need to build it yourself.

One of the nicest design decisions that the WPF team made when building the 3-D drawing classes was to structure them in a similar way as the 2-D drawing classes. That means you'll immediately be able to understand the purpose of a number of core 3-D classes (even if you don't yet know how to use them). Table 27-1 spells out the relationships.

Table 27-1. *2-D Classes and 3-D Classes Compared*

2-D Class	3-D Class	Notes
Visual	Visual3D	Visual3D is the base class for all 3-D objects (objects that are rendered in a Viewport3D container). As with the Visual class, you could use the Visual3D class to derive lightweight 3-D shapes or to create more-complex 3-D controls that provide a richer set of events and framework services. However, you won't get much help. You're more likely to use one of the classes that derive from Visual3D, such as ModelVisual3D or ModelUIElement3D.
Geometry	Geometry3D	The Geometry class is an abstract way to define a 2-D figure. Often geometries are used to define complex figures that are composed of arcs, lines, and polygons. The Geometry3D class is the 3-D analogue—it represents a 3-D surface. However, while there are several 2-D geometries, WPF includes just a single concrete class that derives from Geometry3D: MeshGeometry3D. The MeshGeometry3D class has a central importance in 3-D drawing because you'll use it to define all your 3-D objects.
GeometryDrawing	GeometryModel3D	There are several ways to use a 2-D Geometry object. You can wrap it in a GeometryDrawing and use that to paint the surface of an element or the content of a Visual. The GeometryModel3D class serves the same purpose—it takes a Geometry3D, which can then be used to fill your Visual3D.
Transform	Transform3D	You already know that 2-D transforms are incredibly useful tools for manipulating elements and shapes in all kinds of ways, including moving, skewing, and rotating them. Transforms are also indispensable when performing animations. Classes that derive from Transform3D perform the same magic with 3-D objects. In fact, you'll find surprisingly similar transform classes such as RotateTransform3D, ScaleTransform3D, TranslateTransform3D, Transform3DGroup, and MatrixTransform3D. Of course, the options provided by an extra dimension are considerable, and 3-D transforms are able to warp and distort visuals in ways that look quite different.

At first, you may find it a bit difficult to untangle the relationships between these classes. Essentially, the Viewport3D holds Visual3D objects. To actually give a Visual3D some content, you'll need to define a

Geometry3D that describes the shape and wrap it in a GeometryModel3D. You can then use that as the content for your Visual3D. Figure 27-1 shows this relationship.

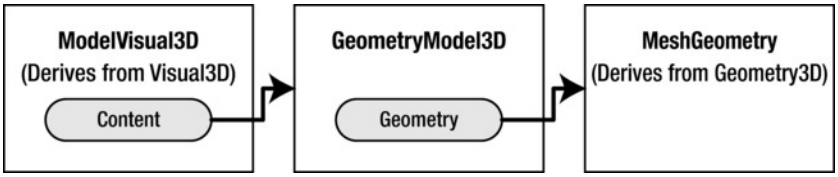

Figure 27-1. *How a 3-D object is defined*

This two-step process—defining the shapes you want to use in abstract and then fusing them with a visual—is an optional approach for 2-D drawing. However, it's mandatory for 3-D drawing because there are no prebuilt 3-D classes in the library. (The members of the WPF team and others have released some sample code online that starts to fill this gap, but it's still evolving.)

The two-step process is also important because 3-D models are a bit more complex than 2-D models. For example, when you create a Geometry3D object, you specify not only the vertexes of your shape, but also the *material* out of which it's composed. Different materials have different properties for reflecting and absorbing light.

Geometry

To build a 3-D object, you need to start by building the geometry. As you've already learned, there's just one class that fills this purpose: MeshGeometry3D.

Unsurprisingly, a MeshGeometry3D object represents a *mesh*. If you've ever dealt with 3-D drawing before (or if you've read a bit about the technology that underlies modern video cards), you may already know that computers prefer to build 3-D drawings out of triangles. That's because a triangle is the simplest, most granular way to define a surface. Triangles are simple because every triangle is defined by just three points (the vertexes at the corner). Arcs and curved surfaces are obviously more complex. Triangles are granular because other straight-edged shapes (squares, rectangles, and more-complex polygons) can be broken down into a collection of triangles. For better or worse, modern day graphics hardware and graphics programming is built on this core abstraction.

Obviously, most of the 3-D objects you want won't look like simple, flat triangles. Instead you'll need to combine triangles—sometimes just a few, but often hundreds or thousands that line up with one another at varying angles. A mesh is this combination of triangles. With enough triangles, you can ultimately create the illusion of anything, including a complex surface. (Of course, there are performance considerations involved, and 3-D scenes often map some sort of bitmap or 2-D content onto a triangle in a mesh to create the illusion of a complex surface with less overhead. WPF supports this technique.)

Understanding how a mesh is defined is one of the first keys to 3-D programming. If you look at the MeshGeometry3D class, you'll find that it adds the four properties listed in Table 27-2.

Table 27-2. *Properties of the MeshGeometry3D Class*

Name	Description
Positions	Contains a collection of all the points that define the mesh. Each point is a vertex in a triangle. For example, if your mesh has 10 completely separate triangles, you'll have 30 points in this collection. More commonly, some of your triangles will join at their edges, which means one point will become the vertex of several triangles. For example, a cube requires 12 triangles (two for each side), but only 8 distinct points. Making matters even more complicated, you may choose to define the same shared vertex multiple times, so that you can better control how separate triangles are shaded with the Normals property.
TriangleIndices	Defines the triangles. Each entry in this collection represents a single triangle by referring to three points from the Positions collection.
Normals	Provides a vector for each vertex (each point in the Positions collection). This vector indicates how the point is angled for lighting calculations. When WPF shades the face of a triangle, it measures the light at each of the three vertexes using the normal vector. Then, it interpolates between these three points to fill the surface of the triangle. Getting the right normal vectors makes a substantial difference to how a 3-D object is shaded—for example, it can make the divisions between triangles blend together or appear as sharp lines.
TextureCoordinates	Defines how a 2-D texture is mapped onto your 3-D object when you use a VisualBrush to paint it. The TextureCoordinates collection provides a 2-D point for each 3-D point in the Positions collection.

You'll consider shading with normals and texture mapping later in this chapter. But first, you'll learn how to build a basic mesh.

The following example shows the simplest possible mesh, which consists of a single triangle. The units you use aren't important because you can move the camera closer or farther away, and you can change the size or placement of individual 3-D objects by using transforms. What *is* important is the coordinate system, which is shown in Figure 27-2. As you can see, the X and Y axes have the same orientation as in 2-D drawing. What's new is the Z axis. As the Z axis value decreases, the point moves farther away. As it increases, the point moves closer.

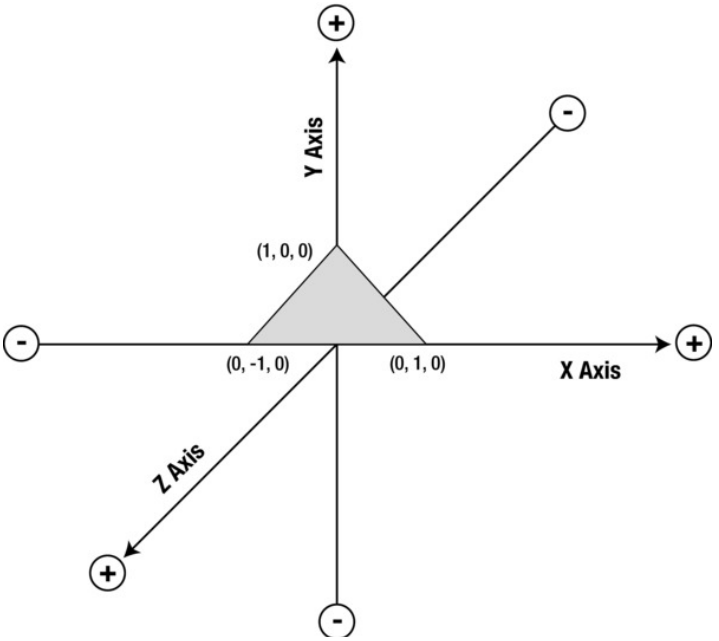

Figure 27-2. *A triangle in 3-D space*

Here's the MeshGeometry element that you can use to define this shape inside a 3-D visual. The MeshGeometry3D object in this example doesn't use the Normals property or the TextureCoordinates property because the shape is so simple and will be painted with a SolidColorBrush:

```
<MeshGeometry3D Positions="-1,0,0  0,1,0  1,0,0" TriangleIndices="0,2,1" />
```

Here, there are obviously just three points, which are listed one after the other in the Positions property. The order you use in the Positions property isn't important because the TriangleIndices property clearly defines the triangle. Essentially, the TriangleIndices property states that there is a single triangle made of point #0, #2, and #1. In other words, the TriangleIndices property tells WPF to draw the triangle by drawing a line from (-1, 0 ,0) to (1, 0, 0) and then to (0, 1, 0).

3-D programming has several subtle, easily violated rules. When defining a shape, you'll face the first one—namely, you must list the points in a counterclockwise order around the Z axis. This example follows that rule. However, you could easily violate it if you changed the TriangleIndices to 0, 1, 2. In this case, you'd still define the same triangle, but that triangle would be backward—in other words, if you look at it down the Z axis (as in Figure 27-2), you'll actually be looking at the *back* of the triangle.

■ **Note** The difference between the back of a 3-D shape and the front is not a trivial one. In some cases, you may paint both with a different brush. Or you may choose not to paint the back at all in order to avoid using any resources for a part of the scene that you'll never see. If you inadvertently define the points in a clockwise order, and you haven't defined the material for the back of your shape, it will disappear from your 3-D scene.

Geometry Model and Surfaces

After you have the properly configured MeshGeometry3D that you want, you need to wrap it in a GeometryModel3D.

The GeometryModel3D class has just three properties: Geometry, Material, and BackMaterial. The Geometry property takes the MeshGeometry3D that defines the shape of your 3-D object. In addition, you can use the Material and BackMaterial properties to define the surface out of which your shape is composed.

The surface is important for two reasons. First, it defines the color of the object (although you can use more-complex brushes that paint textures rather than solid colors). Second, it defines how that material responds to light.

WPF includes four material classes, all of which derive from the abstract Material class in the System.Windows.Media.Media3D namespace). They're listed in Table 27-3. In this example, we'll stick with DiffuseMaterial, which is the most common choice because its behavior is closest to a real-world surface.

Table 27-3. *Material Classes*

Name	Description
DiffuseMaterial	Creates a flat, matte surface. It diffuses light evenly in all directions.
SpecularMaterial	Creates a glossy, highlighted look (think metal or glass). It reflects light back directly, like a mirror.
EmissiveMaterial	Creates a glowing look. It generates its own light (although this light does not reflect off other objects in the scene).
MaterialGroup	Lets you combine more than one material. The materials are then layered on top of one another in the order they're added to the MaterialGroup.

DiffuseMaterial offers a single Brush property that takes the Brush object you want to use to paint the surface of your 3-D object. (If you use anything other than a SolidColorBrush, you'll need to set the MeshGeometry3D.TextureCoordinates property to define the way it's mapped onto the object, as you'll see later in this chapter.)

Here's how you can configure the triangle to be painted with a yellow matte surface:

```
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="-1,0,0  0,1,0  1,0,0" TriangleIndices="0,2,1" />
  </GeometryModel3D.Geometry>

  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Yellow" />
  </GeometryModel3D.Material>
</GeometryModel3D>
```

In this example, the BackMaterial property is not set, so the triangle will disappear if viewed from behind.

All that remains is to use this GeometryModel3D to set the Content property of a ModelVisual3D and then place that ModelVisual3D in a viewport. But in order to see your object, you'll also need two more details: a light source and a camera.

Light Sources

In order to create realistically shaded 3-D objects, WPF uses a lighting model. The basic idea is that you add one (or several) light sources to your 3-D scene. Your objects are then illuminated based on the type of light you've chosen, its position, direction, and intensity.

Before you delve into WPF lighting, it's important that you realize that the WPF lighting model doesn't behave like light in the real world. Although the WPF lighting system is constructed to emulate the real world, calculating true light reflections is a processor-intensive task. WPF uses a number of simplifications that ensure the lighting model is practical, even in animated 3-D scenes with multiple light sources. These simplifications include the following:

- Light effects are calculated for objects *individually*. Light reflected from one object will not reflect off another object. Similarly, an object will not cast a shadow on another object, no matter where it's placed.
- Lighting is calculated at the vertexes of each triangle and then interpolated over the surface of the triangle. (In other words, WPF determines the light strength at each corner and blends that to fill in the triangle.) As a result of this design, objects that have relatively few triangles may not be illuminated correctly. To achieve better lighting, you'll need to divide your shapes into hundreds or thousands of triangles.

Depending on the effect you're trying to achieve, you may need to work around these issues by combining multiple light sources, using different materials, and even adding extra shapes. In fact, getting the precise result you want is part of the art of 3-D scene design.

■ **Note** Even if you don't provide a light source, your object will still be visible. However, without a light source, all you'll see is a solid black silhouette.

WPF provides four light classes, all of which derive from the abstract Light class. Table 27-4 lists them all. In this example, we'll stick with a single DirectionalLight, which is the most common type of lighting.

Table 27-4. *Light Classes*

Name	Description
DirectionalLight	Fills the scene with parallel rays of light traveling in the direction you specify.
AmbientLight	Fills the scene with scattered light.
PointLight	Radiates light in all directions, beginning at a single point in space.
SpotLight	Radiates light outward in a cone, starting from a single point.

Here's how you can define a white DirectionalLight:

```
<DirectionalLight Color="White" Direction="-1,-1,-1" />
```

In this example, the vector that determines the path of the light starts at the origin (0, 0, 0) and goes to (-1, -1, -1). That means that each ray of light is a straight line that travels from top-right front toward the bottom-left back. This makes sense in this example because the triangle (shown in Figure 27-2) is angled to face this light.

When calculating the light direction, it's the angle that's important, not the length of your vector. That means a light direction of (-2, -2, -2) is equivalent to the normalized vector (-1, -1, -1) because the angle it describes is the same.

In this example, the direction of the light doesn't line up exactly with the triangle's surface. If that's the effect you want, you'll need a light source that sends its beams straight down the Z axis, using a direction of (0, 0, -1). This distinction is deliberate. Because the beams strike the triangle at an angle, the triangle's surface will be shaded, which creates a more pleasing effect.

Figure 27-3 shows an approximation of the (-1, -1, -1) directional light as it strikes the triangle. Remember, a directional light fills the entire 3-D space.

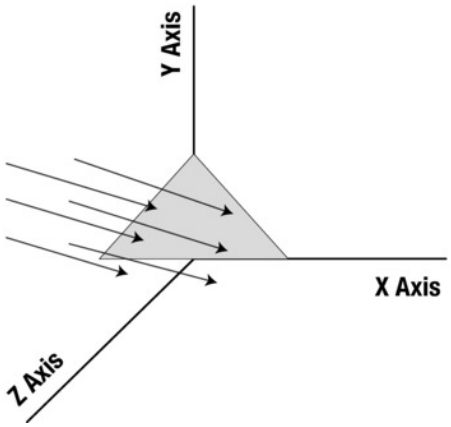

Figure 27-3. *The path of a (-1, -1, -1) directional light*

■ **Note** Directional lights are sometimes compared to sunlight. That's because the light rays received from a faraway light source (such as the sun) become almost parallel.

All light objects derive indirectly from GeometryModel3D. That means that you treat them exactly like 3-D objects by placing them inside a ModelVisual3D and adding them to a viewport. Here's a viewport that includes both the triangle you saw earlier and the light source:

```
<Viewport3D>
  <Viewport3D.Camera>...</Viewport3D.Camera>

  <ModelVisual3D>
    <ModelVisual3D.Content>
      <DirectionalLight Color="White" Direction="-1,-1,-1" />
    </ModelVisual3D.Content>
  </ModelVisual3D>

  <ModelVisual3D>
    <ModelVisual3D.Content>
      <GeometryModel3D>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D Positions="-1,0,0 0,1,0 1,0,0" TriangleIndices="0,2,1" />
        </GeometryModel3D.Geometry>
        <GeometryModel3D.Material>
          <DiffuseMaterial Brush="Yellow" />
```

```
        </GeometryModel3D.Material>
      </GeometryModel3D>
    </ModelVisual3D.Content>
  </ModelVisual3D>

</Viewport3D>
```

There's one detail that's left out of this example—the viewport doesn't include a camera that defines your vantage point on the scene. That's the task you'll tackle in the next section.

A CLOSER LOOK AT 3-D LIGHTING

Along with DirectionalLight, AmbientLight is another all-purpose lighting class. Using AmbientLight on its own gives 3-D shapes a flat look, but you can combine it with another light source to add some illumination that brightens up otherwise darkened areas. The trick is to use an AmbientLight that's less than full strength. Instead of using a white AmbientLight, use one-third white (set the Color property to #555555) or less. You can also set the DiffuseMaterial.AmbientColor property to control how strongly an AmbientLight affects the material in a given mesh. Using white (the default) gives the strongest effect, while using black creates a material that doesn't reflect any ambient light.

The DirectionalLight and AmbientLight are the most useful lights for simple 3-D scenes. The PointLight and SpotLight give the effect you want only if your mesh includes a large number of triangles—typically hundreds. This is due to the way that WPF shades surfaces.

As you've already learned, WPF saves time by calculating the lighting intensity only at the vertexes of a triangle. If your shape uses a small number of triangles, this approximation breaks down. Some points will fall inside the range of the SpotLight or PointLight, while others won't. The result is that some triangles will be illuminated while others will remain in complete darkness. Rather than getting a soft, rounded circle of light on your object, you'll end up with a group of illuminated triangles, giving the illuminated area a jagged edge.

The problem here is that PointLight and SpotLight are used to create soft, circular lighting effects, but you need a very large number of triangles to create a circular shape. (To create a perfect circle, you need one triangle for each pixel that lies on the perimeter of the circle.) If you have a 3-D mesh with hundreds or thousands of triangles, the pattern of partially illuminated triangles can more easily approximate a circle, and you'll get the lighting effect you want.

The Camera

Before a 3-D scene can be rendered, you need to place a camera at the correct position and orient it in the correct direction. You do this by setting the Viewport3D.Camera property with a Camera object.

In essence, the camera determines how a 3-D scene is projected onto the 2-D surface of a Viewport. WPF includes three camera classes: the commonly used PerspectiveCamera and the more exotic OrthographicCamera and MatrixCamera. The PerspectiveCamera renders the scene so that objects that are farther away appear smaller. This is the behavior that most people expect in a 3-D scene. The OrthographicCamera flattens 3-D objects so that the exact scale is preserved, no matter where a shape is positioned. This looks a bit odd, but it's useful for some types of visualization tools. For example, technical drawing applications often rely on this type of view. (Figure 27-4 shows the difference between the PerspectiveCamera and the OrthographicCamera.) Finally, the MatrixCamera allows you to specify a matrix that's used to transform the 3-D scene to 2-D view. It's an advanced tool that's intended for highly

specialized effects and for porting code from other frameworks (such as Direct3D) that use this type of camera.

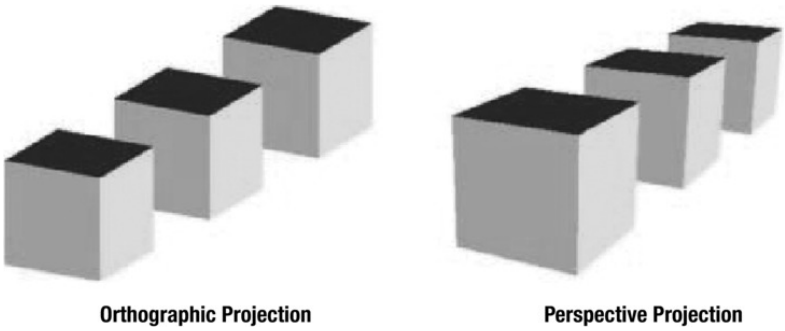

Figure 27-4. *Perspective in different types of cameras*

Choosing the right camera is relatively easy, but placing and configuring it is a bit trickier. The first detail is to specify a point in 3-D space where the camera will be positioned by setting its Position property. The second step is to set a 3-D vector in the LookDirection property that indicates how the camera is oriented. In a typical 3-D scene, you'll place the camera slightly off to one corner by using the Position property, and then tilt it to survey the view by using the LookDirection property.

■ **Note** The position of the camera determines how large your scene appears in the viewport. The closer the camera, the larger the scale. In addition, the viewport is stretched to fit its container and the content inside is scaled accordingly. For example, if you create a viewport that fills a window, you can expand or shrink your scene by resizing the window.

You need to set the Position and LookDirection properties in concert. If you use Position to offset the camera but fail to compensate by turning the camera back in the right direction by using LookDirection, you won't see the content you've created in your 3-D scene. To make sure you're correctly oriented, pick a point that you want to see square on from your camera. You can then calculate the look direction by using this formula:

```
CameraLookDirection = CenterPointOfInterest - CameraPosition
```

In the triangle example, the camera is placed in the top-left corner by using a position of (-2, 2, 2). Assuming you want to focus on the origin point (0, 0, 0), which falls in the middle of the triangle's bottom edge, you would use this look direction:

```
CameraLookDirection = (0, 0, 0) - (-2, 2, 2)
                    = (2, -2, -2)
```

This is equivalent to the normalized vector (1, -1, -1) because the direction it describes is the same. As with the Direction property of a DirectionalLight, it's the direction of the vector that's important, not its magnitude.

After you've set the Position and LookDirection properties, you may also want to set the UpDirection properties. UpDirection determines how the camera is tilted. Ordinarily, UpDirection is set to (0, 1, 0), which means the up direction is straight up, as shown in Figure 27-5.

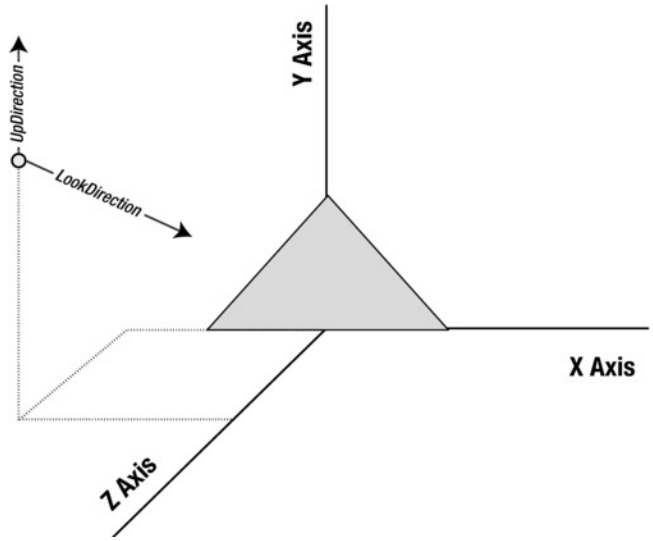

Figure 27-5. *Positioning and angling the camera*

If you offset this slightly—say to (0.25, 1, 0)—the camera is tilted around the X axis, as shown in Figure 27-6. As a result, the 3-D objects will appear to be tilted a bit in the other direction. It's just as if you'd cocked your head to one side while surveying the scene.

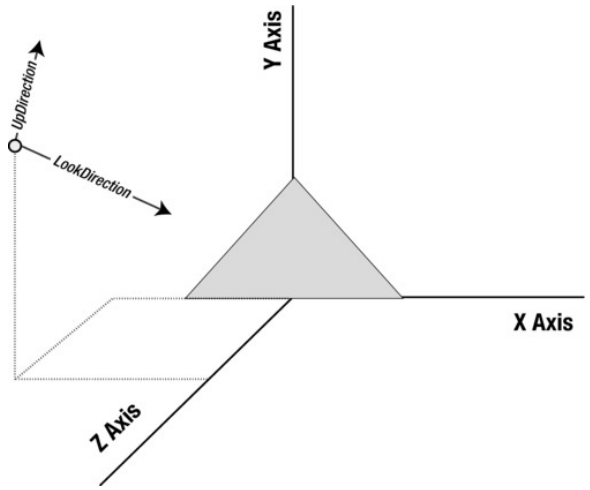

Figure 27-6. *Another way to angle the camera*

With these details in mind, you can define the PerspectiveCamera for the simple one-triangle scene that's been described over the previous sections:

```
<Viewport3D>
  <Viewport3D.Camera>
    <PerspectiveCamera Position="-2,2,2" LookDirection="2,-2,-2"
     UpDirection="0,1,0" />
  </Viewport3D.Camera>
  ...
</Viewport3D>
```

Figure 27-7 shows the final scene.

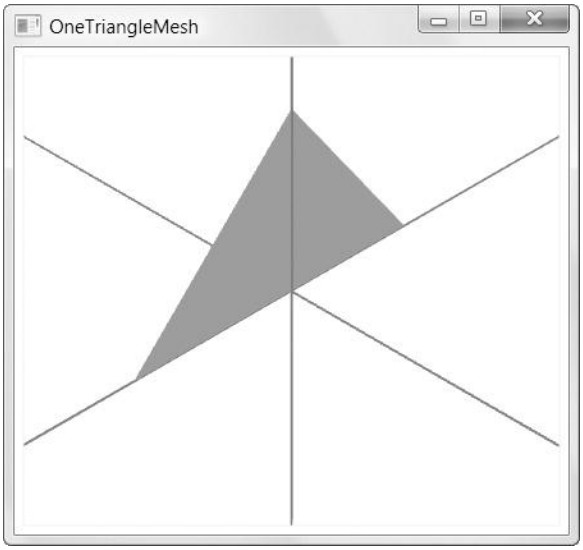

Figure 27-7. *A complete 3-D scene with one triangle*

AXIS LINES

There's one added detail in Figure 27-7: the axis lines. These lines are a great testing tool, as they make it easy to see where your axes are placed. If you render a 3-D scene and nothing appears, the axis lines can help you isolate the potential problem, which could include a camera pointing in the wrong direction or positioned off to one side, or a shape that's flipped backward (and thus invisible). Unfortunately, WPF doesn't include any class for drawing straight lines. Instead, you need to render long, vanishingly narrow triangles.

Fortunately, there's a tool that can help. The WPF 3-D team has created a handy ScreenSpaceLines3D that solves the problem in a freely downloadable class library that's available (with complete source code) at `http://3dtools.codeplex.com`. This project includes several other useful code ingredients, including the Trackball described later in this chapter in the "Interactivity and Animations" section.

The ScreenSpaceLines3D class allows you to draw straight lines with an invariant width. In other words, these lines have the fixed thickness that you choose no matter where you place the camera. (They do not become thicker as the camera gets closer, and thinner as it recedes.) This makes these lines useful to create wireframes, boxes that indicate content regions, vector lines that indicate the normal for lighting calculations,

and so on. These applications are most useful when building a 3-D design tool or when debugging an application. The example in Figure 27-5 uses the ScreenSpaceLines3D class to draw the axis lines.

There are a few other camera properties that are often important. One of these is FieldOfView, which controls how much of your scene you can see at once. FieldOfView is comparable to a zoom lens on a camera—as you decrease the FieldOfView, you see a smaller portion of the scene (which is then enlarged to fit the Viewport3D). As you increase the FieldOfView, you see a larger part of the scene. However, it's important to remember that changing the field of view is *not* the same as moving the camera closer or farther away from the objects in your scene. Smaller fields of view tend to compress the distance between near and far objects, while wider fields of view exaggerate the perspective difference between near and far objects. (If you've played with camera lenses before, you may have noticed this effect.)

■ **Note** The FieldOfView property applies to only the PerspectiveCamera. The OrthographicCamera includes a Width property that's analogous. The Width property determines the viewable area but it doesn't change the perspective because no perspective effect is used for the OrthographicCamera.

The camera classes also include NearPlaneDistance and FarPlaneDistance properties that set the blind spots of the camera. Objects closer than the NearPlaneDistance won't appear at all, and objects farther than the FarPlaneDistance are similarly invisible. Ordinarily, NearPlaneDistance defaults to 0.125, and FarPlaneDistance defaults to Double.PositiveInfinity, which renders both effects negligible. However, in some cases you'll need to change these values to prevent rendering artifacts. The most common example occurs when a complex mesh is extremely close to the camera, which can cause z-fighting (also known as *stitching*). In this situation, the video card is unable to correctly determine which triangles are closest to the camera and should be rendered. The result is a pattern of artifacts of the surface of your mesh.

Z-fighting usually occurs because of floating-point round-off errors in the video card. To avoid this problem, you can increase the NearPlaneDistance to clip objects that are extremely close to the camera. Later in this chapter, you'll see an example that animates the camera so it flies through the center of a torus. To create this effect without causing z-fighting, it's necessary to increase the NearPlaneDistance.

■ **Note** Rendering artifacts are almost always the result of objects close to the camera and a NearPlaneDistance that's too large. Similar problems with very distant objects and the FarPlaneDistance are much less common.

Deeper into 3-D

Going to the trouble of cameras, lights, materials, and mesh geometries is a lot of work for an unimpressive triangle. However, you've now seen the bare bones of WPF's 3-D support. In this section, you'll learn how to use it to introduce more-complex shapes.

After you've mastered the lowly triangle, the next step up is to create a solid, faceted shape by assembling a small group of triangles. In the following example, you'll create the markup for the cube shown in Figure 27-8.

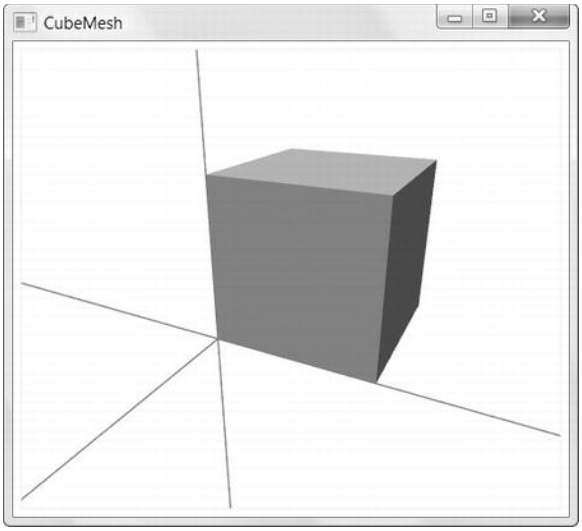

Figure 27-8. *A 3-D cube*

The first challenge to building your cube is determining how to break it down into the triangles that the MeshGeometry object recognizes. Each triangle acts like a flat, 2-D shape.

A cube consists of six square sides. Each square side needs two triangles. Each square side can then be joined to the adjacent side at an angle. Figure 27-9 shows how a cube breaks down into triangles.

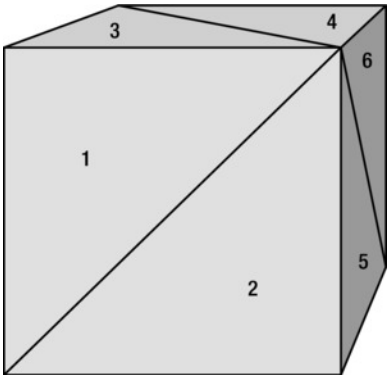

Figure 27-9. *Breaking the cube into triangles*

To reduce overhead and improve performance in a 3-D program, it's common to avoid rendering shapes that you won't see. For example, if you know you'll never look at the underside of the cube shown in Figure 27-8, there's no reason to define the two triangles for that side. However, in this example you'll define every side so you can rotate the cube freely.

Here's a MeshGeometry3D that creates a cube:

```
<MeshGeometry3D Positions="0,0,0   10,0,0   0,10,0   10,10,0
                           0,0,10  10,0,10  0,10,10  10,10,10"
                TriangleIndices="0,2,1  1,2,3  0,4,2  2,4,6
                                 0,1,4  1,5,4  1,7,5  1,3,7
                                 4,5,6  7,6,5  2,6,3  3,6,7" />
```

First, the Positions collection defines the corners of the cube. It begins with the four points in the back (where z = 0) and then adds the four in the front (where z = 10). The TriangleIndices property maps these points to triangles. For example, the first entry in the collection is 0, 2, 1. It creates a triangle from the first point (0, 0, 0) to the second point (0, 0, 10) to the third point (0, 10, 0). This is one of the triangles required for the back side of the square. (The index 1, 2, 3 fills in the other backside triangle.)

Remember, when defining triangles, you must define them in counterclockwise order to make their front side face forward. However, the cube appears to violate that rule. The squares on the front side are defined in counterclockwise order (see the index 4, 5, 6 and 7, 6, 5, for instance), but those on the back side are defined in clockwise order, including the index 0, 2, 1 and 1, 2, 3. This is because the back side of the cube must have its triangle facing backward. To better visualize this, imagine rotating the cube around the Y axis so that the back side is facing forward. Now, the backward-facing triangles will be facing forward, making them completely visible, which is the behavior you want.

Shading and Normals

There's one issue with the cube mesh demonstrated in the previous section. It doesn't create the faceted cube shown in Figure 27-8. Instead, it gives you the cube shown in Figure 27-10, with clearly visible seams where the triangles meet.

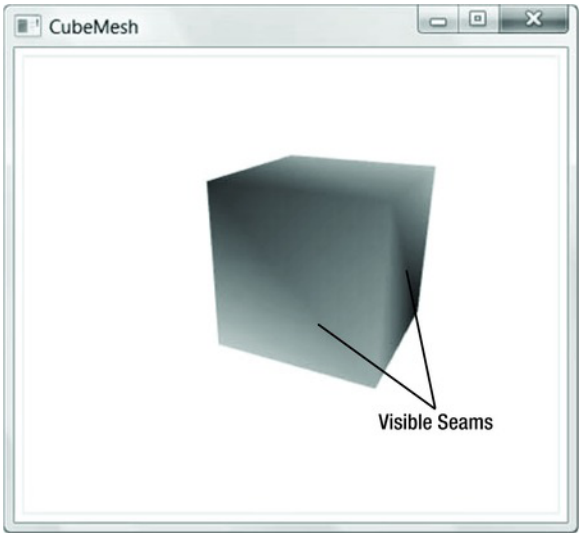

Figure 27-10. *A cube with lighting artifacts*

This problem results from the way that WPF calculates lighting. In order to simplify the calculation process, WPF computes the amount of light that reaches each vertex in a shape—in other words, it pays attention to only the corners of your triangles. It then blends the lighting over the surface of the triangle.

While this ensures that every triangle is nicely shaded, it may cause other artifacts. For example, in this situation it prevents the adjacent triangles that share a cube side from being shaded evenly.

To understand why this problem occurs, you need to know a little more about normals. Each normal defines how a vertex is oriented toward the light source. In most cases, you'll want your normal to be perpendicular to the surface of your triangle.

Figure 27-11 illustrates the front face of a cube. The front face has two triangles and a total of four vertexes. Each of these four vertexes should have a normal that points outward at a right angle to the square's surface. In other words, each normal should have a direction of (0, 0, 1).

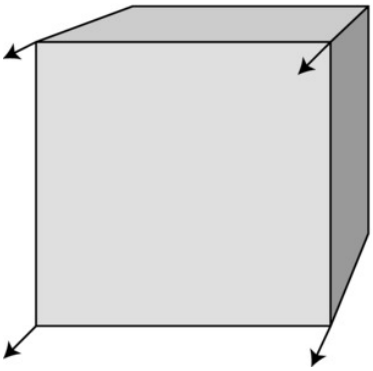

Figure 27-11. *Normals on the front side of a cube*

■ **Tip** Here's another way to think about normals. When the normal vector lines up with the light direction vector, but in opposite directions, the surface will be fully illuminated. In this example, that means a directional light with a direction of (0, 0, -1) will completely light up the front surface of the cube, which is what you expect.

The triangles on the other sides of the square need their own normals as well. In each case, the normals should be perpendicular to the surface. Figure 27-12 fills in the normals on the front, top, and right sides of the cube.

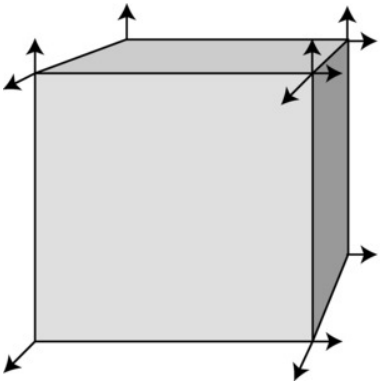

Figure 27-12. *Normals on the visible faces of a cube*

The cube diagrammed in Figure 27-12 is the same cube shown in Figure 27-8. When WPF shades this cube, it examines it one triangle at a time. For example, consider the front surface. Each point faces the directional light in exactly the same way. For that reason, each point will have exactly the same illumination. As a result, when WPF blends the illumination at the four corners, it creates a flat, consistently colored surface with no shading.

So why doesn't the cube you've just created exhibit this lighting behavior? The culprit is the shared points in the Positions collection. Although normals apply to the way triangles are shaded, they're defined on only the vertexes of the triangle. Each point in the Positions collection has just a single normal defined for it. That means if you share points between two triangles, you also end up sharing normals.

That's what's happened in Figure 27-10. The different points on the same side are illuminated differently because they don't all have the same normal. WPF then blends the illumination from these points to fill in the surface of each triangle. This is a reasonable default behavior, but because the blending is performed on each triangle, different triangles won't line up exactly, and you'll see the seams of color where the separate triangles meet.

One easy (but tedious) way to solve this problem is to make sure no points are shared between triangles by declaring each point several times (once for each time it's used). Here's the lengthier markup that does this:

```
<MeshGeometry3D Positions="0,0,0    10,0,0   0,10,0   10,10,0
                           0,0,0   0,0,10   0,10,0   0,10,10
                           0,0,0   10,0,0   0,0,10   10,0,10
                           10,0,0  10,10,10 10,0,10  10,10,0
                           0,0,10  10,0,10  0,10,10  10,10,10
                           0,10,0  0,10,10  10,10,0  10,10,10"
                TriangleIndices="0,2,1     1,2,3
                                 4,5,6     6,5,7
                                 8,9,10    9,11,10
                                 12,13,14  12,15,13
                                 16,17,18  19,18,17
                                 20,21,22  22,21,23" />
```

In this example, this step saves you from needing to code the normals by hand. WPF correctly generates them for you, making each normal perpendicular to the triangle surface, as shown in Figure 27-11. The result is the faceted cube shown in Figure 27-8.

■ **Note** Although this markup is much longer, the overhead is essentially unchanged. That's because WPF always renders your 3-D scene as a collection of distinct triangles, whether or not you share points in the Positions collection.

It's important to realize that you don't always want your normals to match. In the cube example, it's a requirement to get the faceted appearance. However, you might want a different lighting effect. For example, you might want a blended cube that avoids the seam problem shown earlier. In this case, you'll need to define your normal vectors explicitly.

Choosing the right normals can be a bit tricky. However, to get the result you want, keep these two principles in mind:

- To calculate a normal that's perpendicular to a surface, calculate the cross product of the vectors that make up any two sides of your triangle. However, make sure to keep the points in counterclockwise order so that the normal points out from the surface (instead of into it).

- If you want the blending to be consistent over a surface that includes more than one triangle, make sure all the points in all the triangles share the same normal.

To calculate the normal you need for a surface, you can use a bit of C# code. Here's a simple code routine that can help you calculate a normal that's perpendicular to the surface of a triangle based on its three points:

```
private Vector3D CalculateNormal(Point3D p0, Point3D p1, Point3D p2)
{
    Vector3D v0 = new Vector3D(p1.X - p0.X, p1.Y - p0.Y, p1.Z - p0.Z);
    Vector3D v1 = new Vector3D(p2.X - p1.X, p2.Y - p1.Y, p2.Z - p1.Z);
    return Vector3D.CrossProduct(v0, v1);
}
```

Next, you need to set the Normals property by hand by filling it with vectors. Remember, you must add one normal for each position.

The following example smooths the blending between adjacent triangles on the same side of a rectangle by sharing normals. The adjacent triangles on a cube face share two of the same points. Therefore, it's only the two nonshared points that need to be adjusted. As long as they match, the shading will be consistent over the entire surface:

```
<MeshGeometry3D Positions="0,0,0 10,0,0 0,10,0 10,10,0
                           0,0,10 10,0,10 0,10,10 10,10,10"
                TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6
                                 0,1,4 1,5,4 1,7,5 1,3,7
                                 4,5,6 7,6,5 2,6,3 3,6,7"
                Normals="0,1,0 0,1,0 1,0,0 1,0,0
                         0,1,0 0,1,0 1,0,0 1,0,0" />
```

This creates the smoother cube shown in Figure 27-13. Now large portions of the cube end up sharing the same normal. This causes an extremely smooth effect that blends the edges of the cube, making it more difficult to distinguish the sides.

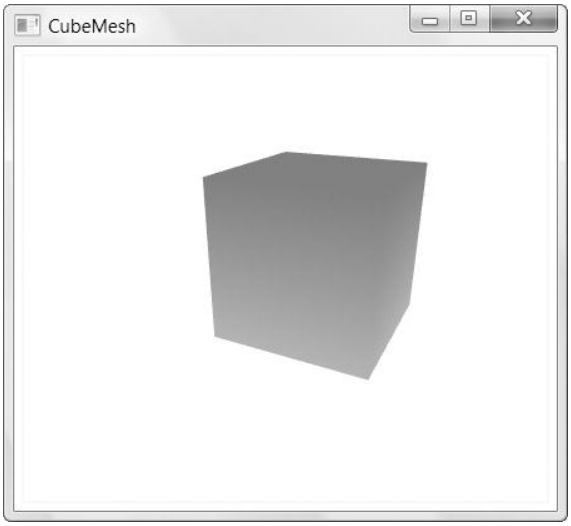

Figure 27-13. *An extremely smooth cube*

This effect isn't correct or incorrect—it simply depends on the effect you're trying to achieve. For example, faceted sides create a more geometric look, while blended sides look more organic. One common trick is to use blending with a large multifaceted polygon to make it look like a sphere, a cylinder, or another sort of curved shape. Because the blending hides the edges of the shape, this effect works remarkably well.

More Complex Shapes

Realistic 3-D scenes usually involve hundreds or thousands of triangles. For example, one approach to building a simple sphere is to split the sphere into bands and then split each band into a faceted series of squares, as shown in the leftmost example in Figure 27-14. Each square then requires two triangles.

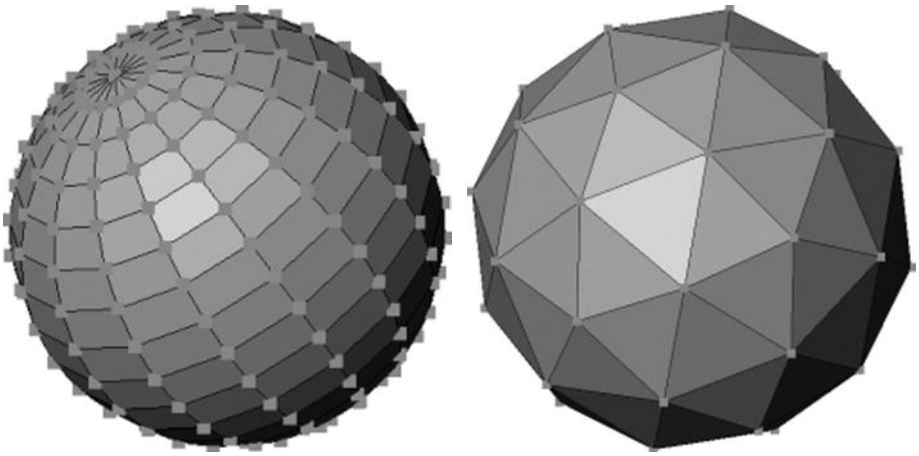

Figure 27-14. *Two ways to model a basic sphere*

To build this sort of nontrivial mesh, you need to construct it in code or use a dedicated 3-D modeling program. The code-only approach requires significant math. The design approach requires a sophisticated 3-D design application.

Fortunately, there are several tools for building 3-D scenes that you can use in WPF applications. Here are a few:

- Blender is an open source toolkit for 3-D modeling. It's available at `www.blender.org`, and there's an experimental XAML export script at `http://xamlexporter.codeplex.com`. Taken together, these provide a sophisticated and completely free platform for building 3-D content for WPF applications.
- Export plug-ins are available for a range of professional 3-D modeling programs such as Autodesk Maya and Newtek's LightWave. For a list of some, check out `http://tinyurl.com/bumqt2y`.

All 3-D modeling programs include basic primitives, such as the sphere, that are built out of smaller triangles. You can then use these primitives to construct a scene. 3-D modeling programs also let you add and position your light sources and apply textures. Some, such as Electric Rain ZAM 3D, also allow you to define animations you want to perform on the objects in your 3-D scene.

Model3DGroup Collections

When working with complex 3-D scenes, you'll usually need to arrange multiple objects. As you already know, a Viewport3D can hold multiple Visual3D objects, each of which uses a different mesh. However, this isn't the best way to build a 3-D scene. You'll get far better performance by creating as few meshes as possible and combining as much content as possible into each mesh.

Obviously, there's another consideration: flexibility. If your scene is broken down into separate objects, you have the ability to hit test, transform, and animate these pieces individually. However, you don't need to create distinct Visual3D objects to get this flexibility. Instead, you can use the Model3DGroup class to place several meshes in a single Visual3D.

Model3DGroup derives from Model3D (as do the GeometryModel3D and Light classes). However, it's designed to group together a combination of meshes. Each mess remains a distinct piece of your scene that you can manipulate individually.

For example, consider the 3-D character shown in Figure 27-15. This character was created in ZAM 3D and exported to XAML. His individual body parts—head, torso, belt, arm, and so on—are separate meshes grouped into a single Model3DGroup object.

Figure 27-15. *A 3-D character*

The following is a portion of the markup, which draws the appropriate meshes from a resource dictionary:

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <Model3DGroup x:Name="Scene" Transform="{DynamicResource SceneTR20}">
      <AmbientLight ... />
      <DirectionalLight ... />
      <DirectionalLight ... />
      <Model3DGroup x:Name="CharacterOR22">
        <Model3DGroup x:Name="PelvisOR24">
          <Model3DGroup x:Name="BeltOR26">
            <GeometryModel3D x:Name="BeltOR26GR27"
             Geometry="{DynamicResource BeltOR26GR27}"
             Material="{DynamicResource ER_Vector___Flat_Orange___DarkMR10}"
             BackMaterial="{DynamicResource ER_Vector___Flat_Orange___DarkMR10}" />
          </Model3DGroup>
          <Model3DGroup x:Name="TorsoOR29">
            <Model3DGroup x:Name="TubesOR31">
              <GeometryModel3D x:Name="TubesOR31GR32"
               Geometry="{DynamicResource TubesOR31GR32}"
               Material="{DynamicResource ER___Default_MaterialMR1}"
               BackMaterial="{DynamicResource ER___Default_MaterialMR1}"/>
            </Model3DGroup>
            ...

  </ModelVisual3D.Content>
</ModelVisual3D>
```

The entire scene is defined in a single ModelVisual3D, which contains a Model3DGroup. That Model3DGroup contains other nested Model3DGroup objects. For example, the top-level Model3DGroup contains the lights and the character, while the Model3DGroup for the character contains another Model3DGroup that contains the torso, and that Model3DGroup contains details such as the arms, which contain the palms, which contain the thumbs, and so on, leading eventually to the GeometryModel3D objects that actually define the objects and their material. As a result of this carefully segmented, nested design (which is implicit in the way you create these objects in a design tool such as ZAM 3D), you can animate these body parts individually, making the character walk, gesture, and so on. (You'll take a look at animating 3-D content a bit later in this chapter in the "Interactivity and Animations" section.)

■ **Note** Remember, the lowest overhead is achieved by using the fewest number of meshes and the fewest number of ModelVisual3D objects. The Model3DGroup allows you to reduce the number of ModelVisual3D objects you use (there's no reason to have more than one) while retaining the flexibility to manipulate parts of your scene separately.

Materials Revisited

So far, you've used just one of the types of material that WPF supports for constructing 3-D objects. The DiffuseMaterial is by far the most useful material type—it scatters light in all directions, like a real-world object.

When you create a DiffuseMaterial, you supply a Brush. So far, the examples you've seen have used solid-color brushes. However, the color you see is determined by the brush color and the lighting. If you have direct, full-strength lighting, you'll see the exact brush color. But if your lighting hits a surface at an angle (as in the previous triangle and cube examples), you'll see a darker, shaded color.

■ **Note** Interestingly, WPF does allow you to make partially transparent 3-D objects. The easiest approach is to set the Opacity property of the brush that you use with the material to a value less than 1.

The SpecularMaterial and EmissiveMaterial types work a bit differently. Both are additively blended into any content that appears underneath. For that reason, the most common way to use both types of material is in conjunction with a DiffuseMaterial.

Consider the SpecularMaterial. It reflects light much more sharply than DiffuseMaterial. You can control how sharply the light is reflected by using the SpecularPower property. Use a low number, and light is reflected more readily, no matter at what angle it strikes the surface. Use a higher number, and direct light is favored more strongly. Thus, a low SpecularPower produces a washed out, shiny effect, while a high SpecularPower produces sharply defined highlights.

On its own, placing a SpecularMaterial over a dark surface creates a glasslike effect. However, SpecularMaterial is more commonly used to add highlights to a DiffuseMaterial. For example, using a white SpecularMaterial on top of a DiffuseMaterial creates a plastic-like surface, while a darker SpecularMaterial and DiffuseMaterial produce a more metallic effect. Figure 27-16 shows two versions of a torus (a 3-D ring). The version on the left uses an ordinary DiffuseMaterial. The version on the right adds a SpecularMaterial on top. The highlights appear in several places because the scene includes two directional lights that are pointed in different directions.

To combine two surfaces, you need to wrap them in a MaterialGroup. Here's the markup that creates the highlights shown in Figure 27-16:

```
<GeometryModel3D>
  <GeometryModel3D.Material>
    <MaterialGroup>
      <DiffuseMaterial>
        <DiffuseMaterial.Brush>
          <SolidColorBrush Color="DarkBlue" />
        </DiffuseMaterial.Brush>
      </DiffuseMaterial>
      <SpecularMaterial SpecularPower="24">
        <SpecularMaterial.Brush>
          <SolidColorBrush Color="LightBlue" />
      </SpecularMaterial.Brush>
    </SpecularMaterial>
  </GeometryModel3D.Material>

  <GeometryModel3D.Geometry>...</GeometryModel3D.Geometry>
<GeometryModel3D>
```

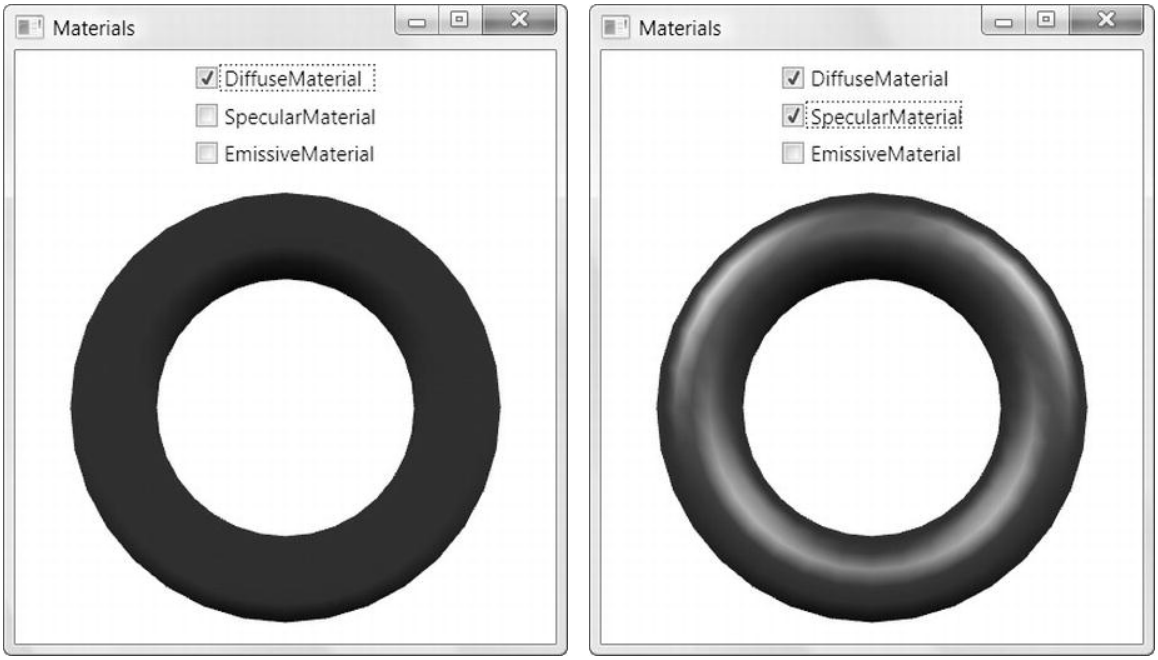

Figure 27-16. *Adding a SpecularMaterial*

■ **Note** If you place a SpecularMaterial or an EmissiveMaterial on a white surface, you won't see anything at all. That's because the SpecularMaterial and EmissiveMaterial contribute their color additively, and the color white is already maxed out with the maximum possible red, green, and blue contributions. To see the full effect of SpecularMaterial or EmissiveMaterial, place them on a black surface (or use them over a black DiffuseMaterial).

The EmissiveMaterial is stranger still. It emits light, which means that a green EmissiveMaterial that's displayed over a dark surface shows up as a flat green silhouette, regardless of whether your scene includes any light sources.

Once again, you can get a more interesting effect by layering an EmissiveMaterial over a DiffuseMaterial. Because of the additive nature of EmissiveMaterial, the colors are blended. For example, if you place a red EmissiveMaterial over a blue DiffuseMaterial, your shape will acquire a purple tinge. The EmissiveMaterial will contribute the same amount of red over the entire surface of the shape, while the DiffuseMaterial will be shaded according to the light sources in your scene.

■ **Tip** The light "radiated" from an EmissiveMaterial doesn't reach other objects. To create the effect of a glowing object that illuminates other nearby objects, you may want to place a light source (such as PointLight) near your EmissiveMaterial.

Texture Mapping

So far, you've used the SolidColorBrush to paint your objects. However, WPF allows you to paint a DiffuseMaterial object by using any brush. That means you can paint it with gradients (LinearGradientBrush and RadialGradientBrush), vector or bitmap images (ImageBrush), or the content from a 2-D element (VisualBrush).

There's one catch. When you use anything other than a SolidColorBrush, you need to supply additional information that tells WPF how to map the 2-D content of the brush onto the 3-D surface you're painting. You supply this information by using the MeshGeometry.TextureCoordinates collection. Depending on your choice, you can tile the brush content, extract just a part of it, and stretch, warp, and otherwise mangle it to fit curved and angular surfaces.

So how does the TextureCoordinates collection work? The basic idea is that each coordinate in your mesh needs a corresponding point in TextureCoordinates. The coordinate in the mesh is a point in 3-D space, while the point in the TextureCoordinates collection is a 2-D point because the content of a brush is always 2-D. The following sections show you how to use texture mapping to display image and video content on a 3-D shape.

Mapping the ImageBrush

The easiest way to understand how TextureCoordinates work is to use an ImageBrush that allows you to paint a bitmap. Here's an example that uses a misty scene of a tree at dawn:

```
<GeometryModel3D.Material>
  <DiffuseMaterial>
    <DiffuseMaterial.Brush>
      <ImageBrush ImageSource="Tree.jpg"></ImageBrush>
    </DiffuseMaterial.Brush>
  </DiffuseMaterial>
</GeometryModel3D.Material>
```

In this example, the ImageBrush is used to paint the content of the cube you created earlier. Depending on the TextureCoordinates you choose, you could stretch the image, wrapping it over the entire cube, or you could put a separate copy of it on each face (as we do in this example). Figure 27-17 shows the end result.

■ **Note** This example adds one extra detail. It uses a Slider at the bottom of the window that allows the user to rotate the cube, viewing it from all angles. This is made possible by a transform, as you'll learn in the next section.

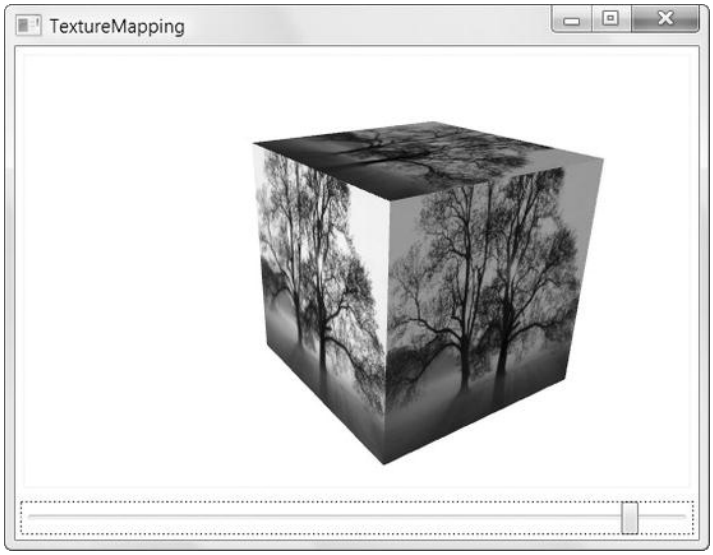

Figure 27-17. *A textured cube*

Initially, the TextureCoordinates collection is empty and your image won't appear on the 3-D surface. To get started with the cube example, you may want to concentrate on mapping just a single face. In the current example, the cube is oriented so that its left side is facing the camera. Here is the mesh for the cube. The two triangles that make up the left (front-facing) side are in bold:

```
<MeshGeometry3D
  Positions="0,0,0   10,0,0   0,10,0   10,10,0
             0,0,0   0,0,10   0,10,0   0,10,10
             0,0,0   10,0,0   0,0,10   10,0,10
             10,0,0  10,10,10 10,0,10   10,10,0
             0,0,10  10,0,10  0,10,10   10,10,10
             0,10,0  0,10,10  10,10,0   10,10,10"
  TriangleIndices="
             0,2,1     1,2,3
             4,5,6     6,5,7
             8,9,10    9,11,10
             12,13,14  12,15,13
             16,17,18  19,18,17
             20,21,22  22,21,23"  />
```

Most of the mesh points aren't mapped at all. In fact, the only points that are mapped are these four, which define the face of the cube that's oriented toward the camera:

```
(0,0,0) (0,0,10) (0,10,0) (0,10,10)
```

Because this is actually a flat surface, mapping is relatively easy. You can choose a set of TextureCoordinates for this face by removing the dimension that has a value of 0 in all four points. (In this example, that's the X coordinate because the visible face is on the left side of the cube.)

Here's the TextureCoordinates that fill this requirement:

```
(0,0) (0,10) (10,0) (10,10)
```

The TextureCoordinates collection uses relative coordinates. To keep things simple, you may want to use 1 to indicate the maximum value. In this example, that transformation is easy:

```
(0,0) (0,1) (1,0) (1,1)
```

This set of TextureCoordinates essentially tells WPF to take the point (0, 0) at the bottom left of the rectangle that represents the brush content, and map that to the corresponding point (0, 0, 0) in 3-D space. Similarly, take the bottom-right corner (0, 1) and map that to (0, 0, 10), make the top-left corner (1, 0) map to (0, 10, 0), and make the top-right corner (1, 1) map to (0, 10, 10).

Here's the cube mesh that uses this texture mapping. All the other coordinates in the Positions collection are mapped to (0, 0), so that the texture is not applied to these areas:

```
<MeshGeometry3D
  Positions="0,0,0   10,0,0   0,10,0   10,10,0
             0,0,0   0,0,10   0,10,0   0,10,10
             0,0,0   10,0,0   0,0,10   10,0,10
             10,0,0  10,10,10 10,0,10  10,10,0
             0,0,10  10,0,10  0,10,10  10,10,10
             0,10,0  0,10,10  10,10,0  10,10,10"
  TriangleIndices="..."
  TextureCoordinates="
             0,0  0,0  0,0  0,0
             0,0  0,1  1,0  1,1
             0,0  0,0  0,0  0,0
             0,0  0,0  0,0  0,0
             0,0  0,0  0,0  0,0
             0,0  0,0  0,0  0,0"  />
```

This markup maps the texture to a single face on the cube. Although it is mapped successfully, the image is turned on its side. To get a top-up image, you need to rearrange your coordinates to use this order:

```
1,1 0,1 1,0 0,0
```

You can extend this process to map each face of the cube. Here's a set of TextureCoordinates that does exactly that and creates the multifaceted cube shown in Figure 27-17:

```
TextureCoordinates="0,0 0,1 1,0 1,1
                    1,1 0,1 1,0 0,0
                    0,0 1,0 0,1 1,1
                    0,0 1,0 0,1 1,1
                    1,1 0,1 1,0 0,0
                    1,1 0,1 1,0 0,0"
```

There are obviously many more effects you can create by tweaking these points. For example, you could stretch your texture around a more complex object such as a sphere. Because the meshes required for this sort of object typically include hundreds of points, you won't fill the TextureCoordinates collection by hand. Instead, you'll rely on a 3-D modeling program (or a math-crunching code routine that does it at runtime). If you want to apply different brushes to different portions of your mesh, you'll need to split your 3-D object into multiple meshes, each of which will have a different material that uses a different brush. You can then combine those meshes into one Model3DGroup for the lowest overhead.

Video and the VisualBrush

Ordinary images aren't the only kind of content you can map to a 3-D surface. You can also map content that changes, such as gradient brushes that have animated values. One common technique in WPF is to map a video to a 3-D surface. As the video plays, its content is displayed in real time on the 3-D surface.

Achieving this somewhat overused effect is surprisingly easy. In fact, you can map a video brush to the faces of a cube, with different orientations, using the exact same set of TextureCoordinates you used in the previous example to map the image. All you need to do is replace the ImageBrush with a more capable VisualBrush and use a MediaElement for your visual. With the help of an event trigger, you can even start a looping playback of your video without requiring any code.

The following markup creates a VisualBrush that performs playback and rotates the cube at the same time, displaying its different axes. (You'll learn more about how you can use animation and rotation to achieve this effect in the next section.)

```
<GeometryModel3D.Material>
  <DiffuseMaterial>
    <DiffuseMaterial.Brush>
      <VisualBrush>
        <VisualBrush.Visual>
          <MediaElement>
            <MediaElement.Triggers>
              <EventTrigger RoutedEvent="MediaElement.Loaded">
                <EventTrigger.Actions>
                  <BeginStoryboard>
                    <Storyboard >
                      <MediaTimeline Source="test.mpg" />
                      <DoubleAnimation Storyboard.TargetName="rotate"
                        Storyboard.TargetProperty="Angle"
                        To="360" Duration="0:0:5" RepeatBehavior="Forever" />
                    </Storyboard>
                  </BeginStoryboard>
                </EventTrigger.Actions>
              </EventTrigger>
            </MediaElement.Triggers>
          </MediaElement>
        </VisualBrush.Visual>
      </VisualBrush>
    </DiffuseMaterial.Brush>
  </DiffuseMaterial>
</GeometryModel3D.Material>
```

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

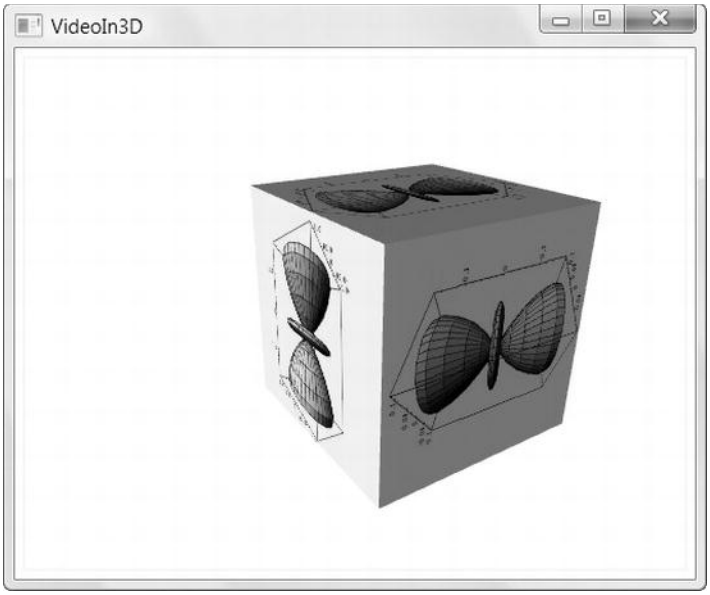

Figure 27-18. *Displaying video on several 3-D surfaces*

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 by 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 such as 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 by 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 data-bound 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 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.

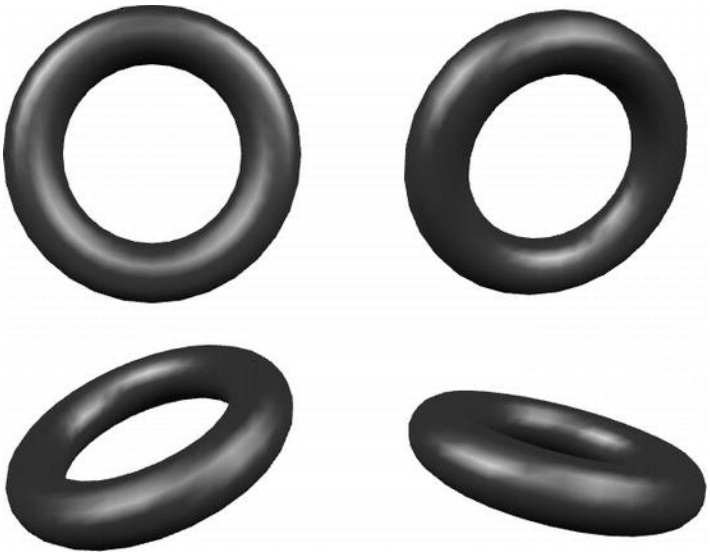

Figure 27-19. *A rotating 3-D shape*

A Flyover

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 flyover 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 by 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.

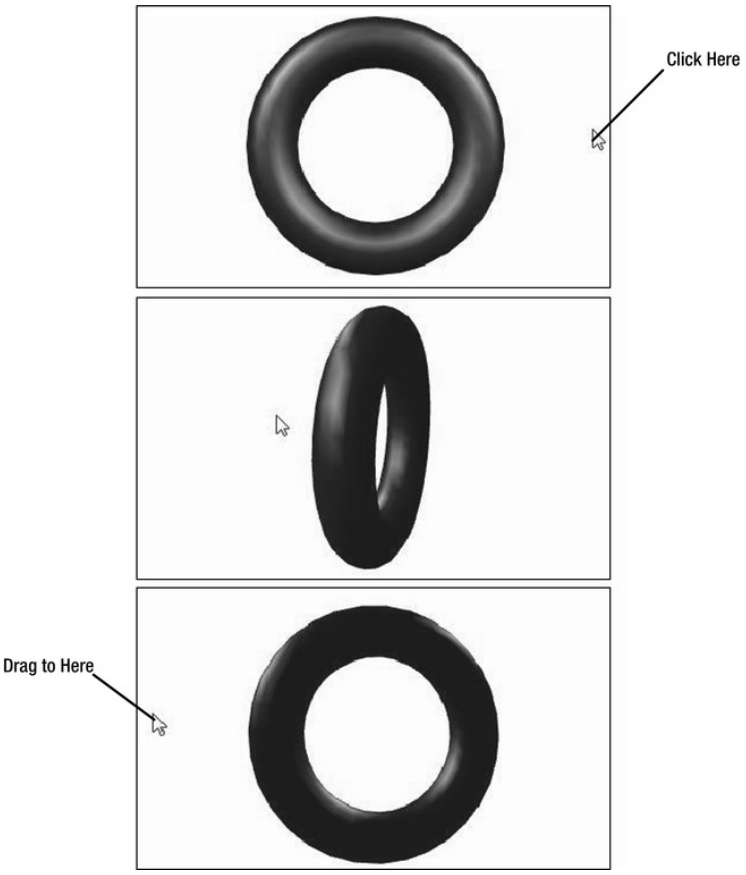

Figure 27-20. *Changing your viewpoint with the virtual trackball*

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. You can download the virtual trackball code with the 3-D tools projects described earlier at `http://3dtools.codeplex.com`.

■ **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>
```

After 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 by 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).

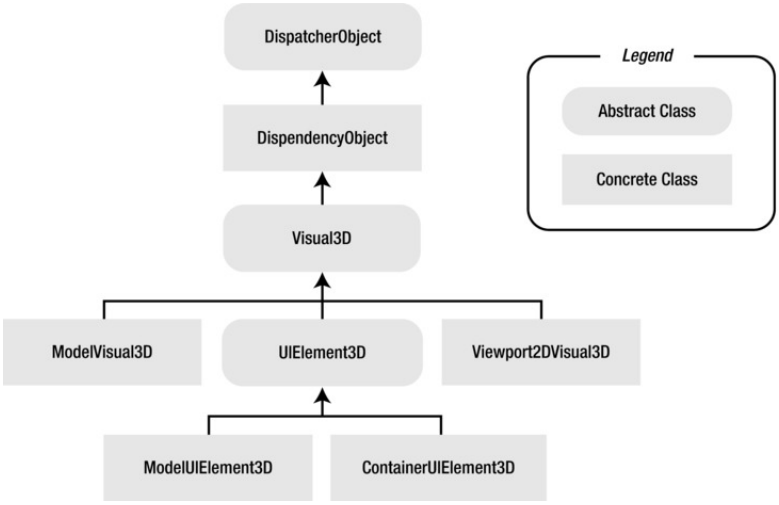

Figure 27-21. *The 3-D visual classes*

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.

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, anytime 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 by 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.

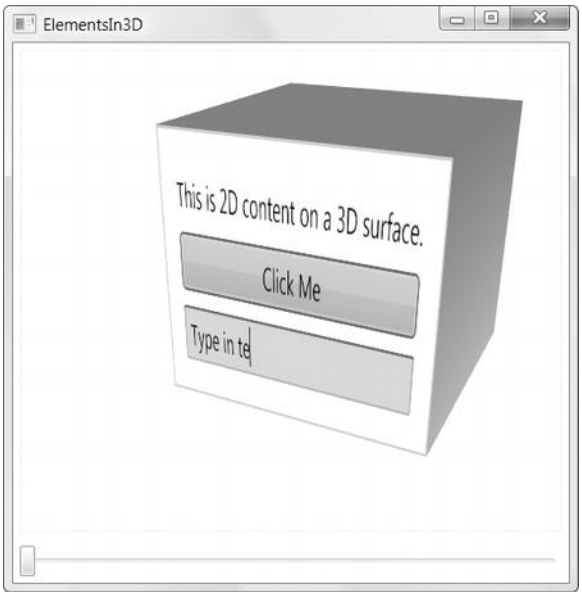

Figure 27-22. *Interactive WPF elements in 3-D*

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 by 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.

The Last Word

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 by using intense math, it's just as possible to export 3-D models from a design tool and manipulate them by 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 old but classic book *3D Math Primer for Graphics and Game Development* by Fletcher Dunn (Wordware Publishing, 2002).

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://3dtools.codeplex.com` 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://tinyurl.com/bumqt2y` 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 Autodesk 3ds Max) to XAML.
- `http://tinyurl.com/np295l` includes classes that wrap the meshes required for three common 3-D primitives: a cone, a sphere, and a cylinder.
- `http://tinyurl.com/97kwul2` provides a SandBox3D project that allows you to load simple 3-D meshes and manipulate them with transforms.