```
  <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 only give the effect you want 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 effect and for porting code from other frameworks (such as Direct3D) that use this type of camera.

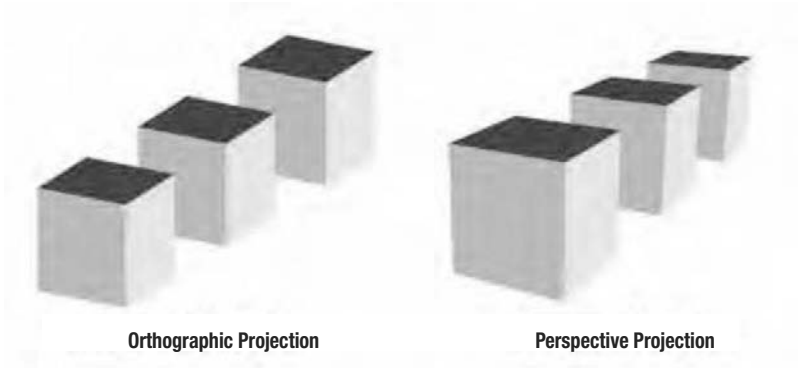

Figure 27-4. Perspective in different types of cameras

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 using the Position property, and then tilt it to survey the view 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 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 using this formula:

```
CameraLookDirection = CenterPointOfInterest - CameraPosition
```

In the triangle example, the camera is placed in the top-left corner 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.

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

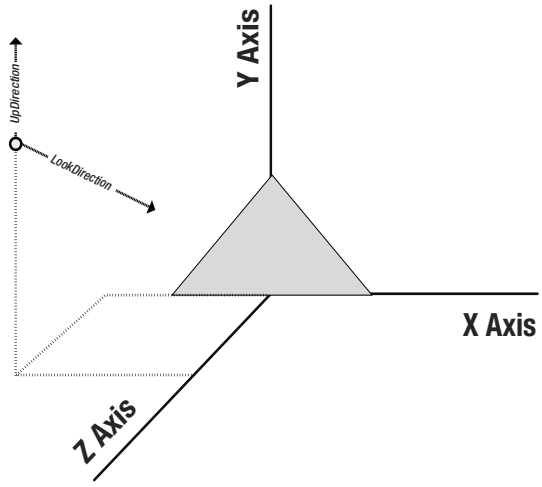

Figure 27-5. *Positioning and angling the camera*

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.

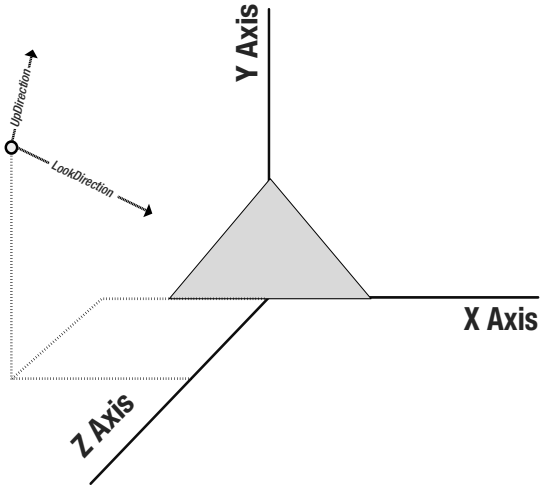

Figure 27-6. *Another way to angle the camera*

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.

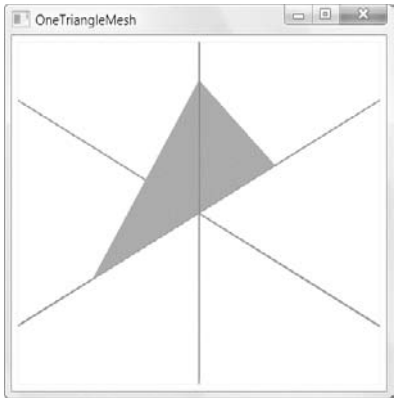

Figure 27-7. *A complete 3-D scene with one triangle*

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 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://www.codeplex.com/3DTools`. 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 only applies to 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, there are some cases where you'll need to change these values to prevent rendering artifacts. The most common example is 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.

Once 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.

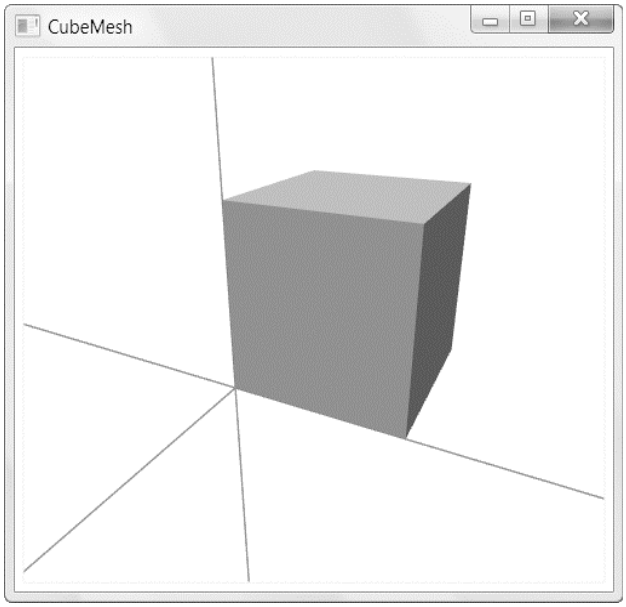

Figure 27-8. *A 3-D cube*

■ **Note** You'll notice that the edges of the cube in Figure 27-8 have smooth, anti-aliased edges. Unfortunately, if you're rendering 3-D on Windows XP you won't get this level of quality. Due to sketchy support in XP video drivers, WPF doesn't attempt to perform anti-aliasing with the edges of 3-D shapes, leaving them jagged.

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.

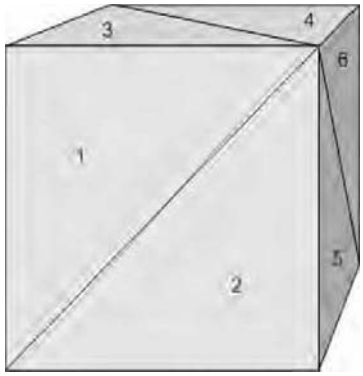

Figure 27-9. *Breaking the cube into triangles*

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.

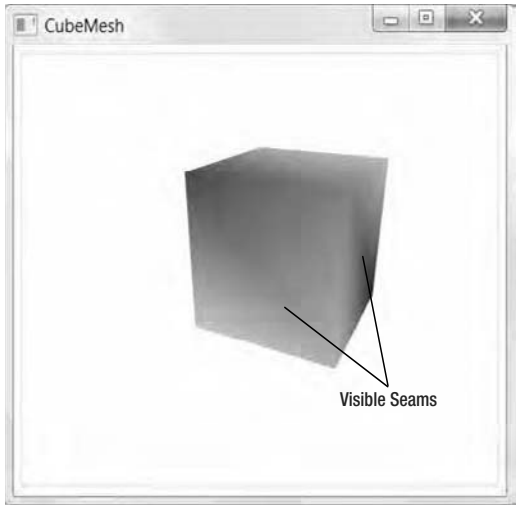

Figure 27-10. *A cube with lighting artifacts*

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 only pays attention to 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).

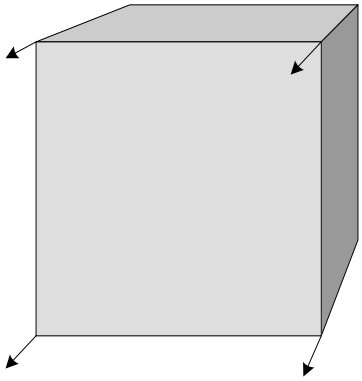

Figure 27-11. *Normals on the front side of a cube*

■ **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.

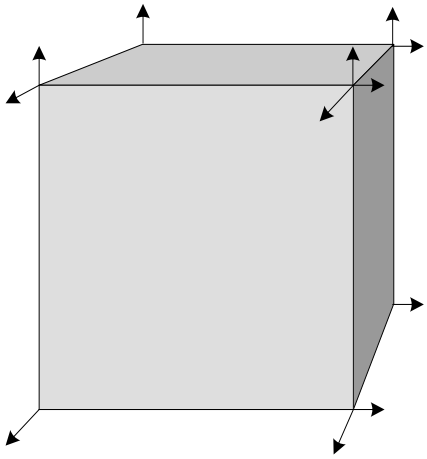

Figure 27-12. *Normals on the visible faces of a cube*

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 only defined on 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 different 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 smoothens 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.

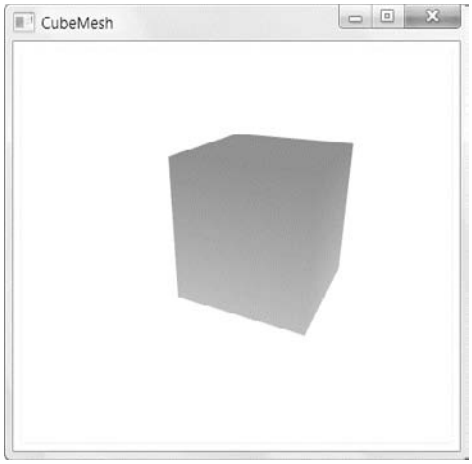

Figure 27-13. *An extremely smooth cube*

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.

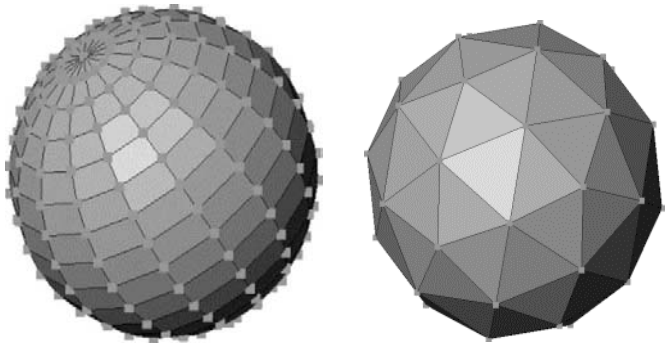

Figure 27-14. *Two ways to model a basic sphere*

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 plenty of tools for building 3-D scenes that you can use in WPF applications. Here are a few:

- ZAM 3D is a 3-D modeling tool designed explicitly for XAML. It's available at `http://www.erain.com/Products/ZAM3D`.
- Blender is an open source toolkit for 3-D modeling. It's available at `http://www.blender.org`, and there's an experimental XAML export script at `http://codeplex.com/xamlexporter`. Taken together, this provides a sophisticated and completely free platform for building 3-D content for WPF applications.
- Export plug-ins are beginning to appear for a range of professional 3-D modeling programs such as Maya and LightWave. For a list of some, check out `http://blogs.msdn.com/mswanson/articles/WPFToolsAndControls.aspx`.

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 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 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 overtop 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 overtop. 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>
```

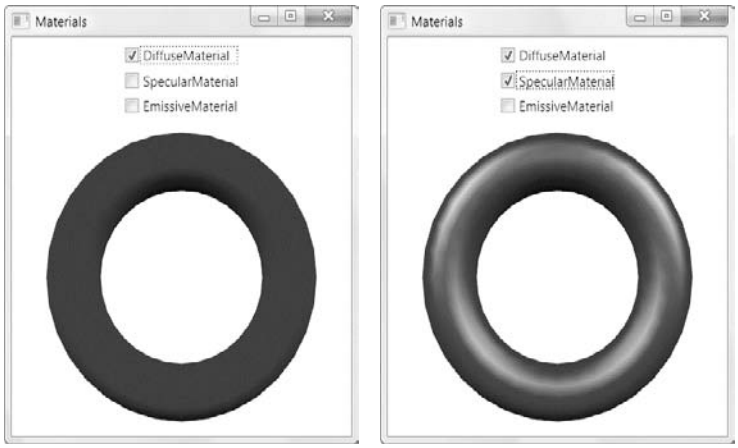

Figure 27-16. *Adding a SpecularMaterial*

■ **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 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 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.

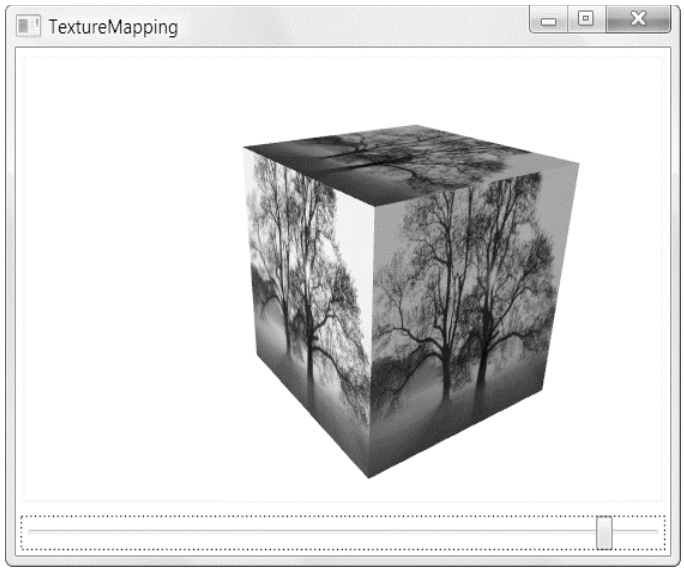

Figure 27-17. *A textured cube*

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 actually 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 like 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>
```