CHAPTER 13

# Geometries and Drawings

In the preceding chapter, you started your exploration of WPF's 2-D drawing features. You considered how you can use simple Shape-derived classes in combination with brushes and transforms to create a variety of graphical effects. However, the concepts you learned still fall far short of what you need to create (and manipulate) detailed 2-D scenes made up of vector art. That's because there's a wide gap between rectangles, ellipses, and polygons and the sort of clip art you see in graphically rich applications.

In this chapter, you'll extend your skills with a few more concepts. You'll learn how more-complex drawings are defined in WPF, how to model arcs and curves, and how you can convert existing vector art to the XAML format you need. You'll also consider the most efficient ways to work with complex images—in other words, how you can reduce the overhead involved in managing hundreds or thousands of shapes. This begins with replacing the simple shapes you learned about in the previous chapter with the more powerful Path class, which can wrap complex geometries.

## Paths and Geometries

In the previous chapter, you took a look at a number of classes that derive from Shape, including Rectangle, Ellipse, Polygon, and Polyline. However, there's one Shape-derived class that you haven't considered yet, and it's the most powerful by far. The Path class has the ability to encompass any simple shape, groups of shapes, and more-complex ingredients such as curves.

The Path class includes a single property, named Data, that accepts a Geometry object that defines the shape (or shapes) the path includes. You can't create a Geometry object directly, because it's an abstract class. Instead, you need to use one of the seven derived classes listed in Table 13-1.

***Table 13-1.*** *Geometry Classes*

| Name | Description |
|---|---|
| LineGeometry | Represents a straight line. The geometry equivalent of the Line shape. |
| RectangleGeometry | Represents a rectangle (optionally with rounded corners). The geometry equivalent of the Rectangle shape. |
| EllipseGeometry | Represents an ellipse. The geometry equivalent of the Ellipse shape. |
| GeometryGroup | Adds any number of Geometry objects to a single path, using the EvenOdd or Nonzero fill rule to determine what regions to fill. |
| CombinedGeometry | Merges two geometries into one shape. The CombineMode property allows you to choose how the two are combined. |

| Name | Description |
|---|---|
| PathGeometry | Represents a more complex figure that's composed of arcs, curves, and lines, and can be open or closed. |
| StreamGeometry | A read-only lightweight equivalent to PathGeometry. StreamGeometry saves memory because it doesn't hold the individual segments of your path in memory all at once. However, it can't be modified after it has been created. |

At this point, you might be wondering what the difference is between a path and a geometry. The geometry *defines* a shape. A path allows you to *draw* the shape. Thus, the Geometry object defines details such as the coordinates and size of your shape, and the Path object supplies the Stroke and Fill brushes you'll use to paint it. The Path class also includes the features it inherits from the UIElement infrastructure, such as mouse and keyboard handling.

However, the geometry classes aren't quite as simple as they seem. For one thing, they all inherit from Freezable (through the base Geometry class), which gives them support for change notification. As a result, if you use a geometry to create a path, and then modify the geometry after the fact, your path will be redrawn automatically. The geometry classes can also be used to define drawings that you can apply through a brush, which gives you an easy way to paint complex content that doesn't need the user-interactivity features of the Path class. You'll consider this ability in the "Drawings" section later in this chapter.

In the following sections, you'll explore all the classes that derive from Geometry.

## Line, Rectangle, and Ellipse Geometries

The LineGeometry, RectangleGeometry, and EllipseGeometry classes map directly to the Line, Rectangle, and Ellipse shapes that you learned about in Chapter 12. For example, you can convert this markup that uses the Rectangle element:

```
<Rectangle Fill="Yellow" Stroke="Blue"
  Width="100" Height="50" ></Rectangle>
```

to this markup that uses the Path element:

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <RectangleGeometry Rect="0,0 100,50"></RectangleGeometry>
  </Path.Data>
</Path>
```

The only real difference is that the Rectangle shape takes Height and Width values, while the RectangleGeometry takes four numbers that describe the size *and* location of the rectangle. The first two numbers describe the X and Y coordinates point where the top-left corner will be placed, and the last two numbers set the width and height of the rectangle. You can start the rectangle out at (0, 0) to get the same effect as an ordinary Rectangle element, or you can offset the rectangle by using different values. The RectangleGeometry class also includes the RadiusX and RadiusY properties, which let you round the corners (as described in the preceding chapter).

Similarly, you can convert the following Line:

```
<Line Stroke="Blue" X1="0" Y1="0" X2="10" Y2="100"></Line>
```

to this LineGeometry:

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <LineGeometry StartPoint="0,0" EndPoint="10,100"></LineGeometry>
  </Path.Data>
</Path>
```

and you can convert an Ellipse like this:

```
<Ellipse Fill="Yellow" Stroke="Blue"
  Width="100" Height="50" HorizontalAlignment="Left"></Ellipse>
```

to this EllipseGeometry:

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <EllipseGeometry RadiusX="50" RadiusY="25" Center="50,25"></EllipseGeometry>
  </Path.Data>
</Path>
```

Notice that the two radius values are simply half of the width and height values. You can also use the Center property to offset the location of the ellipse. In this example, the center is placed in the exact middle of the ellipse bounding box, so that it's drawn in exactly the same way as the Ellipse shape.

Overall, these simple geometries work in the same way as the corresponding shapes. You get the added ability to offset rectangles and ellipses, but that's not necessary if you're placing your shapes on a Canvas, which already gives you the ability to position your shapes at a specific point In fact, if this were all you could do with geometries, you probably wouldn't bother to use the Path element. The difference appears when you decide to combine more than one geometry in the same path, as described in the next section.

## Combining Shapes with GeometryGroup

The simplest way to combine geometries is to use the GeometryGroup, and nest the other Geometry-derived objects inside. Here's an example that places an ellipse next to a square:

```
<Path Fill="Yellow" Stroke="Blue" Margin="5" Canvas.Top="10" Canvas.Left="10" >
  <Path.Data>
    <GeometryGroup>
      <RectangleGeometry Rect="0,0 100,100"></RectangleGeometry>
      <EllipseGeometry Center="150,50" RadiusX="35" RadiusY="25"></EllipseGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>
```

The effect of this markup is the same as if you had supplied two Path elements, one with the RectangleGeometry and one with the EllipseGeometry (and that's the same as if you had used a Rectangle and Ellipse shape instead). However, there's one advantage to this approach. You've replaced two elements with one, which means you've reduced the overhead of your user interface. In general, a window that uses a smaller number of elements with more-complex geometries will perform faster than a window that has a large number of elements with simpler geometries. This effect won't be apparent in a window that has just a few dozen shapes, but it may become significant in one that requires hundreds or thousands.

Of course, there's also a drawback to combining geometries in a single Path element: you won't be able to perform event handling of the different shapes separately. Instead, the Path element will fire all mouse events. However, you can still manipulate the nested RectangleGeometry and EllipseGeometry objects independently to change the overall path. For example, each geometry provides a Transform property, which you can set to stretch, skew, or rotate that part of the path.

Another advantage of geometries is that you can reuse the same geometry in several separate Path elements. No code is necessary—you simply need to define the geometry in a Resources collection and refer to it in your path with the StaticExtension or DynamicExtension markup extensions. Here's an example that rewrites the markup shown previously to show instances of the CombinedGeometry, at two locations on a Canvas and with two fill colors:

```
<Window.Resources>
  <GeometryGroup x:Key="Geometry">
    <RectangleGeometry Rect="0 ,0 100 ,100"></RectangleGeometry>
    <EllipseGeometry Center="150, 50" RadiusX="35" RadiusY="25"></EllipseGeometry>
  </GeometryGroup>
</Window.Resources>

<Canvas>
  <Path Fill="Yellow" Stroke="Blue" Margin="5" Canvas.Top="10" Canvas.Left="10"
   Data="{StaticResource Geometry}">
  </Path>
  <Path Fill="Green" Stroke="Blue" Margin="5" Canvas.Top="150" Canvas.Left="10"
   Data="{StaticResource Geometry}">
  </Path>
</Canvas>
```

The GeometryGroup becomes more interesting when your shapes intersect. Rather than simply treating your drawing as a combination of solid shapes, the GeometryGroup uses its FillRule property (which can be EvenOdd or Nonzero, as described in Chapter 12) to decide which shapes to fill. Consider what happens if you alter the markup shown earlier like this, placing the ellipse over the square:

```
<Path Fill="Yellow" Stroke="Blue" Margin="5" Canvas.Top="10" Canvas.Left="10" >
  <Path.Data>
    <GeometryGroup>
      <RectangleGeometry Rect="0,0 100,100"></RectangleGeometry>
      <EllipseGeometry Center="50,50" RadiusX="35" RadiusY="25"></EllipseGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>
```

This markup creates a square with an ellipse-shaped hole in it. If you change FillRule to Nonzero, you'll get a solid ellipse over a solid rectangle, both with the same yellow fill.

You could create the square-with-a-hole effect by simply superimposing a white-filled ellipse over your square. However, the GeometryGroup class becomes more useful if you have content underneath, which is typical in a complex drawing. Because the ellipse is treated as a hole in your shape (not another shape with a different fill), any content underneath shows through. For example, if you add this line of text:

```
<TextBlock Canvas.Top="50" Canvas.Left="20" FontSize="25" FontWeight="Bold">
  Hello There</TextBlock>
```

you'll get the result shown in Figure 13-1.

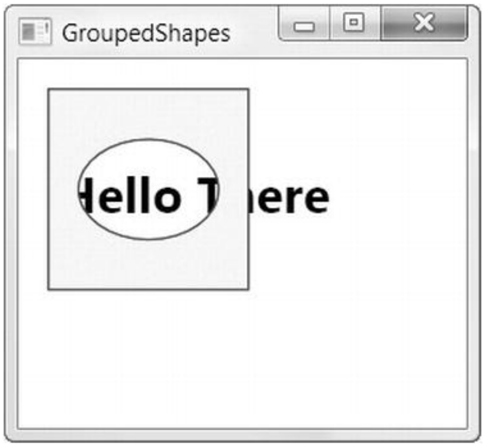

***Figure 13-1.*** *A path that uses two shapes*

---

■ **Note** Remember that objects are drawn in the order they are processed. In other words, if you want the text to appear underneath your shape, make sure you add the TextBlock to your markup before the Path element. Or if you're using a Canvas or Grid to hold your content, you can set the attached Panel.ZIndex property on your elements to place them explicitly, as described in Chapter 3.

---

## Fusing Geometries with CombinedGeometry

The GeometryGroup class is an invaluable tool for building complex shapes out of the basic primitives (rectangle, ellipse, and line). However, it has obvious limitations. It works great for creating a shape by drawing one shape and "subtracting" other shapes from inside. However, it's difficult to get the result you want if the shape borders intersect one another, and it's no help if you want to remove part of a shape.

The CombinedGeometry class is tailor-made for combining shapes that overlap, and where neither shape contains the other completely. Unlike GeometryGroup, CombinedGeometry takes just two geometries, which you supply by using the Geometry1 and Geometry2 properties. CombinedGeometry doesn't include the FillRule property. Instead, it has the much more powerful GeometryCombineMode property, which takes one of four values, as described in Table 13-2.

***Table 13-2.*** *Values from the GeometryCombineMode Enumeration*

| Name | Description |
|---|---|
| Union | Creates a shape that includes all the areas of the two geometries. |
| Intersect | Creates a shape that contains the area that's shared between the two geometries. |
| Xor | Creates a shape that contains the area that isn't shared between the two geometries. In other words, it's as if the shapes were combined (using a Union) and then the shared part (the Intersect) were removed. |
| Exclude | Creates a shape that includes all the area from the first geometry, not including the area that's in the second geometry. |

For example, here's how you can merge two shapes to create one shape with the total area by using GeometryCombineMode.Union:

```
<Path Fill="Yellow" Stroke="Blue" Margin="5">
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <RectangleGeometry Rect="0,0 100,100"></RectangleGeometry>
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <EllipseGeometry Center="85,50" RadiusX="65" RadiusY="35"></EllipseGeometry>
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

Figure 13-2 shows this shape, as well as the result of combining the same shapes in every other way possible.

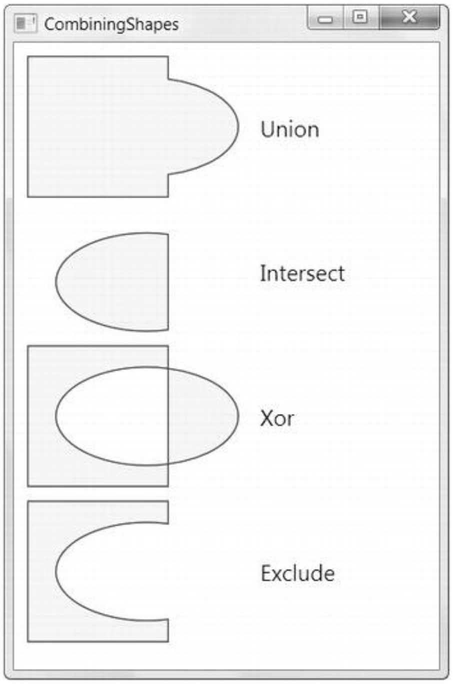

***Figure 13-2.*** *Combining shapes*

The fact that a CombinedGeometry can combine only two shapes may seem like a significant limitation, but it's not. You can build a shape that involves dozens of distinct geometries or more—you simply need to use nested CombinedGeometry objects. For example, one CombinedGeometry object might combine two other CombinedGeometry objects, which themselves can combine more geometries. Using this technique, you can build up detailed shapes.

To understand how this works, consider the simple "no" sign (a circle with a slash through it) shown in Figure 13-3. Although there isn't any WPF primitive that resembles this shape, you can assemble it quite quickly using CombinedGeometry objects.

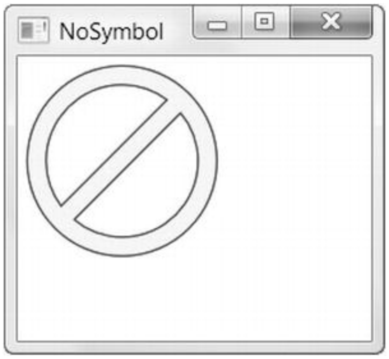

***Figure 13-3.*** *Several combined shapes*

It makes sense to start by drawing the ellipse that represents the outer edge of the shape. Then, using a CombinedGeometry with the GeometryCombineMode.Exclude, you can remove a smaller ellipse from the inside. Here's the markup that you need:

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Exclude">
      <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="50,50" RadiusX="50" RadiusY="50"></EllipseGeometry>
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <EllipseGeometry Center="50,50" RadiusX="40" RadiusY="40"></EllipseGeometry>
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

This gets you part of the way, but you still need the slash through the middle. The easiest way to add this element is to use a rectangle that's tilted to the side. You can accomplish this using the RectangleGeometry with a RotateTransform of 45 degrees:

```
<RectangleGeometry Rect="44,5 10,90">
  <RectangleGeometry.Transform>
    <RotateTransform Angle="45" CenterX="50" CenterY="50"></RotateTransform>
  </RectangleGeometry.Transform>
</RectangleGeometry>
```

■ **Note** When applying a transform to a geometry, you use the Transform property (not RenderTransform or LayoutTransform). That's because the geometry defines the shape, and any transforms are always applied before the path is used in your layout.

The final step is to combine this geometry with the combined geometry that created the hollow circle. In this case, you need to use GeometryCombineMode.Union to add the rectangle to your shape.

Here's the complete markup for the symbol:

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <CombinedGeometry GeometryCombineMode="Exclude">
          <CombinedGeometry.Geometry1>
            <EllipseGeometry Center="50,50"
             RadiusX="50" RadiusY="50"></EllipseGeometry>
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry Center="50,50"
             RadiusX="40" RadiusY="40"></EllipseGeometry>
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
      </CombinedGeometry.Geometry1>

      <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="44,5 10,90">
          <RectangleGeometry.Transform>
            <RotateTransform Angle="45" CenterX="50" CenterY="50"></RotateTransform>
          </RectangleGeometry.Transform>
        </RectangleGeometry>
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

■ **Note** A GeometryGroup object can't influence the fill or stroke brushes used to color your shape. These details are set by the path. Therefore, you need to create separate Path objects if you want to color parts of your path differently.

## Drawing Curves and Lines with PathGeometry

PathGeometry is the superpower of geometries. It can draw anything that the other geometries can, and much more. The only drawback is a lengthier (and somewhat more complex) syntax.

Every PathGeometry object is built out of one or more PathFigure objects (which are stored in the PathGeometry.Figures collection). Each PathFigure is a continuous set of connected lines and curves that can be closed or open. The figure is closed if the end of the last line in the figure connects to the beginning of the first line.

The PathFigure class has four key properties, as described in Table 13-3.

***Table 13-3.*** *PathFigure Properties*

| Name | Description |
|---|---|
| StartPoint | This is a point that indicates where the line for the figure begins. |
| Segments | This is a collection of PathSegment objects that are used to draw the figure. |
| IsClosed | If true, WPF adds a straight line to connect the starting and ending points (if they aren't the same). |
| IsFilled | If true, the area inside the figure is filled in using the Path.Fill brush. |

So far, this all sounds fairly straightforward. The PathFigure is a shape that's drawn using an unbroken line that consists of a number of segments. However, the trick is that there are several type of segments, all of which derive from the PathSegment class. Some are simple, like the LineSegment that draws a straight line. Others, like the BezierSegment, draw curves and are correspondingly more complex.

You can mix and match different segments freely to build your figure. Table 13-4 lists the segment classes you can use.

***Table 13-4.*** *PathSegment Classes*

| Name | Description |
|---|---|
| LineSegment | Creates a straight line between two points. |
| ArcSegment | Creates an elliptical arc between two points. |
| BezierSegment | Creates a Bézier curve between two points. |
| QuadraticBezierSegment | Creates a simpler form of Bézier curve that has one control point instead of two, and is faster to calculate. |
| PolyLineSegment | Creates a series of straight lines. You can get the same effect by using multiple LineSegment objects, but a single PolyLineSegment is more concise. |
| PolyBezierSegment | Creates a series of Bézier curves. |
| PolyQuadraticBezierSegment | Creates a series of simpler quadratic Bézier curves. |

## Straight Lines

It's easy enough to create simple lines by using the LineSegment and PathGeometry classes. You simply set the StartPoint and add one LineSegment for each section of the line. The LineSegment.Point property identifies the end point of each segment.

For example, the following markup begins at (10, 100), draws a straight line to (100, 100), and then draws a line from that point to (100, 50). Because the PathFigure.IsClosed property is set to true, a final line segment is added, connecting (100, 50) to (0, 0). The final result is a right-angled triangle.

```
<Path Stroke="Blue">
  <Path.Data>
    <PathGeometry>
      <PathFigure IsClosed="True" StartPoint="10,100">
        <LineSegment Point="100,100" />
        <LineSegment Point="100,50" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

---

■ **Note** Remember that each PathGeometry can contain an unlimited number of PathFigure objects. That means you can create several separate open or closed figures that are all considered part of the same path.

---

## Arcs

Arcs are a little more interesting than straight lines. You identify the end point of the line by using the ArcSegment.Point property, just as you would with a LineSegment. However, the PathFigure draws a curved line from the starting point (or the end point of the previous segment) to the end point of your arc. This curved connecting line is actually a portion of the edge of an ellipse.

Obviously, the end point doesn't provide enough information to draw the arc, because many curves (some gentle, some more extreme) could connect two points. You also need to indicate the size of the imaginary ellipse that's being used to draw the arc. You do this by using the ArcSegment.Size property, which supplies the X radius and the Y radius of the ellipse. The larger the ellipse size of the imaginary ellipse, the more gradually its edge curves.

---

■ **Note** For any two points, there is a practical maximum and minimum size for the ellipse. The maximum occurs when you create an ellipse so large that the line segment you're drawing appears straight. Increasing the size beyond this point has no effect. The minimum occurs when the ellipse is small enough that a full semicircle connects the two points. Shrinking the size beyond this point also has no effect.

---

Here's an example that creates the gentle arc shown in Figure 13-4:

```
<Path Stroke="Blue" StrokeThickness="3">
  <Path.Data>
    <PathGeometry>
      <PathFigure IsClosed="False" StartPoint="10,100" >
        <ArcSegment Point="250,150" Size="200,300" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

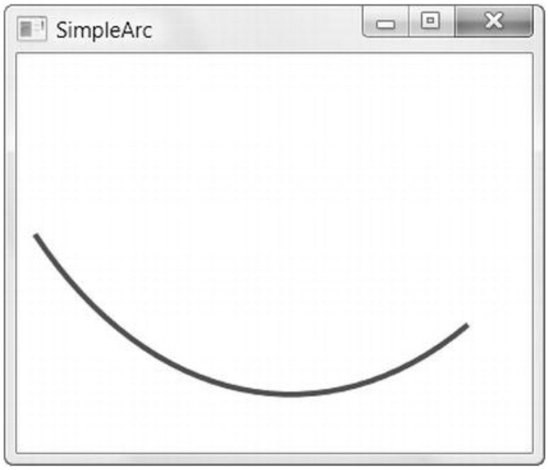

***Figure 13-4.*** *A simple arc*

So far, arcs sound fairly straightforward. However, it turns out that even with the start and end points and the size of the ellipse, you still don't have all the information you need to draw your arc unambiguously. In the previous example, you're relying on two default values that may not be set to your liking.

To understand the problem, you need to consider the other ways that an arc can connect the same two points. If you picture two points on an ellipse, it's clear that you can connect them in two ways: by going around the short side, or by going around the long side. Figure 13-5 illustrates these choices.

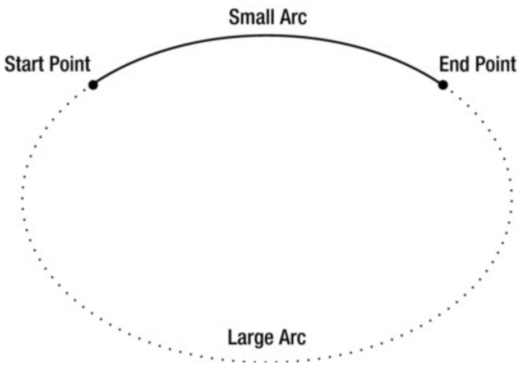

***Figure 13-5.*** *Two ways to trace a curve along an ellipse*

You set the direction by using the ArcSegment.IsLargeArc property, which can be true or false. The default value is false, which means you get the shorter of the two arcs.

Even after you've set the direction, there is still one point of ambiguity: where the ellipse is placed. For example, imagine you draw an arc that connects a point on the left with a point on the right, using the shortest possible arc. The curve that connects these two points could be stretched down and then up (as it does in Figure 13-4), or it could be flipped so that it curves up and then down. The arc you get depends on the order in which you define the two points in the arc and the ArcSegment.SweepDirection property, which can be Counterclockwise (the default) or Clockwise. Figure 13-6 shows the difference.

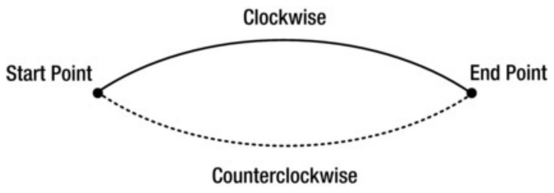

***Figure 13-6.*** *Two ways to flip a curve*

## Bézier Curves

Bézier curves connect two line segments by using a complex mathematical formula that incorporates two *control points* that determine how the curve is shaped. Bézier curves are an ingredient in virtually every vector drawing application ever created because they're remarkably flexible. Using nothing more than a start point, an end point, and two control points, you can create a surprisingly wide variety of smooth curves (including loops). Figure 13-7 shows a classic Bézier curve. Two small circles indicate the control points, and a dashed line connects each control point to the end of the line it affects the most.

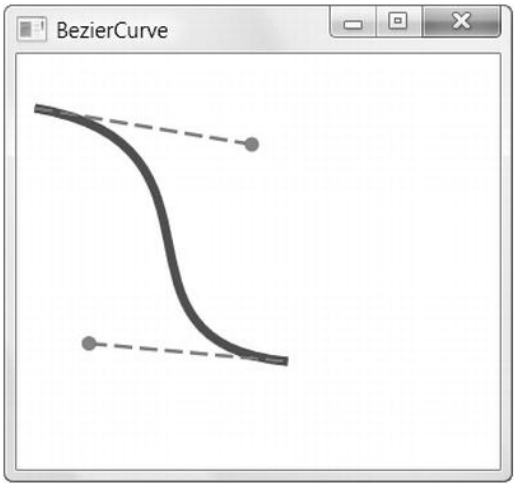

***Figure 13-7.*** *A Bézier curve*

Even without understanding the math underpinnings, it's fairly easy to get the "feel" of how Bézier curves work. Essentially, the two control points do all the magic. They influence the curve in two ways:

- At the starting point, a Bézier curve runs parallel with the line that connects it to the first control point. At the ending point, the curve runs parallel with the line that connects it to the end point. (In between, it curves.)
- The degree of curvature is determined by the distance to the two control points. If one control point is farther away, it exerts a stronger "pull."

To define a Bézier curve in markup, you supply three points. The first two points (BezierSegment.Point1 and BezierSegment.Point2) are the control points. The third point (BezierSegment.Point3) is the end point of the curve. As always, the starting point is that starting point of the path or wherever the previous segment leaves off.

The example shown in Figure 13-7 includes three separate components, each of which uses a different stroke and thus requires a separate Path element. The first path creates the curve, the second adds the dashed lines, and the third applies the circles that indicate the control points. Here's the complete markup:

```
<Canvas>
  <Path Stroke="Blue" StrokeThickness="5" Canvas.Top="20">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="10,10">
          <BezierSegment Point1="130,30" Point2="40,140"
           Point3="150,150"></BezierSegment>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
  <Path Stroke="Green" StrokeThickness="2" StrokeDashArray="5 2" Canvas.Top="20">
    <Path.Data>
      <GeometryGroup>
        <LineGeometry StartPoint="10,10" EndPoint="130,30"></LineGeometry>
        <LineGeometry StartPoint="40,140" EndPoint="150,150"></LineGeometry>
      </GeometryGroup>
    </Path.Data>
  </Path>
  <Path Fill="Red" Stroke="Red" StrokeThickness="8"  Canvas.Top="20">
    <Path.Data>
      <GeometryGroup>
        <EllipseGeometry Center="130,30"></EllipseGeometry>
        <EllipseGeometry Center="40,140"></EllipseGeometry>
      </GeometryGroup>
    </Path.Data>
  </Path>
</Canvas>
```

Trying to code Bézier paths is a recipe for many thankless hours of trial-and-error computer coding. You're much more likely to draw your curves (and many other graphical elements) in a dedicated drawing program that has an export-to-XAML feature or in Expression Design.

---

■ **Tip** To learn more about the algorithm that underlies the Bézier curve, you can read an informative Wikipedia article on the subject at `http://en.wikipedia.org/wiki/Bezier_curve`.

---

## Using the Geometry Mini-Language

The geometries you've seen so far have been relatively concise, with only a few points. More-complex geometries are conceptually the same but can easily require hundreds of segments. Defining each line, arc, and curve in a complex path is extremely verbose and unnecessary. After all, it's likely that complex paths will be generated by a design tool, rather than written by hand, so the clarity of the markup isn't all that important. With this in mind, the creators of WPF added a more concise alternate syntax for defining geometries that allows you to represent detailed figures with much smaller amounts of markup. This syntax is often described as the *geometry mini-language* (and sometimes the *path mini-language* because of its application with the Path element).

To understand the mini-language, you need to realize that it is essentially a long string holding a series of commands. These commands are read by a type converter, which then creates the corresponding geometry. Each command is a single letter and is optionally followed by a few bits of numeric information (such as X and Y coordinates) separated by spaces. Each command is also separated from the previous command with a space.

For example, a bit earlier, you created a basic triangle by using a closed path with two line segments. Here's the markup that did the trick:

```
<Path Stroke="Blue">
  <Path.Data>
    <PathGeometry>
      <PathFigure IsClosed="True" StartPoint="10,100">
        <LineSegment Point="100,100" />
        <LineSegment Point="100,50" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

Here's how you could duplicate this figure by using the mini-language:

```
<Path Stroke="Blue" Data="M 10,100 L 100,100 L 100,50 Z"/>
```

This path uses a sequence of four commands. The first command (M) creates the PathFigure and sets the starting point to (10, 100). The following two commands (L) create line segments. The final command (Z) ends the PathFigure and sets the IsClosed property to true. The commas in this string are optional, as are the spaces between the command and its parameters, but you must leave at least one space between adjacent parameters and commands. That means you can reduce the syntax even further to this less-readable form:

```
<Path Stroke="Blue" Data="M10 100 L100 100 L100 50 Z"/>
```

When creating a geometry with the mini-language, you are actually creating a StreamGeometry object, not a PathGeometry. As a result, you won't be able to modify the geometry later in your code. If this isn't acceptable, you can create a PathGeometry explicitly but use the same syntax to define its collection of PathFigure objects. Here's how:

```
<Path Stroke="Blue">
  <Path.Data>
    <PathGeometry Figures="M 10,100 L 100,100 L 100,50 Z" />
  </Path.Data>
</Path>
```

The geometry mini-language is easy to grasp. It uses a fairly small set of commands, which are detailed in Table 13-5. Parameters are shown in italics.

***Table 13-5.** Commands for the Geometry Mini-Language*

| Command | Description |
|---|---|
| F *value* | Sets the Geometry.FillRule property. Use 0 for EvenOdd or 1 for Nonzero. This command must appear at the beginning of the string (if you decide to use it). |
| M *x,y* | Creates a new PathFigure for the geometry and sets its start point. This command must be used before any other commands except F. However, you can also use it during your drawing sequence to move the origin of your coordinate system. (The M stands for *move*.) |
| L *x,y* | Creates a LineSegment to the specified point. |
| H *x* | Creates a horizontal LineSegment by using the specified X value and keeping the Y value constant. |
| V *y* | Creates a vertical LineSegment by using the specified Y value and keeping the X value constant. |
| A *radiusX, radiusY degrees isLargeArc, isClockwise x,y* | Creates an ArcSegment to the indicated point. You specify the radii of the ellipse that describes the arc, the number of degrees the arc is rotated, and Boolean flags that set the IsLargeArc and SweepDirection properties described earlier. |
| C x1,y1 x2,y2 x,y | Creates a BezierSegment to the indicated point, using control points at (x1, y1) and (x2, y2). |
| Q x1, y1 x,y | Creates a QuadraticBezierSegment to the indicated point, with one control point at (x1, y1). |
| S x2,y2 x,y | Creates a smooth BezierSegment by using the second control point from the previous BezierSegment as the first control point in the new BezierSegment. |
| Z | Ends the current PathFigure and sets IsClosed to true. You don't need to use this command if you don't want to set IsClosed to true. Instead, simply use M if you want to start a new PathFigure or end the string. |

■ **Tip** There's one more trick in the geometry mini-language. You can use a command in lowercase if you want its parameters to be evaluated relative to the previous point rather than using absolute coordinates.

## Clipping with Geometry

As you've seen, geometries are the most powerful way to create a shape. However, geometries aren't limited to the Path element. They're also used anywhere you need to supply the abstract definition of a shape (rather than draw a real, concrete shape in a window).

Another place geometries are used is to set the Clip property, which is provided by all elements. The Clip property allows you to constrain the outer bounds of an element to fit a specific geometry. You can use the Clip property to create exotic effects. Although it's commonly used to trim image content in an Image element, you can use the Clip property with any element. The only limitation is that you'll need a closed geometry if you actually want to see anything—individual curves and line segments aren't of much use.

The following example defines a single geometry that's used to clip two elements: an Image element that contains a bitmap, and a standard Button element. The results are shown in Figure 13-8.

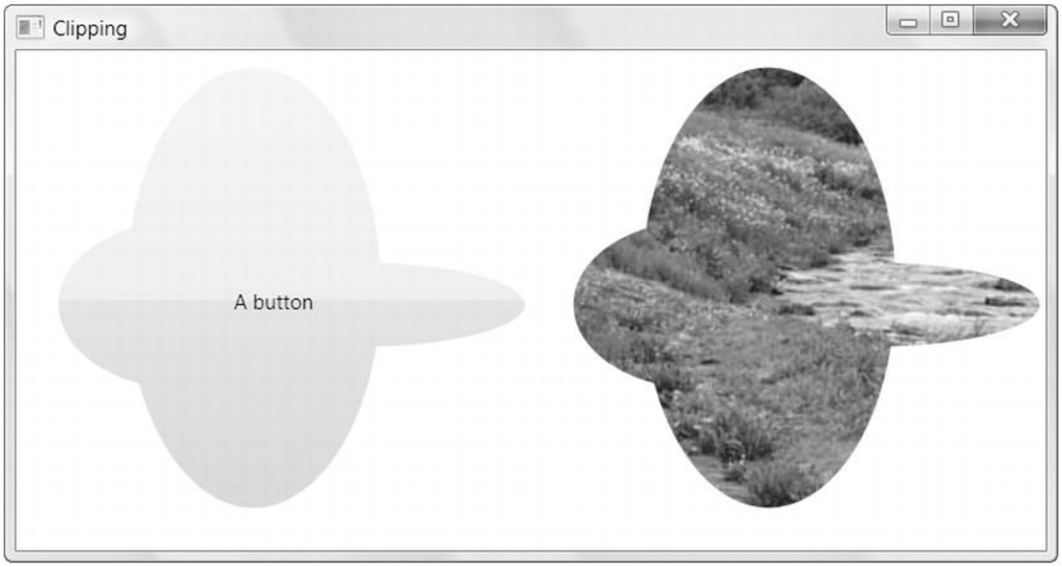

*Figure 13-8. Clipping two elements*

Here's the markup for this example:

```
<Window.Resources>
  <GeometryGroup x:Key="clipGeometry" FillRule="Nonzero">
    <EllipseGeometry RadiusX="75" RadiusY="50" Center="100,150"></EllipseGeometry>
    <EllipseGeometry RadiusX="100" RadiusY="25" Center="200,150"></EllipseGeometry>
    <EllipseGeometry RadiusX="75" RadiusY="130" Center="140,140"></EllipseGeometry>
  </GeometryGroup>
</Window.Resources>
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Button Clip="{StaticResource clipGeometry}">A button</Button>
  <Image Grid.Column="1" Clip="{StaticResource clipGeometry}"
   Stretch="None" Source="creek.jpg"></Image>
</Grid>
```

There's one limitation with clipping. The clipping you set doesn't take the size of the element into account. In other words, if the button in Figure 13-8 becomes larger or smaller when the window is resized, the clipped region will remain the same and show a different portion of the button. One possible solution is to wrap the element in a Viewbox to provide automatic rescaling. However, this causes *everything* to resize proportionately, including the details you do want to resize (the clip region and button surface) and those you might not (the button text and the line that draws the button border).

In the next section, you'll go a bit further with Geometry objects and use them to define a lightweight drawing that can be used in a variety of ways.

## Drawings

As you've learned, the abstract Geometry class represents a shape or a path. The abstract Drawing class plays a complementary role. It represents a 2-D drawing; in other words, it contains all the information you need to display a piece of vector or bitmap art.

Although there are several types of drawing classes, the GeometryDrawing is the one that works with the geometries you've learned about so far. It adds the stroke and fill details that determine how the geometry should be painted. You can think of a GeometryDrawing as a single shape in a piece of vector clip art. For example, it's possible to convert a standard Windows Metafile Format (.wmf) into a collection of GeometryDrawing objects that are ready to insert into your user interface. (In fact, you'll learn how to do exactly this in the "Exporting Clip Art" section a little later in this chapter.)

It helps to consider a simple example. Earlier, you saw how to define a simple PathGeometry that represents a triangle:

```
<PathGeometry>
  <PathFigure IsClosed="True" StartPoint="10,100">
    <LineSegment Point="100,100" />
    <LineSegment Point="100,50" />
  </PathFigure>
</PathGeometry>
```

You can use this PathGeometry to build a GeometryDrawing like so:

```
<GeometryDrawing Brush="Yellow">
  <GeometryDrawing.Pen>
    <Pen Brush="Blue" Thickness="3"></Pen>
  </GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <PathGeometry>
    <PathFigure IsClosed="True" StartPoint="10,100">
      <LineSegment Point="100,100" />
      <LineSegment Point="100,50" />
    </PathFigure>
  </PathGeometry>
  </GeometryDrawing.Geometry>
</GeometryDrawing>
```

Here, the PathGeometry defines the shape (a triangle). The GeometryDrawing defines the shape's appearance (a yellow triangle with a blue outline). Neither the PathGeometry nor the GeometryDrawing is an element, so you can't use either one directly to add your custom-drawn content to a window. Instead, you'll need to use another class that supports drawings, as described in the next section.

---

■ **Note** The GeometryDrawing class introduces a new detail: the System.Windows.Media.Pen class. The Pen class provides the Brush and Thickness properties used in the previous example, along with all the stroke-related properties you learned about with shapes (StartLine, EndLineCap, DashStyle, DashCap, LineJoin, and MiterLimit). In fact, most Shape-derived classes use Pen objects internally in their drawing code but expose pen-related properties directly for ease of use.

---

GeometryDrawing isn't the only drawing class in WPF (although it is the most relevant one when considering 2-D vector graphics). In fact, the Drawing class is meant to represent *all* types of 2-D graphics, and there's a small group of classes that derive from it. Table 13-6 lists them all.

***Table 13-6.*** *The Drawing Classes*

| Class | Description | Properties |
|---|---|---|
| GeometryDrawing | Wraps a geometry with the brush that fills it and the pen that outlines it. | Geometry, Brush, Pen |
| ImageDrawing | Wraps an image (typically, a file-based bitmap image) with a rectangle that defines its bounds. | ImageSource, Rect |
| VideoDrawing | Combines a MediaPlayer that's used to play a video file with a rectangle that defines its bounds. Chapter 26 has the details about WPF's multimedia support. | Player, Rect |
| GlyphRunDrawing | Wraps a low-level text object known as a GlyphRun with a brush that paints it. | GlyphRun, ForegroundBrush |
| DrawingGroup | Combines a collection of Drawing objects of any type. The DrawingGroup allows you to create composite drawings, and apply effects to the entire collection at once using one of its properties. | BitmapEffect, BitmapEffectInput, Children, ClipGeometry, GuidelineSet, Opacity, OpacityMask, Transform |

## Displaying a Drawing

Because Drawing-derived classes are not elements, they can't be placed in your user interface. Instead, to display a drawing, you need to use one of three classes listed in Table 13-7.

***Table 13-7.*** *Classes for Displaying a Drawing*

| Class | Derives From | Description |
|---|---|---|
| DrawingImage | ImageSource | Allows you to host a drawing inside an Image element. |
| DrawingBrush | Brush | Allows you to wrap a drawing with a brush, which you can then use to paint any surface. |
| DrawingVisual | Visual | Allows you to place a drawing in a lower-level visual object. Visuals don't have the overhead of true elements, but can still be displayed if you implement the required infrastructure. You'll learn more about using visuals in Chapter 14. |

There's a common theme in all of these classes. Quite simply, they give you a way to display your 2-D content with less overhead.

For example, imagine you want to use a piece of vector art to create the icon for a button. The most convenient (and resource-intensive) way to do this is to place a Canvas inside the button, and place a series of Shape-derived elements inside the Canvas:

```
<Button ... >
  <Canvas ... >
    <Polyline ... >
    <Polyline ... >
    <Rectangle ... >
    <Ellipse ... >
    <Polygon ... >
    ...
  </Canvas>
</Button>
```

As you already know, if you take this approach, each element is completely independent, with its own memory footprint, event handling, and so on. A better approach is to reduce the number of elements by using the Path element. Because each path has a single stroke and fill, you'll still need a large number of Path objects, but you'll probably be able to reduce the number of elements somewhat:

```
<Button ... >
  <Canvas ... >
    <Path ... >
    <Path ... >
    ...
  </Canvas>
</Button>
```

Once you start using the Path element, you've made the switch from separate shapes to distinct geometries. You can carry the abstraction one level further by extracting the geometry, stroke, and fill information from the path, and turning it into a drawing. You can then fuse your drawings together in a DrawingGroup and place that DrawingGroup in a DrawingImage, which can in turn be placed in an Image element. Here's the new markup this process creates:

```
<Button ... >
  <Image ... >
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing>
          <DrawingGroup>
            <GeometryDrawing ... >
            <GeometryDrawing ... >
            <GeometryDrawing ... >
            ...
          </DrawingGroup>
        </DrawingImage.Drawing>
      </DrawingImage>
    <Image.Source>
  </Image>
</Button>
```

This is a significant change. It hasn't simplified your markup, as you've simply substituted one GeometryDrawing object for each Path object. However, it *has* reduced the number of elements and hence the overhead that's required. The previous example created a Canvas inside the button and added a separate element for each path. This example requires just one nested element: the Image inside the button. The trade-off is that you no longer have the ability to handle events for each distinct path (for example, you can't detect mouse clicks on separate regions of the drawing). But in a static image that's used for a button, it's unlikely that you want this ability anyway.

---

■ **Note** It's easy to confuse DrawingImage and ImageDrawing, two WPF classes with awkwardly similar names. DrawingImage is used to place a drawing inside an Image element. Typically, you'll use it to put vector content in an Image. ImageDrawing is completely different—it's a Drawing-derived class that accepts bitmap content. This allows you to combine GeometryDrawing and ImageDrawing objects in one DrawingGroup, thereby creating a drawing with vector and bitmap content that you can use however you want.

---

Although the DrawingImage gives you the majority of the savings, you can still get a bit more efficient and remove one more element with the help of the DrawingBrush.

The basic idea is to wrap your DrawingImage in a DrawingBrush, like so:

```
<Button ... >
  <Button.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <DrawingGroup>
          <GeometryDrawing ... >
          <GeometryDrawing ... >
          <GeometryDrawing ... >
          ...
        </DrawingGroup>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Button.Background>
</Button>
```

The DrawingBrush approach isn't exactly the same as the DrawingImage approach shown earlier, because the default way that an Image sizes its content is different. The default Image.Stretch property is Uniform, which scales the image up or down to fit the available space. The default DrawingBrush.Stretch property is Fill, which may distort your image.

When changing the Stretch property of a DrawingBrush, you may also want to adjust the Viewport setting to explicitly tweak the location and size of the drawing in the fill region. For example, this markup scales the drawing used by the drawing brush to take 90 percent of the fill area:

```
<DrawingBrush Stretch="Fill" Viewport="0,0 0.9,0.9">
```

This is useful with the button example because it gives some space for the border around the button. Because the DrawingBrush isn't an element, it won't be placed using the WPF layout process. That means that unlike the Image, the placement of the content in the DrawingBrush won't take the Button.Padding value into account.

---

■ **Tip** Using DrawingBrush objects also allows you to create some effects that wouldn't otherwise be possible, such as tiling. Because DrawingBrush derives from TileBrush, you can use the TileMode property to repeat a drawing in a pattern across your fill region. Chapter 12 has the full details about tiling with the TileBrush.

---

One quirk with the DrawingBrush approach is that the content disappears when you move the mouse over the button and a new brush is used to paint its surface. But when you use the Image approach, the picture remains unaffected. To deal with this issue, you need to create a custom control template for the button that doesn't paint its background in the same way. This technique is demonstrated in Chapter 17.

Either way, whether you use a DrawingImage on its own or wrapped in a DrawingBrush, you should also consider refactoring your markup by using *resources*. The basic idea is to define each DrawingImage or DrawingBrush as a distinct resource, which you can then refer to when needed. This is a particularly good idea if you want to show the same content in more than one element or in more than one window, because you simply need to reuse the resource, rather than copy a whole block of markup.

## Exporting Clip Art

Although all of these examples have declared their drawings inline, a more common approach is to place some portion of this content in a resource dictionary so it can be reused throughout your application (and modified in one place). It's up to you how you break down this markup into resources, but two common choices are to store a dictionary full of DrawingImage objects or a dictionary stocked with DrawingBrush objects. Optionally, you can factor out the Geometry objects and store them as separate resources. (This is handy if you use the same geometry in more than one drawing, with different colors.)

Of course, very few developers will code much (if any) art by hand. Instead, they'll use dedicated design tools that export the XAML content they need. Most design tools don't support XAML export yet, although there are a wide variety of plug-ins and converters that fill the gaps. Here are some examples:

- `www.mikeswanson.com/XAMLExport` has a free XAML plug-in for Adobe Illustrator.
- `www.mikeswanson.com/swf2xaml` has a free XAML converter for Adobe Flash files.
- Expression Design, Microsoft's illustration and graphic design program, has a built-in XAML export feature. In can read a variety of vector art file formats, including the Windows Metafile Format (.wmf), which allows you to import existing clip art and export it as XAML.

However, even if you use one of these tools, the knowledge you've learned about geometries and drawings is still important for several reasons.

First, many programs allow you to choose whether you want to export a drawing as a combination of separate elements in a Canvas or as a collection of DrawingBrush or DrawingImage resources. Usually, the Canvas choice is the default, because it preserves more features. However, if you're using a large number of drawings, your drawings are complex, or you simply want to use the least amount of memory for static graphics such as button icons, it's a much better idea to use DrawingBrush or DrawingImage resources. Better still, these formats are separated from the rest of your user interface, so it's easier to update them later. (In fact, you could even place your DrawingBrush or DrawingImage resources in a separately compiled DLL assembly, as described in Chapter 10.)

---

■ **Tip** To save resources in Expression Design, you must explicitly choose Resource Dictionary instead of Canvas in the Document Format list box.

---

Another reason that it's important to understand the plumbing behind 2-D graphics is that it makes it far easier for you to manipulate them. For example, you alter a standard 2-D graphic by modifying the brushes used to paint various shapes, applying transforms to individual geometries, or altering the opacity or transform of an entire layer of shapes (through a DrawingGroup object). More dramatically, you can add, remove, or alter individual geometries. These techniques can be easily combined with the animation skills you'll pick up in Chapter 15 and Chapter 16. For example, it's easy to rotate a Geometry object by modifying the Angle property of a RotateTransform, fade a layer of shapes into existence by using DrawingGroup.Opacity, or create a swirling gradient effect by animating a LinearGradientBrush that paints the fill for a GeometryDrawing.

---

■ **Tip** If you're really curious, you can hunt down the resources used by other WPF applications. The basic technique is to use a tool such as Reflector (`www.reflector.net`) and open the assembly with the resources. You can extract one of the BAML resources and decompile it back to XAML. Of course, most companies won't take kindly to developers who steal their handcrafted graphics to use in their own applications!

---

## The Last Word

In this chapter, you delved deeper into WPF's 2-D drawing model. You began with a thorough look at the Path class, the most powerful of WPF's shape classes, and the geometry model that it uses. Then you considered how you could use a geometry to build a drawing, and to use that drawing to display lightweight, noninteractive graphics. In the next chapter, you'll consider an even leaner approach—forgoing elements and using the lower-level Visual class to perform your rendering by hand.