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 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. One product that uses this approach is Expression Blend.

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% 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 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:

- `http://www.mikeswanson.com/XAMLExport` has a free XAML plug-in for Adobe Illustrator.
- `http://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 like 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 why 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 can 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 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 .NET Reflector (`http://www.red-gate.com/products/reflector`) to find the assembly with the resources. You can then use a .NET Reflector plug-in (`http://reflectoraddins.codeplex.com`) to 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.

CHAPTER 14

Effects and Visuals

In the previous two chapters, you explored the core concepts of 2-D drawing in WPF. Now that you have a solid understanding of the fundamentals—such as shapes, brushes, transforms, and drawings—it's worth digging down to WPF's lower-level graphics features.

Usually, you'll turn to these features when raw performance becomes an issue, or when you need access to individual pixels (or both). In this chapter, you'll consider three WPF techniques that can help you out:

- **Visuals.** If you want to build a program for drawing vector art, or you plan to create a canvas with thousands of shapes that can be manipulated individually, WPF's element system and shape classes will only slow you down. Instead, you need a leaner approach, which is to use the lower-level Visual class to perform your rendering by hand.
- **Pixel shaders.** If you want to apply complex visual effects (like blurs and color tuning) to an element, the easiest approach is to alter individual pixels with a pixel shader. Best of all, pixel shaders are hardware-accelerated for blistering performance, and there are plenty of ready-made effects that you can drop into your applications with minimal effort.
- **The WriteableBitmap.** It's far more work, but the WriteableBitmap class lets you own a bitmap in its entirety—meaning you can set and inspect any of its pixels. You can use this feature in complex data visualization scenarios (for example, when graphing scientific data), or just to generate an eye-popping effect from scratch.

What's New Although previous versions of WPF had support for bitmap effects, WPF 3.5 SP1 added a new effect model. Now the original bitmap effects are considered obsolete, because they don't (and never will) support hardware acceleration. WPF 3.5 SP1 also introduced the WriteableBitmap class, which you'll explore in this chapter.

Visuals

In the previous chapter, you learned the best ways to deal with modest amounts of graphical content. By using geometries, drawings, and paths, you reduce the overhead of your 2-D art. Even if you're

using complex compound shapes with layered effects and gradient brushes, this is an approach that performs well.

However, this design isn't suitable for drawing-intensive applications that need to render a huge number of graphical elements. For example, consider a mapping program, a physics modeling program that demonstrates particle collisions, or a side-scrolling game. The problem posed by these applications isn't the complexity of the art, but the sheer number of individual graphical elements. Even if you replace your Path elements with lighter weight Geometry objects, the overhead will still hamper the application's performance.

The WPF solution for this sort of situation is to use the lower-level *visual layer* model. The basic idea is that you define each graphical element as a Visual object, which is an extremely lightweight ingredient that has less overhead than a Geometry object or a Path object. You can then use a single element to render all your visuals in a window.

In the following sections, you'll learn how to create visuals, manipulate them, and perform hit testing. Along the way, you'll build a basic vector-based drawing application that lets you add squares to a drawing surface, select them, and drag them around.

Drawing Visuals

Visual is an abstract class, so you can't create an instance of it. Instead, you need to use one of the classes that derive from Visual. These include UIElement (which is the root of WPF's element model), Viewport3DVisual (which allows you to display 3-D content, as described in Chapter 27), and ContainerVisual (which is a basic container that holds other visuals). But the most useful derived class is DrawingVisual, which derives from ContainerVisual and adds the support you need to "draw" the graphical content you want to place in your visual.

To draw content in a DrawingVisual, you call the DrawingVisual.RenderOpen() method. This method returns a DrawingContext that you can use to define the content of your visual. When you're finished, you call DrawingContext.Close(). Here's how it all unfolds:

```
DrawingVisual visual = new DrawingVisual();
DrawingContext dc = visual.RenderOpen();
// (Perform drawing here.)
dc.Close();
```

Essentially, the DrawingContext class is made up of methods that add some graphical detail to your visual. You call these methods to draw various shapes, apply transforms, change the opacity, and so on. Table 14-1 lists the methods of the DrawingContext class.

Table 14-1. *DrawingContext Methods*

Name	Description
DrawLine(), DrawRectangle(), DrawRoundedRectangle(), and DrawEllipse()	Draw the specified shape at the point you specify, with the fill and outline you specify. These methods mirror the shapes you saw in Chapter 12.
DrawGeometry () and DrawDrawing()	Draw more complex Geometry objects and Drawing objects.

Name	Description
DrawText()	Draws text at the specified location. You specify the text, font, fill, and other details by passing a FormattedText object to this method. You can use DrawText() to draw wrapped text if you set the FormattedText.MaxTextWidth property.
DrawImage()	Draws a bitmap image in a specific region (as defined by a Rect).
DrawVideo()	Draws video content (wrapped in a MediaPlayer object) in a specific region. Chapter 26 has the full details about video rendering in WPF.
Pop()	Reverses the last Push*Xxx*() method that was called. You use the Push*Xxx*() method to temporarily apply one or more effects, and the Pop() method to reverse them.
PushClip()	Limits drawing to a specific clip region. Content that falls outside this region isn't drawn.
PushEffect ()	Applies a BitmapEffect to subsequent drawing operations.
PushOpacity() and PushOpacityMask()	Apply a new opacity setting or opacity mask (see Chapter 12) to make subsequent drawing operations partially transparent.
PushTransform()	Sets a Transform object that will be applied to subsequent drawing operations. You can use a transformation to scale, displace, rotate, or skew content.

Here's an example that creates a visual that contains a basic black triangle with no fill:

```
DrawingVisual visual = new DrawingVisual();
using (DrawingContext dc = visual.RenderOpen())
{
    Pen drawingPen = new Pen(Brushes.Black, 3);
    dc.DrawLine(drawingPen, new Point(0, 50), new Point(50, 0));
    dc.DrawLine(drawingPen, new Point(50, 0), new Point(100, 50));
    dc.DrawLine(drawingPen, new Point(0, 50), new Point(100, 50));
}
```

As you call the DrawingContext methods, you aren't actually painting your visual; rather, you're defining its visual appearance. When you finish by calling Close(), the completed drawing is stored in the visual and exposed through the read-only DrawingVisual.Drawing property. WPF retains the Drawing object so that it can repaint the window when needed.

The order of your drawing code is important. Later drawing actions can write content overtop of what already exists. The Push*Xxx*() methods apply settings that will apply to future drawing operations. For example, you can use PushOpacity() to change the opacity level, which will then affect all subsequent drawing operations. You can use Pop() to reverse the most recent Push*Xxx*() method. If you call more than one Push*Xxx*() method, you can switch them off one at a time with subsequent Pop() calls.

Once you've closed the DrawingContext, you can't modify your visual any further. However, you can apply a transform or change a visual's overall opacity (using the Transform and Opacity properties of the DrawingVisual class). If you want to supply completely new content, you can call RenderOpen() again and repeat the drawing process.

■ **Tip** Many drawing methods use Pen and Brush objects. If you plan to draw many visuals with the same stroke and fill, or if you expect to render the same visual multiple times (in order to change its content), it's worth creating the Pen and Brush objects you need upfront and holding on to them over the lifetime of your window.

Visuals are used in several different ways. In the remainder of this chapter, you'll learn how to place a DrawingVisual in a window and perform hit testing for it. You can also use a DrawingVisual to define content you want to print, as you'll see in Chapter 29. Finally, you can use visuals to render a custom-drawn element by overriding the OnRender() method, as you'll see in Chapter 18. In fact, that's exactly how the shape classes that you learned about in Chapter 12 do their work. For example, here's the rendering code that the Rectangle element uses to paint itself:

```
protected override void OnRender(DrawingContext drawingContext)
{
    Pen pen = base.GetPen();
    drawingContext.DrawRoundedRectangle(base.Fill, pen, this._rect,
      this.RadiusX, this.RadiusY);
}
```

Wrapping Visuals in an Element

Defining a visual is the most important step in visual-layer programming, but it's not enough to actually show your visual content onscreen. To display a visual, you need the help of a full-fledged WPF element that can add it to the visual tree. At first glance, this seems to reduce the benefit of visual-layer programming—after all, isn't the whole point to avoid elements and their high overhead? However, a single element has the ability to display an unlimited number of elements. Thus, you can easily create a window that holds only one or two elements but hosts thousands of visuals.

To host a visual in an element, you need to perform the following tasks:

- Call the AddVisualChild() and AddLogicalChild() methods of your element to register your visual. Technically speaking, these tasks aren't required to make the visual appear, but they are required to ensure it is tracked correctly, appears in the visual and logical tree, and works with other WPF features such as hit testing.
- Override the VisualChildrenCount property and return the number of visuals you've added.
- Override the GetVisualChild() method and add the code needed to return your visual when it's requested by index number.

When you override VisualChildrenCount and GetVisualChild(), you are essentially hijacking that element. If you're using a content control, decorator, or panel that can hold nested elements, these

elements will no longer be rendered. For example, if you override these two methods in a custom window, you won't see the rest of the window content. Instead, you'll see only the visuals that you've added.

For this reason, it's common to create a dedicated custom class that wraps the visuals you want to display. For example, consider the window shown in Figure 14-1. It allows the user to add squares (each of which is a visual) to a custom Canvas.

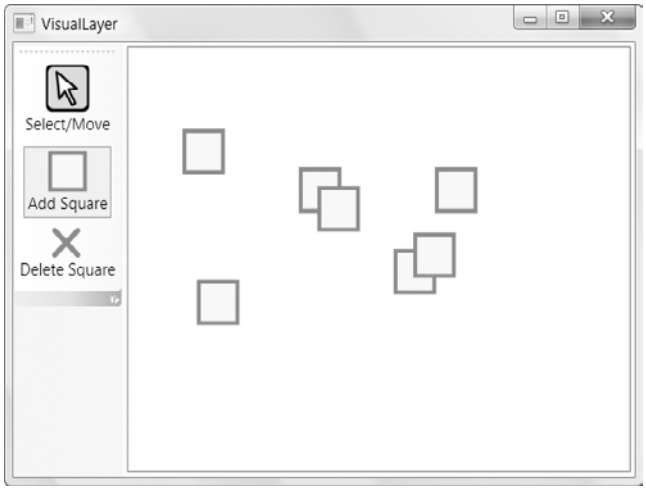

Figure 14-1. *Drawing visuals*

On the left side of the window in Figure 14-1 is a toolbar with three RadioButton objects. As you'll discover in Chapter 25, the ToolBar changes the way some basic controls are rendered, such as buttons. By using a group of RadioButton objects, you can create a set of linked buttons. When you click one of the buttons in this set, it is selected and remains "pushed," while the previously selected button reverts to its normal appearance.

On the right side of the window in Figure 14-1 is a custom Canvas named DrawingCanvas, which stores a collection of visuals internally. DrawingCanvas returns the total number of squares in the VisualChildrenCount property, and uses the GetVisualChild() method to provide access to each visual in the collection. Here's how these details are implemented:

```
public class DrawingCanvas : Canvas
{
    private List<Visual> visuals = new List<Visual>();

    protected override int VisualChildrenCount
    {
        get { return visuals.Count; }
    }

    protected override Visual GetVisualChild(int index)
    {
        return visuals[index];
    }
    ...
```

Additionally, the DrawingCanvas includes an AddVisual() method and a DeleteVisual() method to make it easy for the consuming code to insert visuals into the collection, with the appropriate tracking:

```
    ...
    public void AddVisual(Visual visual)
    {
        visuals.Add(visual);

        base.AddVisualChild(visual);
        base.AddLogicalChild(visual);
    }

    public void DeleteVisual(Visual visual)
    {
        visuals.Remove(visual);

        base.RemoveVisualChild(visual);
        base.RemoveLogicalChild(visual);
    }
}
```

The DrawingCanvas doesn't include the logic for drawing squares, selecting them, and moving them. That's because this functionality is controlled at the application layer. This makes sense because there might be several different drawing tools, all of which work with the same DrawingCanvas. Depending on which button the user clicks, the user might be able to draw different types of shapes or use different stroke and fill colors. All of these details are specific to the window. The DrawingCanvas simply provides the functionality for hosting, rendering, and tracking your visuals.

Here's how the DrawingCanvas is declared in the XAML markup for the window:

```
<local:DrawingCanvas x:Name="drawingSurface" Background="White" ClipToBounds="True"
  MouseLeftButtonDown="drawingSurface_MouseLeftButtonDown"
  MouseLeftButtonUp="drawingSurface_MouseLeftButtonUp"
  MouseMove="drawingSurface_MouseMove" />
```

■ **Tip** By setting the background to white (rather than transparent), it's possible to intercept all mouse clicks on the canvas surface.

Now that you've considered the DrawingCanvas container, it's worth considering the event handling code that creates the squares. The starting point is the event handler for the MouseLeftButton. It's at this point that the code determines what operation is being performed—square creation, square deletion, or square selection. At the moment, we're just interested in the first task:

```
private void drawingSurface_MouseLeftButtonDown(object sender,
  MouseButtonEventArgs e)
{
    Point pointClicked = e.GetPosition(drawingSurface);

    if (cmdAdd.IsChecked == true)
```

```
    {
        // Create, draw, and add the new square.
        DrawingVisual visual = new DrawingVisual();
        DrawSquare(visual, pointClicked, false);
        drawingSurface.AddVisual(visual);
    }
    ...
}
```

The actual work is performed by a custom method named DrawSquare(). This approach is useful because the square drawing needs to be triggered at several different points in the code. Obviously, DrawSquare() is required when the square is first created. It's also used when the appearance of the square changes for any reason (such as when it's selected).

The DrawSquare() method accepts three parameters: the DrawingVisual to draw, the point for the top-left corner of the square, and a Boolean flag that indicates whether the square is currently selected, in which case it is given a different fill color.

Here's the modest rendering code:

```
// Drawing constants.
private Brush drawingBrush = Brushes.AliceBlue;
private Brush selectedDrawingBrush = Brushes.LightGoldenrodYellow;
private Pen drawingPen = new Pen(Brushes.SteelBlue, 3);
private Size squareSize = new Size(30, 30);

private void DrawSquare(DrawingVisual visual, Point topLeftCorner, bool isSelected)
{
    using (DrawingContext dc = visual.RenderOpen())
    {
        Brush brush = drawingBrush;
        if (isSelected) brush = selectedDrawingBrush;

        dc.DrawRectangle(brush, drawingPen,
          new Rect(topLeftCorner, squareSize));
    }
}
```

This is all you need to display a visual in a window: some code that renders the visual, and a container that handles the necessary tracking details. However, there's a bit more work to do if you want to add interactivity to your visuals, as you'll see in the following section.

Hit Testing

The square-drawing application not only allows users to draw squares, but it also allows them to move and delete existing squares. In order to perform either of these tasks, your code needs to be able to intercept a mouse click and find the visual at the clicked location. This task is called *hit testing*.

To support hit testing, it makes sense to add a GetVisual() method to the DrawingCanvas class. This method takes a point and returns the matching DrawingVisual. To do its work, it uses the static VisualTreeHelper.HitTest() method. Here's the complete code for the GetVisual() method:

```
public DrawingVisual GetVisual(Point point)
{
    HitTestResult hitResult = VisualTreeHelper.HitTest(this, point);
    return hitResult.VisualHit as DrawingVisual;
}
```

In this case, the code ignores any hit object that isn't a DrawingVisual, including the DrawingCanvas itself. If no squares are clicked, the GetVisual() method returns a null reference.

The delete feature makes use of the GetVisual() method. When the delete command is selected and a square is clicked, the MouseLeftButtonDown event handler uses this code to remove it:

```
else if (cmdDelete.IsChecked == true)
{
    DrawingVisual visual = drawingSurface.GetVisual(pointClicked);
    if (visual != null) drawingSurface.DeleteVisual(visual);
}
```

Similar code supports the dragging feature, but it needs a way to keep track of the fact that dragging is underway. Three fields in the window class serve this purpose—isDragging, clickOffset, and selectedVisual:

```
private bool isDragging = false;
private DrawingVisual selectedVisual;
private Vector clickOffset;
```

When the user clicks a shape, the isDragging field is set to true, the selectedVisual is set to the visual that was clicked, and the clickOffset records the space between the top-left corner of the square and the point where the user clicked. Here's the code from the MouseLeftButtonDown event handler:

```
else if (cmdSelectMove.IsChecked == true)
{
    DrawingVisual visual = drawingSurface.GetVisual(pointClicked);
    if (visual != null)
    {
        // Find the top-left corner of the square.
        // This is done by looking at the current bounds and
        // removing half the border (pen thickness).
        // An alternate solution would be to store the top-left
        // point of every visual in a collection in the
        // DrawingCanvas, and provide this point when hit testing.
        Point topLeftCorner = new Point(
          visual.ContentBounds.TopLeft.X + drawingPen.Thickness / 2,
          visual.ContentBounds.TopLeft.Y + drawingPen.Thickness / 2);
        DrawSquare(visual, topLeftCorner, true);

        clickOffset = topLeftCorner - pointClicked;
        isDragging = true;

        if (selectedVisual != null && selectedVisual != visual)
```

```
        {
            // The selection has changed. Clear the previous selection.
            ClearSelection();
        }
        selectedVisual = visual;
    }
}
```

Along with basic bookkeeping, this code calls DrawSquare() to rerender the DrawingVisual, giving it the new color. The code also uses another custom method, named ClearSelection(), to repaint the previously selected square so it returns to its normal appearance:

```
private void ClearSelection()
{
    Point topLeftCorner = new Point(
      selectedVisual.ContentBounds.TopLeft.X + drawingPen.Thickness / 2,
    selectedVisual.ContentBounds.TopLeft.Y + drawingPen.Thickness / 2);
    DrawSquare(selectedVisual, topLeftCorner, false);
    selectedVisual = null;
}
```

■ **Note** Remember that the DrawSquare() method defines the content for the square—it doesn't actually paint it in the window. For that reason, you don't need to worry about inadvertently painting overtop of another square that should be underneath. WPF manages the painting process, ensuring that visuals are painted in the order they are returned by the GetVisualChild() method (which is the order in which they are defined in the visuals collection).

Next, you need to actually move the square as the user drags, and end the dragging operation when the user releases the left mouse button. Both of these tasks are accomplished with some straightforward event handling code:

```
private void drawingSurface_MouseMove(object sender, MouseEventArgs e)
{
    if (isDragging)
    {
        Point pointDragged = e.GetPosition(drawingSurface) + clickOffset;
        DrawSquare(selectedVisual, pointDragged, true);
    }
}

private void drawingSurface_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    isDragging = false;
}
```

Complex Hit Testing

In the previous example, the hit-testing code always returns the topmost visual (or a null reference if the space is empty). However, the VisualTreeHelper class includes two overloads to the HitTest() method that allow you to perform more sophisticated hit testing. Using these methods, you can retrieve all the visuals that are at a specified point, even if they're obscured underneath other visuals. You can also find all the visuals that fall in a given geometry.

To use this more advanced hit-testing behavior, you need to create a callback. The VisualTreeHelper will then walk through your visuals from top to bottom (in the reverse order that you created them). Each time it finds a match, it calls your callback with the details. You can then choose to stop the search (if you've dug down enough levels) or continue until no more visuals remain.

The following code implements this technique by adding a GetVisuals() method to the DrawingCanvas. GetVisuals() accepts a Geometry object, which it uses for hit testing. It creates the callback delegate, clears the collection of hit test results, and then starts the hit-testing process by calling the VisualTreeHelper.HitTest() method. When the process is finished, it returns a collection with all the visuals that were found:

```
private List<DrawingVisual> hits = new List<DrawingVisual>();

public List<DrawingVisual> GetVisuals(Geometry region)
{
    // Remove matches from the previous search.
    hits.Clear();

    // Prepare the parameters for the hit test operation
    // (the geometry and callback).
    GeometryHitTestParameters parameters = new GeometryHitTestParameters(region);
    HitTestResultCallback callback =
      new HitTestResultCallback(this.HitTestCallback);

    // Search for hits.
    VisualTreeHelper.HitTest(this, null, callback, parameters);
    return hits;
}
```

■ **Tip** In this example, the callback is implemented by a separately defined method named HitTestResultCallback(). Both HitTestResultCallback() and GetVisuals() use the hits collection, so it must be defined as a member field. However, you could remove this requirement by using an anonymous method for the callback, which you would declare inside the GetVisuals() method.

The callback method implements your hit-testing behavior. Ordinarily, the HitTestResult object provides just a single property (VisualHit), but you can cast it to one of two derived types depending on the type of hit test you're performing.

If you're hit testing a point, you can cast HitTestResult to PointHitTestResult, which provides a relatively uninteresting PointHit property that returns the original point you used to perform the hit test. But if you're hit testing a Geometry object, as in this example, you can cast HitTestResult to

GeometryHitTestResult and get access to the IntersectionDetail property. This property tells you whether your geometry completely wraps the visual (FullyInside), the geometry and visual simply overlap (Intersects), or your hit-tested geometry falls within the visual (FullyContains). In this example, hits are counted only if the visual is completely inside the hit-tested region. Finally, at the end of the callback, you can return one of two values from the HitTestResultBehavior enumeration: Continue to keep looking for hits, or Stop to end the process.

```
private HitTestResultBehavior HitTestCallback(HitTestResult result)
{
    GeometryHitTestResult geometryResult = (GeometryHitTestResult)result;
    DrawingVisual visual = result.VisualHit as DrawingVisual;

    // Only include matches that are DrawingVisual objects and
    // that are completely inside the geometry.
    if (visual != null &&
        geometryResult.IntersectionDetail == IntersectionDetail.FullyInside)
    {
        hits.Add(visual);
    }
    return HitTestResultBehavior.Continue;
}
```

Using the GetVisuals() method, you can create the sophisticated selection box effect shown in Figure 14-2. Here, the user draws a box around a group of squares. The application then reports the number of squares in the region.

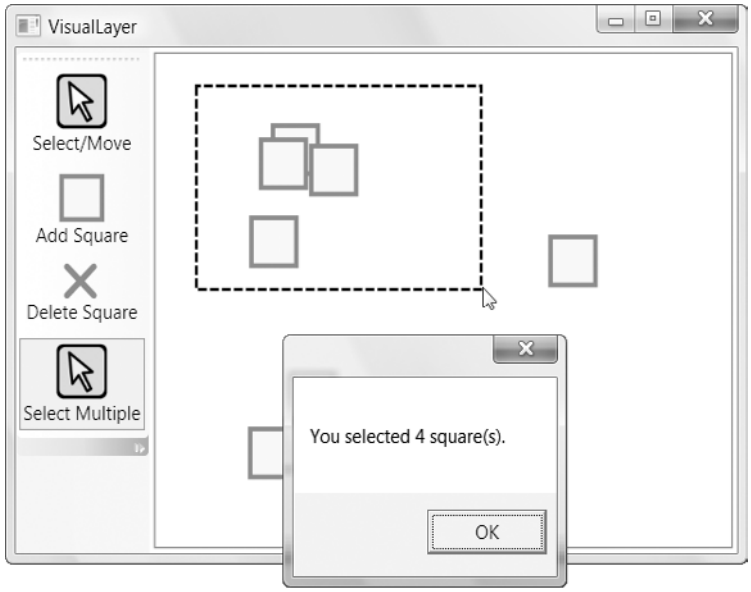

Figure 14-2. *Advanced hit testing*

To create the selection square, the window simply adds another DrawingVisual to the DrawingCanvas. The window also stores a reference to the selection square as a member field, along with a flag named isMultiSelecting that keeps track of when the selection box is being drawn, and a field named selectionSquareTopLeft that tracks the top-left corner of the current selection box:

```
private DrawingVisual selectionSquare;
private bool isMultiSelecting = false;
private Point selectionSquareTopLeft;
```

In order to implement the selection box feature, you need to add some code to the event handlers you've already seen. When the mouse is clicked, you need to create the selection box, switch isMultiSelecting to true, and capture the mouse. Here's the code that does this work in the MouseLeftButtonDown event handler:

```
else if (cmdSelectMultiple.IsChecked == true)
{
    selectionSquare = new DrawingVisual();
    drawingSurface.AddVisual(selectionSquare);

    selectionSquareTopLeft = pointClicked;
    isMultiSelecting = true;

    // Make sure we get the MouseLeftButtonUp event even if the user
    // moves off the Canvas. Otherwise, two selection squares could
    // be drawn at once.
    drawingSurface.CaptureMouse();
}
```

Now, when the mouse moves, you can check if the selection box is currently active, and draw it if it is. To do so, you need this code in the MouseMove event handler:

```
else if (isMultiSelecting)
{
    Point pointDragged = e.GetPosition(drawingSurface);
    DrawSelectionSquare(selectionSquareTopLeft, pointDragged);
}
```

The actual drawing takes place in a dedicated method named DrawSelectionSquare(), which looks a fair bit like the DrawSquare() method you considered earlier:

```
private Brush selectionSquareBrush = Brushes.Transparent;
private Pen selectionSquarePen = new Pen(Brushes.Black, 2);

private void DrawSelectionSquare(Point point1, Point point2)
{
    selectionSquarePen.DashStyle = DashStyles.Dash;

    using (DrawingContext dc = selectionSquare.RenderOpen())
    {
        dc.DrawRectangle(selectionSquareBrush, selectionSquarePen,
          new Rect(point1, point2));
    }
}
```

Finally, when the mouse is released you can perform the hit testing, show the message box, and then remove the selections square. To do so, you need this code in the MouseLeftButtonUp event handler:

```
if (isMultiSelecting)
{
    // Display all the squares in this region.
    RectangleGeometry geometry = new RectangleGeometry(
      new Rect(selectionSquareTopLeft, e.GetPosition(drawingSurface)));
    List<DrawingVisual> visualsInRegion =
      drawingSurface.GetVisuals(geometry);

    MessageBox.Show(String.Format("You selected {0} square(s).",
      visualsInRegion.Count));

    isMultiSelecting = false;
    drawingSurface.DeleteVisual(selectionSquare);
    drawingSurface.ReleaseMouseCapture();
}
```

Effects

WPF provides visual effects that you can apply to any element. The goal of effects is to give you an easy, declarative way to enhance the appearance of text, images, buttons, and other controls. Rather than write your own drawing code, you simply use one of the classes that derives from Effect (in the System.Windows.Media.Effects namespace) to get instant effects such as blurs, glows, and drop shadows.

Table 14-2 lists the effect classes that you can use.

Table 14-2. *Effects*

Name	Description	Properties
BlurEffect	Blurs the content in your element.	Radius, KernelType, RenderingBias
DropShadowEffect	Adds a rectangular drop shadow behind your element.	BlurRadius, Color, Direction, Opacity, ShadowDepth, RenderingBias
ShaderEffect	Applies a pixel shader, which is a ready-made effect that's defined in High Level Shading Language (HLSL) and already compiled.	PixelShader

The Effect-derived classes listed in Table 14-2 shouldn't be confused with bitmap effects, which derive from the BitmapEffect class in the same namespace. Although bitmap effects have a similar programming model, they have several significant limitations:

- Bitmap effects don't support pixel shaders, which are the most powerful and flexible way to create reusable effects.
- Bitmap effects are implemented in unmanaged code, and so require a fully trusted application. Therefore, you can't use bitmap effects in a browser-based XBAP application (Chapter 24).
- Bitmap effects are always rendered in software and don't use the resources of the video card. This makes them slow, especially when dealing with large numbers of elements or elements that have a large visual surface.

The BitmapEffect class dates back to the first version of WPF, which didn't include the Effect class. Bitmap effects remain only for backward compatibility.

The following sections dig deeper into the effect model and demonstrate the three Effect-derived classes: BlurEffect, DropShadowEffect, and ShaderEffect.

BlurEffect

WPF's simplest effect is the BlurEffect class. It blurs the content of an element, as though you're looking at it through an out-of-focus lens. You increase the level of blur by increasing the value of the Radius property (the default value is 5).

To use any effect, you create the appropriate effect object and set the Effect property of the corresponding element:

```
<Button Content="Blurred (Radius=2)" Padding="5" Margin="3">
  <Button.Effect>
    <BlurEffect Radius="2"></BlurEffect>
  </Button.Effect>
</Button>
```

Figure 14-3 shows three different blurs (where Radius is 2, 5, and 20) applied to a stack of buttons.

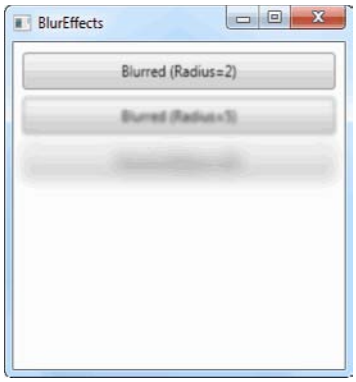

Figure 14-3. *Blurred buttons*