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.

Effects: If you want to apply complex visual effects (such as blurs and color tuning) to an element, the easiest approach is to alter individual pixels with a specialized tool called a *pixel shader*. 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.

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.
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 containing 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 on top 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.

After 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 up front and holding on to them over the lifetime of your window.

Visuals are used in several 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 visuals. 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.

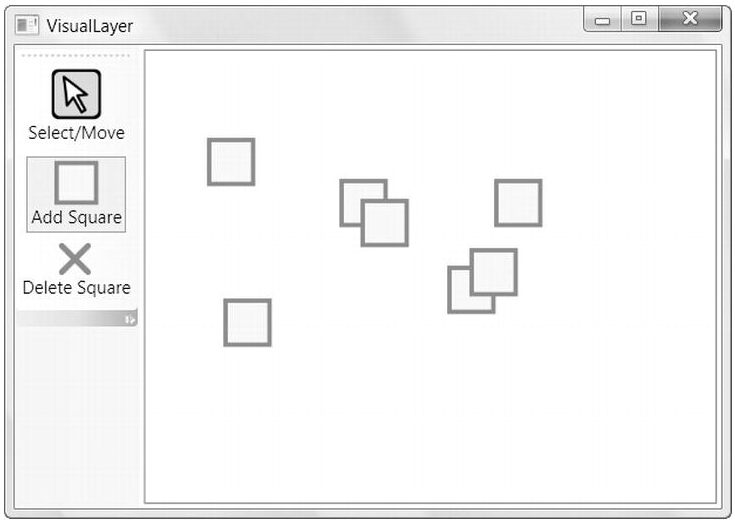

Figure 14-1. *Drawing visuals*

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 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 interested in just 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 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. 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 on top 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.

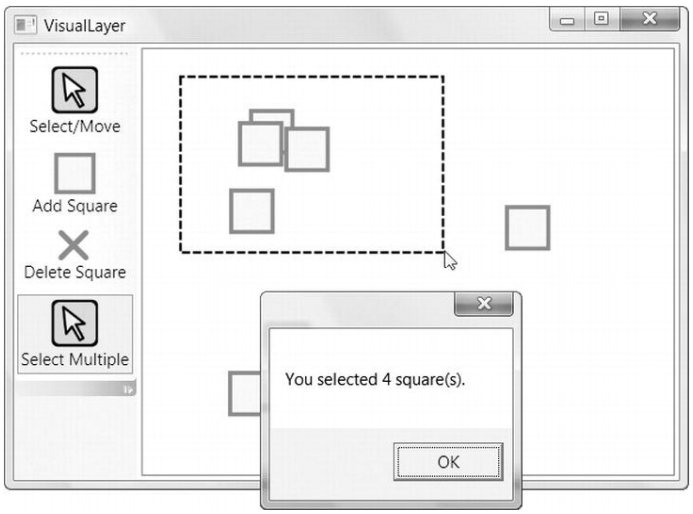

Figure 14-2. *Advanced hit testing*

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;
```

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 whether 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 selection 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 blurs (where Radius is 2, 5, and 20) applied to a stack of buttons.

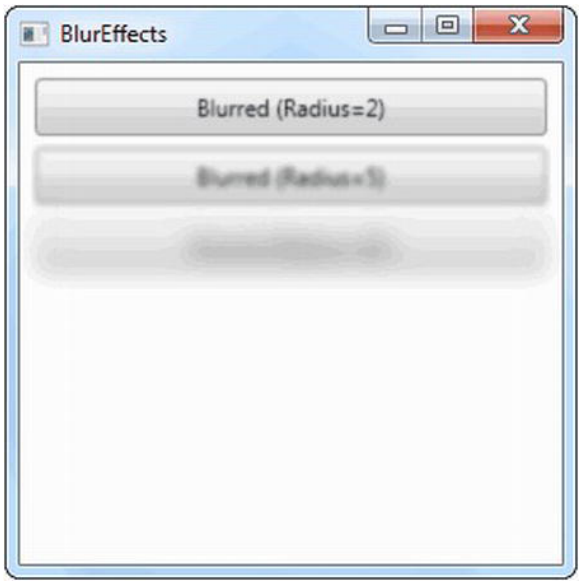

Figure 14-3. *Blurred buttons*

DropShadowEffect

DropShadowEffect adds a slightly offset shadow behind an element. You have several properties to play with, as listed in Table 14-3.

Table 14-3. *DropShadowEffect Properties*

Name	Description
Color	Sets the color of the drop shadow (the default is Black).
ShadowDepth	Determines how far the shadow is from the content, in pixels (the default is 5). Use a ShadowDepth of 0 to create an outer-glow effect, which adds a halo of color around your content.
BlurRadius	Blurs the drop shadow, much like the Radius property of BlurEffect (the default is 5).
Opacity	Makes the drop shadow partially transparent, using a fractional value between 1 (fully opaque, the default) and 0 (fully transparent).
Direction	Specifies where the drop shadow should be positioned relative to the content, as an angle from 0 to 360. Use 0 to place the shadow on the right side, and increase the value to move the shadow counterclockwise. The default is 315, which places it to the lower right of the element.

Figure 14-4 shows several drop-shadow effects on a TextBlock. Here's the markup for all of them:

```
<TextBlock FontSize="20" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Basic dropshadow</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect Color="SlateBlue"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Light blue dropshadow</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="White" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect BlurRadius="15"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Blurred dropshadow with white text</TextBlock.Text>
</TextBlock>

<TextBlock FontSize="20" Foreground="Magenta" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect ShadowDepth="0"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Close dropshadow</TextBlock.Text>
</TextBlock>
```

```
<TextBlock FontSize="20" Foreground="LimeGreen" Margin="5">
  <TextBlock.Effect>
    <DropShadowEffect ShadowDepth="25"></DropShadowEffect>
  </TextBlock.Effect>
  <TextBlock.Text>Distant dropshadow</TextBlock.Text>
</TextBlock>
```

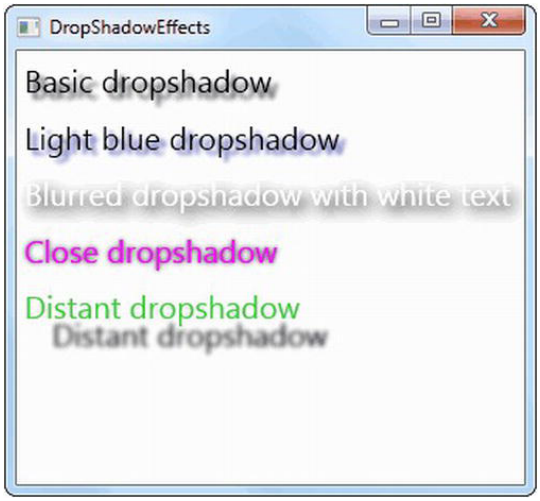

Figure 14-4. *Drop shadows*

There is no class for grouping effects, which means you can apply only a single effect to an element at a time. However, you can sometimes simulate multiple effects by adding them to higher-level containers (for example, using the drop-shadow effect for a TextBlock and then placing it in a Stack Panel that uses the blur effect). In most cases, you should avoid this workaround, because it multiplies the rendering work and reduces performance. Instead, look for a single effect that can do everything you need.

ShaderEffect

The ShaderEffect class doesn't represent a ready-to-use effect. Instead, it's an abstract class from which you derive your own custom pixel shaders. By using ShaderEffect (or a custom effect that derives from it), you gain the ability to go far beyond mere blurs and drop shadows.

Contrary to what you may expect, the logic that implements a pixel shader isn't written in C# code directly in the effect class. Instead, pixel shaders are written using High Level Shader Language (HLSL), which was created as part of Microsoft DirectX. (The benefit is obvious—because DirectX and HLSL have been around for many years, graphics developers have already created scores of pixel-shader routines that you can use in your own code.)

To create a pixel shader, you need to be able to write and compile HLSL code. To perform the compilation, you can use Microsoft's fxc.exe command-line tool, which is included with the Windows SDK for Windows 8 at `http://tinyurl.com/8ea7r43`. (Despite the name, the Windows SDK for Windows 8 supports Windows 7 as well.) A more convenient option is to use the free Shazzam tool (`http://shazzam-`

tool.com). Shazzam provides an editor for HLSL files, which includes the ability to try them on sample images. It also includes several sample pixel shaders that you can use as the basis for custom effects.

Although authoring your own HLSL files is beyond the scope of this book, using an existing HLSL file isn't. After you've compiled your HLSL file to a .ps file, you can use it in a project. Simply add the file to an existing WPF project, select it in the Solution Explorer, and set its Build Action to Resource. Finally, you must create a custom class that derives from ShaderEffect and uses this resource.

For example, if you're using a custom pixel shader that's compiled in a file named Effect.ps, you can use the following code:

```
public class CustomEffect : ShaderEffect
{
    public CustomEffect()
    {
        // Use the URI syntax described in Chapter 7 to refer to your resource.
        // AssemblyName;component/ResourceFileName
        Uri pixelShaderUri = new Uri("Effect.ps", UriKind.Relative);

        // Load the information from the .ps file.
        PixelShader = new PixelShader();
        PixelShader.UriSource = pixelShaderUri;
    }
}
```

You can now use the custom pixel shader in any window. First, make the namespace available by adding a mapping like this:

```
<Window xmlns:local="clr-namespace:CustomEffectTest" ...>
```

Now, create an instance of the custom effect class, and use it to set the Effect property of an element:

```
<Image>
  <Image.Effect>
    <local:CustomEffect></local:CustomEffect>
  </Image.Effect>
</Image>
```

You can get a bit more complicated than this if you use a pixel shader that takes certain input arguments. In this case, you need to create the corresponding dependency properties by calling the static RegisterPixelShaderSamplerProperty() method.

A crafty pixel shader is as powerful as the plug-ins used in graphics software such as Adobe Photoshop. It can do anything from adding a basic drop shadow to imposing more-ambitious effects such as blurs, glows, watery ripples, embossing, sharpening, and so on. Pixel shaders can also create eye-popping effects when they're combined with animation that alters their parameters in real time, as you'll see in Chapter 16.

■ **Tip** Unless you're a hard-core graphics programmer, the best way to get more-advanced pixel shaders isn't to write the HLSL yourself. Instead, look for existing HLSL examples or, even better, third-party WPF components that provide custom-effect classes. The gold standard is the free Windows Presentation Foundation Pixel Shader Effects Library at http://codeplex.com/wpffx. It includes a long list of dazzling effects such as swirls, color inversion, and pixelation. Even more useful, it includes transition effects that combine pixel shaders with the animation capabilities described in Chapter 15.

The WriteableBitmap Class

WPF allows you to show bitmaps with the Image element. However, displaying a picture this way is a strictly one-way affair. Your application takes a ready-made bitmap, reads it, and displays it in the window. On its own, the Image element doesn't give you a way to create or edit bitmap information.

This is where WriteableBitmap fits in. It derives from BitmapSource, which is the class you use when setting the Image.Source property (either directly, when you set the image in code, or implicitly, when you set it in XAML). But whereas BitmapSource is a read-only reflection of bitmap data, WriteableBitmap is a modifiable array of pixels that opens up many interesting possibilities.

■ **Note** It's important to realize that the WriteableBitmap isn't the best way for most applications to draw graphical content. If you need a lower-level alternative to WPF's element system, you should begin by checking out the Visual class demonstrated earlier in this chapter. For example, the Visual class is the perfect tool for creating a charting tool or a simple animated game. The WriteableBitmap is better suited to applications that need to manipulate individual pixels—for example, a fractal generator, a sound analyzer, a visualization tool for scientific data, or an application that processes raw image data from an external hardware device (for example, a webcam). Although the WriteableBitmap gives you fine-grained control, it's complex and requires much more code than the other approaches.

Generating a Bitmap

To generate a bitmap with WriteableBitmap, you must supply a few key pieces of information: its width and height in pixels, its DPI resolution in both dimensions, and the image format.

Here's an example that creates an image as big as the current window:

```
WriteableBitmap wb = new WriteableBitmap((int)this.ActualWidth,
  (int)this.ActualHeight, 96, 96, PixelFormats.Bgra32, null);
```

The PixelFormats enumeration has a long list of pixel formats, but only about half are considered writeable formats and are supported by the WriteableBitmap class. Here are the ones you can use:

Bgra32: This format (the one used in the current example) uses 32-bit sRGB color. That means that each pixel is represented by 32 bits, or 4 bytes. The first byte represents the contribution of the blue channel (as a number from 0 to 255). The second byte is for the green channel, the third is for the red channel, and the fourth is for the alpha value (where 0 is completely transparent and 255 is completely opaque). As you can see, the order of the colors (blue, green, red, alpha) matches the letters in the name *Bgra*32.

Bgr32: This format uses 4 bytes per pixel, just like Bgra32. The difference is that the alpha channel is ignored. You can use this format when transparency is not required.

Pbgra32: This format uses 4 bytes per pixel, just like Bgra32. The difference is the way it handles semitransparent pixels. In order to optimize the performance of opacity calculations, each color byte is *premultiplied* (hence the *P* in Pbgra32). This means each color byte is multiplied by the alpha value and divided by 255. So a partially transparent pixel that has the B, G, R, A values (255, 100, 0, 200) in Bgra32 would become (200, 78, 0, 200) in Pbgra32.

BlackWhite, Gray2, Gray4, Gray8: These are the black-and-white and grayscale formats. The number following the word *Gray* corresponds to the number of bits per pixel. Thus, these formats are compact, but they don't support color.

Indexed1, Indexed2, Indexed4, Indexed8: These are indexed formats, which means that each pixel points to a value in a color palette. When using one of these formats, you must pass the corresponding ColorPalette object as the last WriteableBitmap constructor argument. The number following the word *Indexed* corresponds to the number of bits per pixel. The indexed formats are compact, slightly more complex to use, and support far fewer colors—2, 14, 16, or 256 colors, respectively.

The top three formats—Bgra32, Bgr32, and Pbgra32—are by far the most common choices.

Writing to a WriteableBitmap

A WriteableBitmap begins with 0 values for all its bytes. Essentially, it's a big, black rectangle.

To fill a WriteableBitmap with content, you use the WritePixels() method. WritePixels() copies an array of bytes into the bitmap at the position you specify. You can call WritePixels() to set a single pixel, the entire bitmap, or a rectangular region that you choose. To get pixels out of the WriteableBitmap, you use the CopyPixels() method, which transfers the bytes you want into a byte array. Taken together, the WritePixels() and CopyPixels() methods don't give you the most convenient programming model to work with, but that's the cost of low-level pixel access.

To use WritePixels() successfully, you need to understand your image format and how it encodes pixels into bytes. For example, in the 32-bit bitmap type Bgra32, each pixel requires 4 bytes, one each for the blue, green, red, and alpha components. Here's how you can set them by hand, and then transfer them into an array:

```
byte blue = 100;
byte green = 50;
byte red = 50;
byte alpha = 255;

byte[] colorData = {blue, green, red, alpha};
```

Note that the order is critical here. The byte array must follow the blue, green, red, alpha sequence set out in the Bgra32 standard.

When you call WritePixels(), you supply an Int32Rect that indicates the rectangular region of the bitmap that you want to update. The Int32Rect wraps four pieces of information: the X and Y coordinate of the top-left corner of the update region, and the width and height of the update region.

The following code takes the colorData array shown in the preceding code and uses it to set the first pixel in the WriteableBitmap:

```
// Update a single pixel. It's a region starting at (0,0)
// that's 1 pixel wide and 1 pixel high.
```

```
Int32Rect rect = new Int32Rect(0, 0, 1, 1);

// Write the 4 bytes from the array into the bitmap.
wb.WritePixels(rect, colorData, 4, 0);
```

Using this approach, you could create a code routine that generates a WriteableBitmap. It simply needs to loop over all the columns and rows in the image, updating a single pixel in each iteration.

```
for (int x = 0; x < wb.PixelWidth; x++)
{
    for (int y = 0; y < wb.PixelHeight; y++)
    {
        // Pick a pixel color using a formula of your choosing.
        byte blue = ...
        byte green = ...
        byte red = ...
        byte alpha = ...

        // Create the byte array.
        byte[] colorData = {blue, green, red, alpha};

        // Pick the position where the pixel will be drawn.
        Int32Rect rect = new Int32Rect(x, y, 1, 1);

        // Calculate the stride.
        int stride = wb.PixelWidth * wb.Format.BitsPerPixel / 8;

        // Write the pixel.
        wb.WritePixels(rect, colorData, stride, 0);
    }
}
```

This code includes one additional detail: a calculation for the *stride*, which the WritePixels() method requires. Technically, the stride is the number of bytes required for each row of pixel data. You can calculate this by multiplying the number of pixels in a row by the number of bits in a pixel for your format (usually 4, as with the Bgra32 format used in this example), and then dividing the result by 8 to convert it from bits to bytes.

After the pixel-generating process is finished, you need to display the final bitmap. Typically, you'll use an Image element to do the job:

```
img.Source = wb;
```

Even after writing and displaying a bitmap, you're still free to read and modify pixels in the WriteableBitmap. This gives you the ability to build more-specialized routines for bitmap editing and bitmap hit testing.

Using More-Efficient Pixel Writing

Although the code shown in the previous section works, it's not the best approach. If you need to write a large amount of pixel data at once—or even the entire image—you're better off using bigger chunks That's because there's a certain amount of overhead for calling WritePixels(), and the more often you call it, the longer you'll delay your application.

Figure 14-5 shows a test application that's included with the samples for this chapter. It creates a dynamic bitmap by filling pixels with a mostly random pattern interspersed with regular gridlines. The downloadable code performs this task in two ways: using the pixel-by-pixel approach explained in the previous section and using the single-write strategy you'll see next. If you test this application, you'll find that the single-write technique is far faster.

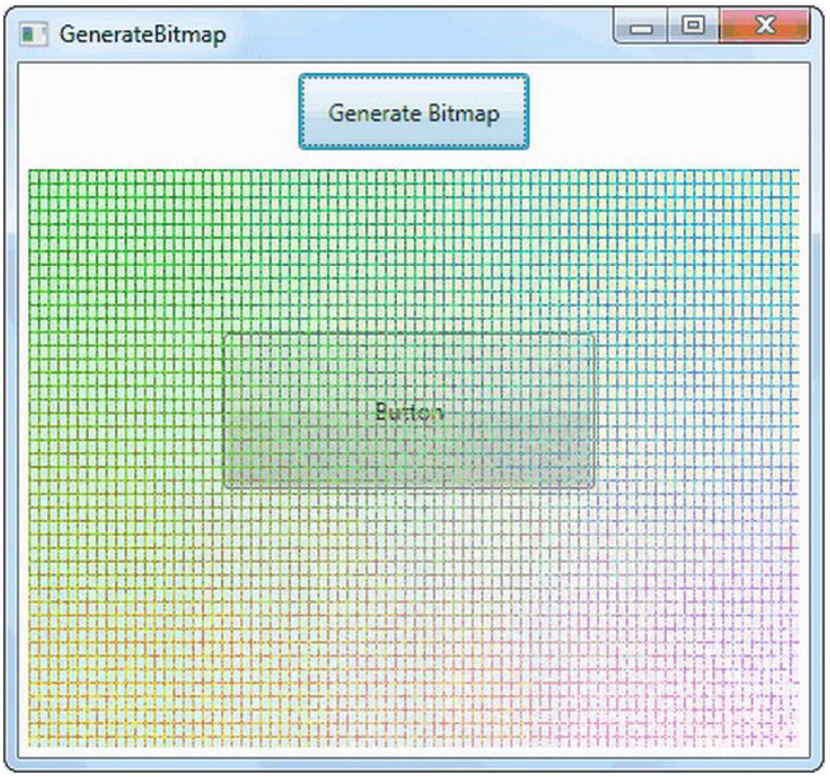

Figure 14-5. *A dynamically generated bitmap*

■ **Tip** For a more practical (and much longer) example of the WriteableBitmap at work, check out the example at `http://tinyurl.com/y8hnvsl`, which uses it to model a chemical reaction.

To update more than one pixel at once, you need to understand how the pixels are packaged together in your byte array. Regardless of the format you're using, your update buffer will hold a one-dimensional array of bytes. This array supplies values for the pixels in a rectangular section of the image, stretching from left to right to fill each row, and then from top to bottom.

To find a specific pixel, you need to use the following formula, which steps down the number of rows and then moves to the appropriate position in that row:

```
(y * wb.PixelWidth + x) * BytesPerPixel
```

For example, to set the pixel (40, 100) in a Bgra32 bitmap (which has 4 bytes per pixel), you use this code:

```
int pixelOffset = (40 + 100 * wb.PixelWidth) * wb.Format.BitsPerPixel/8;
pixels[pixelOffset] = blue;
pixels[pixelOffset + 1] = green;
pixels[pixelOffset + 2] = red;
pixels[pixelOffset + 3] = alpha;
```

With that in mind, here's the complete code that creates the bitmap shown in Figure 14-5, by first filling all the data in a single array, and then copying it to the WriteableBitmap with just one call to WritePixels():

```
// Create the bitmap, with the dimensions of the image placeholder.
WriteableBitmap wb = new WriteableBitmap((int)img.Width,
  (int)img.Height, 96, 96, PixelFormats.Bgra32, null);

// Define the update square (which is as big as the entire image).
Int32Rect rect = new Int32Rect(0, 0, (int)img.Width, (int)img.Height);

byte[] pixels = new byte[(int)img.Width * (int)img.Height *
  wb.Format.BitsPerPixel / 8];
Random rand = new Random();
for (int y = 0; y < wb.PixelHeight; y++)
{
    for (int x = 0; x < wb.PixelWidth; x++)
    {
        int alpha = 0;
        int red = 0;
        int green = 0;
        int blue = 0;

        // Determine the pixel's color.
        if ((x % 5 == 0) || (y % 7 == 0))
        {
            red = (int)((double)y / wb.PixelHeight * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)x / wb.PixelWidth * 255);
            alpha = 255;
        }
        else
        {
            red = (int)((double)x / wb.PixelWidth * 255);
            green = rand.Next(100, 255);
            blue = (int)((double)y / wb.PixelHeight * 255);
            alpha = 50;
        }

        int pixelOffset = (x + y * wb.PixelWidth) * wb.Format.BitsPerPixel/8;
        pixels[pixelOffset] = (byte)blue;
        pixels[pixelOffset + 1] = (byte)green;
        pixels[pixelOffset + 2] = (byte)red;
```

```
            pixels[pixelOffset + 3] = (byte)alpha;
        }

        // Copy the byte array into the image in one step.
        int stride = (wb.PixelWidth * wb.Format.BitsPerPixel) / 8;
        wb.WritePixels(rect, pixels, stride, 0);
    }

    // Show the bitmap in an Image element.
    img.Source = wb;
```

In a realistic application, you're likely to choose an approach that falls somewhere in between. You won't write one pixel a time if you need to update large sections of a bitmap, because that approach would probably be prohibitively slow. But you won't hold all of the image data in memory at once, because it could be very large. (After all, a 1000×1000 pixel image that requires 4 bytes per pixel needs nearly 4 MB of memory, which is not yet excessive but not trivial either.) Instead, you should aim to write large chunks of image data rather than individual pixels, especially if you're generating an entire bitmap at once.

■ **Tip** If you need to make frequent updates to the image data in a WriteableBitmap, and you want to make these updates from another thread, you can optimize the code even more by using the WriteableBitmap back buffer. The basic process is this: use the Lock() method to reserve the back buffer, obtain a pointer to the back buffer, update it, indicate the changed region by calling AddDirtyRect(), and then release the back buffer by calling Unlock(). This process requires unsafe code, and is beyond the scope of this book, but you can see a basic example in the Visual Studio help under the WriteableBitmap topic.

The Last Word

In this chapter, you looked at three topics that go beyond WPF's standard 2-D drawing support. First, you tackled the lower-level visual layer, which is the most efficient way to display graphics in WPF. Using the visual layer, you saw how you could build a basic drawing application that uses sophisticated hit testing. Next, you learned about pixel shaders, a way to fuse graphical effects originally designed for next-generation games into any WPF application. Not only are pixel shaders nearly effortless to use, but there's already a huge library of free pixel shaders that you can drop into your applications right now. Finally, you considered the WriteableBitmap, a powerful but more limited tool that lets you create a bitmap image, and directly manipulate the individual pixels that compose it.