Essential Silverlight 3

Ashraf Michail

Library of Congress Cataloging-in-Publication Data
Michail, Ashraf.
Essential Silverlight 3 / Ashraf Michail.
p. cm.
Includes index.
ISBN 978-0-321-55416-1 (pbk. : alk. paper) 1. Silverlight (Electronic resource) 2. Multimedia systems. 3. Websites—Design. 4. Application software—Development.
I. Title.

QA76.575.M52187 2009 2009
006.7—dc22

2009026788

ISBN-13: 978-0-321-55416-1
ISBN-10: 0-321-55416-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Second printing, December 2009

Foreword

SILVERLIGHT IS A cross-browser, cross-platform plugin that enables developers to build rich media and RIA Web experiences. It allows developers to use .NET within the browser to create scenarios that can't be achieved using any other Web technology, and to use high productivity tools like Visual Studio and Expression Studio when doing so.

The new Silverlight 3 release delivers more than 200 new features for developers to leverage, including the ability to run Silverlight applications in or out of the browser; stream HD and H.264 video; display 3D graphics and use GPU acceleration; take advantage of richer data-binding support; use new bitmap APIs and animation easing capabilities; and much, much more.

Ashraf Michail, the author of the book you are holding in your hands, is one of the star developers of the Silverlight team. He has been involved in the architecture since we first began working on it and brought to the effort his deep experience with and understanding of Silverlight's "big brother," the Windows Presentation Foundation. Ashraf knows intimately the ins and outs of the Silverlight runtime, and in this book he has distilled his understanding into a clear, concise exposition.

This isn't the only book on Silverlight 3, by any means: I'm happy to say that several other good Silverlight 3 books are in the works. However, this is the only book that focuses on the Silverlight runtime and gives you an in-depth understanding of how it works. Once you know what's really happening in the runtime engine, you'll be in a position to take full advantage of Silverlight's speed and power. Look for the "Under the Hood" section at the end of each chapter of this book: You won't find that anywhere else.

Also look for the "Technical Insight," "Debugging Tip," and "Performance Tip" sections scattered throughout the book. These are the nuggets that you can take away and use to make your own applications fly. Is too much data slowing down your ComboBox display? You can filter your Items Source to speed that up, as explained in Chapter 10. Would your application benefit from GPU acceleration? Chapter 12 shows you how to make that happen, explains the four stages of the GPU acceleration pipeline—and explains the tradeoffs involved in GPU acceleration on each supported operating system so that you can understand why turning on GPU acceleration might not always help.

The book you are holding contains the keys to writing great Silverlight 3 applications. Use them well!

—Scott Guthrie,
corporate vice president,
Microsoft Developer Division

Preface

Silverlight is a rapidly growing Web technology designed to deliver media and rich Internet applications on multiple operating systems and Web browsers. Silverlight consists of a runtime, a set of tools, and libraries for your applications. End users can download the free runtime from http://www.silverlight.net to run Silverlight applications.

You can develop Silverlight applications with Visual Studio using familiar .NET languages. Designers can create artwork and animation for Silverlight using tools such as Expression Blend and Expression Design or import assets from other popular design tools. You can create video playback experiences with tools such as Expression Media Encoder or image zoom experiences with Deep Zoom Composer.

In addition to the default libraries built into the Silverlight runtime, you can download a variety of libraries from http://www.codeplex.com/Silverlight. You can use these libraries in your applications, modify the source code, or use it as an example for how to write Silverlight components.

In this book, you will learn how to build a Silverlight application and use features such as graphics, text, input, animation, layout, media, controls, and data binding. In addition to information on how to use Silverlight features, this book describes feature design principles and how those features work "under the hood." The design principles help you understand whether the feature meets the needs of your application. The "under the hood" information gives you a look at how the feature is implemented in Silverlight so that you can get the most out of Silverlight. For example, the

Under the Hood sections give you a deeper understanding of the behavior, performance characteristics, and limits of a feature.

Who Should Read This Book?

If you want to get started writing your first Silverlight applications or if you are an expert Silverlight developer, you will learn a lot from this book. For the beginner, this book explains the concepts required to write your applications. For the advanced Silverlight developer, you will find in-depth information on how Silverlight works "under the hood" and how to use that information to get the most out of Silverlight.

Prerequisites

Before reading this book, you should be familiar with how to build a basic Web page and have a working knowledge of how to write .NET applications in C#.

Organization

Each chapter in this book contains three sections: Principles, Features, and Under the Hood. The Principles indicate the feature design goals to help you determine if that feature will meet your application needs. The Features section describes how you can use the feature. The Under the Hood section describes how the feature is implemented in Silverlight and how you can use that information to improve your applications. Throughout the book, there are many Performance Tips and Technical Insights.

The book consists of the following 12 chapters:

- Chapter 1, "Silverlight Overview," is an introduction to Silverlight and the content in the remainder of the book. Chapter 1 highlights the features that are new in Silverlight 3 and not available in Silverlight 2.
- Chapter 2, "Applications," explains how to create and deploy a Silverlight application including an explanation of all the critical components.

- Chapter 3, "Graphics," presents an in-depth explanation of all the graphics primitives.
- Chapter 4, "Text," explains how to display high quality text.
- Chapter 5, "Input Events," provides information on using mouse and keyboard input in your application.
- Chapter 6, "Animation," describes how to add animations to your applications.
- Chapter 7, "Layout," describes how to use the layout system to automatically size and position elements in your application.
- Chapter 8, "Media," explains how you can make a video player with Silverlight.
- Chapter 9, "Controls," explains how you can use styles, customize built-in controls, and create your own customizable controls.
- Chapter 10, "Data Binding," describes how you can connect data to your application user interface controls.
- Chapter 11, "Effects," describes how to use built-in effects and create your own custom effects. Effects are a new feature in Silverlight 3.
- Chapter 12, "GPU Acceleration," explains how to use your graphics process unit (GPU) to improve the performance of animation and video playback. GPU acceleration is a new feature in Silverlight 3.

3
Graphics

IN THIS CHAPTER, you will learn how to add rich vector graphics and images to your application. You will also learn how to optimize performance and image quality of those graphics elements. In particular, Chapter 3 will discuss the following:

- The graphics system design principles
- The elements for displaying graphics
- The problems the Silverlight runtime solves "under the hood" and the problems your application must solve

Graphics Principles

The Silverlight graphics API makes it easy to add vector graphics, bitmap images, and text to your applications. This section describes the graphics API design principles.

Vector Graphics and Bitmap Images

Bitmap images are a common method of adding graphics to an application. However, bitmap images become blurry when scaled up, as shown in Figure 3.1, or aliased when scaled down, as shown in Figure 3.2. Unlike a bitmap image, if you scale a vector graphic, it will remain sharp as shown in Figure 3.3. Both vector graphics and bitmap images are useful in most

FIGURE 3.1: Scaling up an image of a circle

FIGURE 3.2: Scaling down an image of a circle

FIGURE 3.3: Scaling a vector graphic circle

applications. For example, a user interface control looks better at different sizes and resolutions with vector graphics instead of bitmap images. Bitmap images are useful for displaying content that is not easily expressible in vector form such as digital photographs or visual effects not supported by the runtime.

New in Silverlight 3

There are a number of techniques for scaling down an image to produce a result better than that shown in Figure 3.2. Silverlight 3 converts your bitmap to a set of smaller images at various sizes using a better algorithm. This conversion happens once per bitmap and Silverlight caches the result of the conversion for the lifetime of your bitmap. During an animation, Silverlight 3 dynamically selects the right resolution bitmap to display. This process can increase memory usage by 33%, but substantially improves the display quality of scaled down images and 3D transforms to get a result close to that shown in Figure 3.3.

When scaling up bitmaps in Silverlight 3, the image remains blurry as shown in Figure 3.1.

Retained Mode

There are two types of graphics API: retained mode and immediate mode. A retained mode API automatically responds to changes to a graph of objects. An immediate mode API requires you to issue all the draw commands to describe a change. For example, to remove the rectangle shown in Figure 3.4 in a retained mode system, simply call a function to remove that element. The retained mode graphics system is responsible for redrawing the background, the triangle beneath, and the ellipse above. To remove the same rectangle shown in Figure 3.4 with an immediate mode API, you need to make three calls to draw the background, the triangle beneath it, and the ellipse above as shown in Figure 3.5.

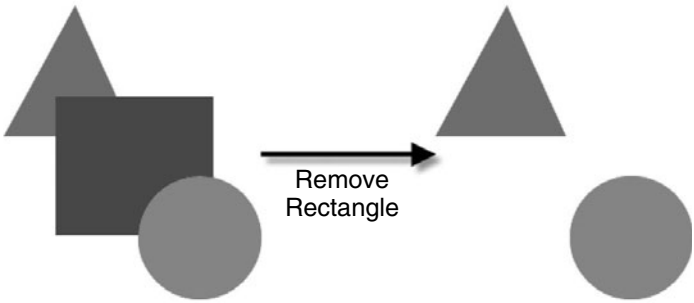

FIGURE 3.4: Removing a rectangle with a retained mode API

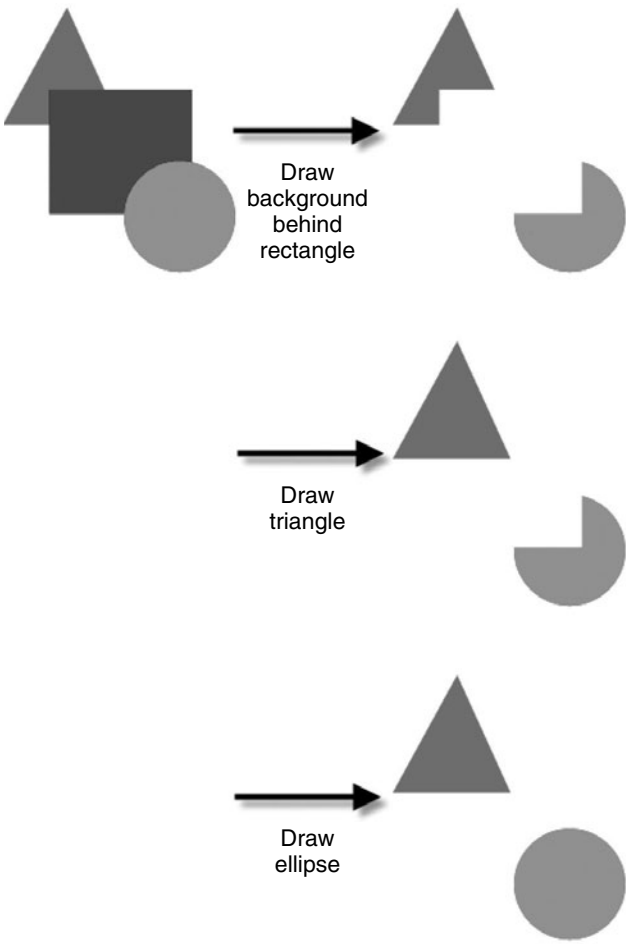

FIGURE 3.5: Removing a rectangle with an immediate mode API

A retained mode API enables you to do the following:

- Construct a set of graphics elements
- Change properties of the graphics elements
- Build a graph representing the relationship between those elements
- Manipulate the graph structure

A retained mode graphics API is easier to use than an immediate mode API and enables the underlying system to provide automatic performance optimizations such as drawing incrementally and avoiding the drawing of occluded shapes. Silverlight provides a retained mode system to optimize for ease of use, animating vector graphics content, and for building applications composed of UI controls.

In Silverlight, you can construct the retained mode graph declaratively with a XAML file:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- triangle -->
    <Path
        Fill="Green"
        Data="F1 M 128,12L 12,224L 224,224"
    />

    <!-- rectangle -->
    <Rectangle
        Fill="Blue"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

    <!-- circle -->
    <Ellipse
        Fill="Red"
        Canvas.Left="230"
        Canvas.Top="288"
        Width="200"
        Height="200"
    />

</Canvas>
```

Alternatively, you can construct the retained mode graph with code:

```
Canvas canvas = new Canvas();

//
// Make the triangle
//

Path path = new Path();

path.Fill = new SolidColorBrush(Colors.Green);

PathFigure pathFigure = new PathFigure();

pathFigure.StartPoint = new Point(128, 12);

LineSegment lineSegment1 = new LineSegment();
lineSegment1.Point = new Point(12, 224);
pathFigure.Segments.Add(lineSegment1);

LineSegment lineSegment2 = new LineSegment();
lineSegment2.Point = new Point(224, 224);
pathFigure.Segments.Add(lineSegment2);

PathGeometry pathGeometry = new PathGeometry();
pathGeometry.Figures = new PathFigureCollection();

pathGeometry.Figures.Add(pathFigure);

path.Data = pathGeometry;

canvas.Children.Add(path);

//
// Make the rectangle
//

Rectangle rectangle = new Rectangle();

rectangle.Fill = new SolidColorBrush(Colors.Blue);
Canvas.SetLeft(rectangle, 96);
Canvas.SetTop(rectangle, 160);
rectangle.Width = 256;
rectangle.Height = 224;

canvas.Children.Add(rectangle);

//
// Make the circle
//
```

```
Ellipse ellipse = new Ellipse();

ellipse.Fill = new SolidColorBrush(Colors.Red);
Canvas.SetLeft(ellipse, 230);
Canvas.SetTop(ellipse, 288);
ellipse.Width = 200;
ellipse.Height = 200;

canvas.Children.Add(ellipse);
```

Cross Platform Consistency

An important goal for the Silverlight graphics engine is to enable a developer to write his or her application once and have it run consistently across a variety of operating systems and browsers. Each operating system has a local graphics library. However, these local operating system graphics libraries differ significantly in feature set, performance, and image quality. To ensure cross-platform consistency and performance, Silverlight includes its own rendering engine.

Tools

Silverlight is capable of loading vector and image content from designer tools and integrating with developer written application code. For vector graphics and animation, you can use Expression Design and Expression Blend to generate XAML content for use with the Silverlight runtime. There are also a variety of free XAML exporters available to generate XAML content including an Adobe Illustrator exporter, an XPS print driver, and several others.

Balancing Image Quality and Speed

In addition to displaying static XAML, Silverlight provides real-time animation at 60 frames per second. However, real-time performance is highly dependent on the application content, the specific hardware configuration of the target machine, the resolution of the target display, the operating system, and the hosting Web browser.

When an application does not meet the 60 frame per second goal, the Silverlight team uses the following two options to improve performance:

1. Make optimizations to components in the Silverlight runtime
2. Lower image quality to achieve better speed

Reducing image quality for speed is the most controversial optimization technique. The Silverlight application must look good. However, it is possible to trade off image quality in a manner that is generally acceptable to the end user. For example, vector graphics rendering makes some quality sacrifices but still maintains an acceptable visual bar. On the other hand, end users have a much higher standard for text quality and the Silverlight runtime spends more CPU time rendering text clearly.

Graphics Elements

As previously discussed, the Silverlight runtime can load and draw vector graphics XAML on a variety of platforms. The graphics elements can also be specified programmatically when you use C# or JavaScript. This section describes the elements you can use to display vector graphics and images in your applications.

Shapes

This section starts the graphics element discussion with the base class for all graphics primitives: the `Shape` element. The `Shape` element provides the following:

- A `Fill` property to specify how the shape interior is drawn
- A `Stretch` property to indicate how a `Shape` element stretches to a specified `Width` and `Height`
- `Stroke` and `StrokeThickness` properties to specify the pen behavior

The elements that derive from `Shape` define the shape's geometry. These elements include `Rectangle`, `Ellipse`, `Line`, `Polygon`, `Polyline`, and `Path`.

Rectangle

To draw a rectangle at a specified position, place it in a `Canvas` element and specify its `Canvas.Top`, `Canvas.Left`, `Width`, `Height`, and `Fill` color:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

You can add an outline to the rectangle as shown in Figure 3.6 by setting the `Stroke` property to specify the color and the `StrokeThickness` property to specify the thickness:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

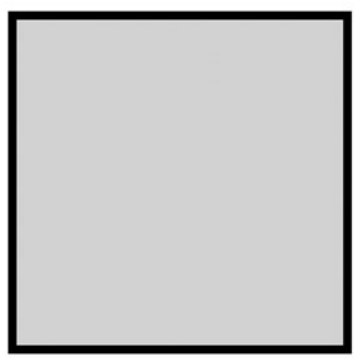

FIGURE 3.6: Rectangle element with an outline

You can use the `Rectangle` element to draw the rounded rectangles shown in Figure 3.7 by specifying the `RadiusX` and `RadiusY` properties:

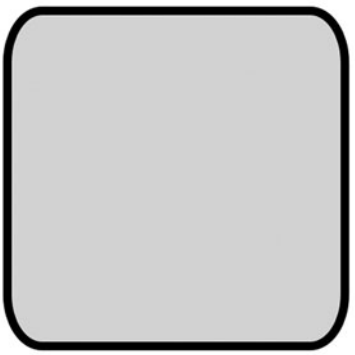

FIGURE 3.7: Rectangle element with rounded corners

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
        RadiusX="40"
        RadiusY="60"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

Ellipse

As with the `Rectangle` element, you can position an `Ellipse` element with the same `Canvas.Top`, `Canvas.Left`, `Width`, and `Height` properties. Silverlight stretches the ellipse to fit the specified bounds as shown in Figure 3.8.

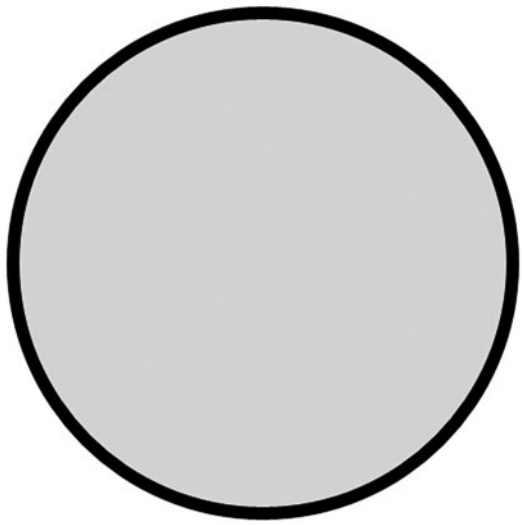

FIGURE 3.8: Ellipse element

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

Line

To draw a line, you can use the `Line` element and set its `X1`, `Y1`, `X2`, `Y2` properties. As with all other shapes, you can use the `Stroke` property to specify the fill color and the `StrokeThickness` property to specify the thickness as shown in Figure 3.9.

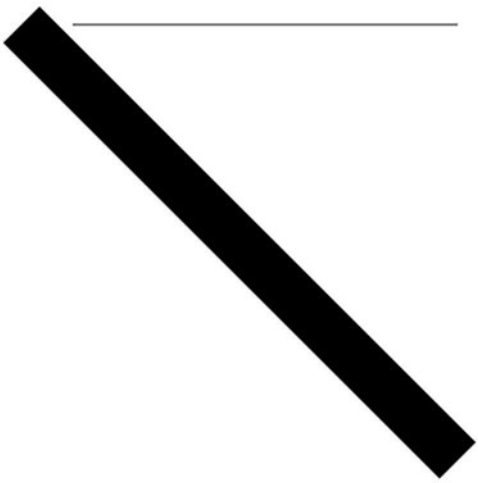

FIGURE 3.9: Line element

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- thick diagonal line -->
    <Line
        Stroke="Black"
        StrokeThickness="40"
        X1="60"
        Y1="60"
        X2="400"
        Y2="400"
    />

    <!-- one pixel horizontal line -->
    <Line
        Stroke="Black"
        StrokeThickness="1"
        X1="100"
        Y1="60"
        X2="400"
        Y2="60"
    />

</Canvas>
```

If you look closely at the pixels for the horizontal line shown in Figure 3.10, you will see it has a height of two pixels despite the one pixel `StrokeThickness`

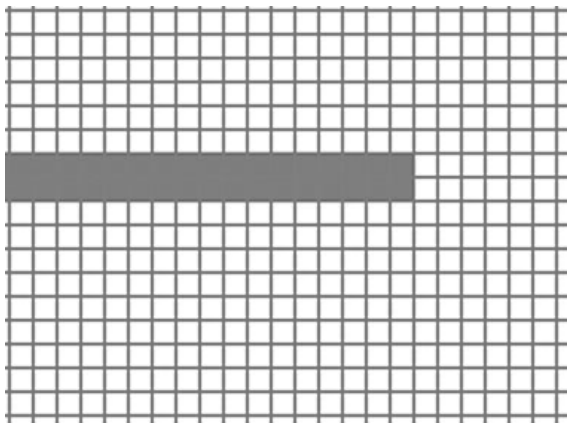

FIGURE 3.10: Pixels rendered for a Line element

specified in the XAML. Furthermore, the line is gray instead of the specified black color. To understand this rendered result, consider the following equivalent `Rectangle` element:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- one pixel horizontal line drawn as a rectangle -->
    <Rectangle
        Fill="Black"
        Canvas.Left="99.5"
        Canvas.Top="59.5"
        Width="301"
        Height="1"
    />

</Canvas>
```

The previous `Rectangle` element has half integer values for its position. The reason for the half pixel coordinates is that the line expands by half `StrokeThickness` in either direction. Because `StrokeThickness` is equal to one, the line adjusts the top and left coordinates by -0.5. Because the rectangle is between two pixels, it gets anti-aliased and occupies two pixels with a lighter color. If you want sharp horizontal and vertical lines, you should draw a rectangle positioned at integer coordinates to get the result shown in Figure 3.11.

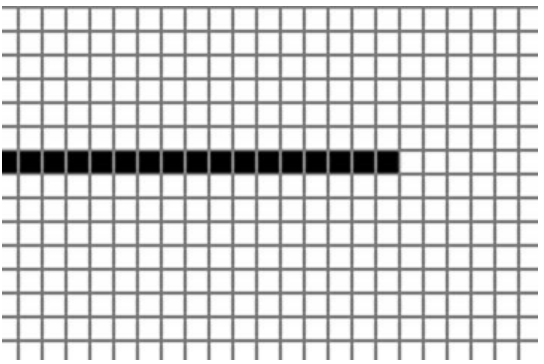

FIGURE 3.11: Sharp horizontal line drawn with a Rectangle element

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- one pixel horizontal line drawn as a rectangle -->
    <Rectangle
        Fill="Black"
        Canvas.Left="99"
        Canvas.Top="59"
        Width="301"
        Height="1"
    />

</Canvas>
```

Path

The `Path` element extends the `Shape` element by providing a `Data` property that specifies the geometry object. The `Rectangle`, `Ellipse`, and `Line` elements previously discussed are all expressible with the more general `Path` element. For example, instead of specifying a `Rectangle` element, we can specify a `Path` element with a `RectangleGeometry`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        Fill="Blue"
```

```
        Stroke="Black"
        StrokeThickness="10"
    >
        <Path.Data>
            <RectangleGeometry Rect="96,160,256,224"/>
        </Path.Data>
    </Path>

</Canvas>
```

The `Path` element syntax is more verbose than the specialized shapes syntax. However, because Silverlight converts all shapes internally to `Path` elements, if you understand how the `Path` element draws you will understand how all shapes draw.

In addition to expressing specialized shapes, the `Path` element can express a geometry consisting of a collection of `Figure` elements. A `Figure` element is a connected set of line segments and Bezier segments. The most common method to specify these figures, line segments, and curves is the path mini-language. For example, to draw the shape in Figure 3.12 you would do the following:

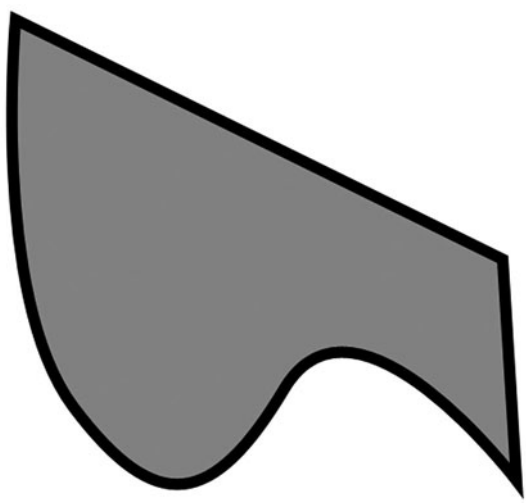

Figure 3.12: Example path

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="Red"
        Data="M 14,16
              C 14,16 -8,256 64,352
              C 136,448 185,440 247,336
              C 309,233 448,416 448,416
              L 436,224
              Z"
    />

</Canvas>
```

The commands supported by the mini-language include those shown in Table 3.1. Each command is followed by the action it takes.

An alternative form of specifying the geometry is to use the expanded XAML syntax:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="Red"
    >
        <Path.Data>
            <PathGeometry>
               <PathGeometry.Figures>
                  <PathFigure StartPoint="14, 16" IsClosed="true">
                     <PathFigure.Segments>
                        <BezierSegment
                            Point1="14,16"
                            Point2="-8,256"
                            Point3="64,352"
                        />
                        <BezierSegment
                            Point1="136,448"
                            Point2="185,440"
                            Point3="247,336"
                        />
                        <BezierSegment
                            Point1="309,233"
                            Point2="448,416"
                            Point3="448,416"
```

```
                        />
                        <LineSegment
                            Point="436,224"
                        />
                    </PathFigure.Segments>
                </PathFigure>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>

</Canvas>
```

TABLE 3.1: Path Mini-language Commands

Command	Action
`M x,y`	Move to position *x,y*
`L x,y`	Draw a line from the current position to position *x,y*
`C x1,y1, x2,y2, x3,y3`	Draw a cubic Bezier segment with control points consisting of the current position, (*x1,y1*), (*x2,y2*), and (*x3,y3*)
`Q x1,y1, x2,y2`	Draw a quadratic Bezier segment with control points consisting of the current position, (*x1,y1*), and (*x2,y2*)
`H x`	Draw a horizontal line from the current position *x0,y0* to position *x,y0*
`V y`	Draw a vertical line from the current position *x0,y0* to position *x0,y*
`Z`	Close a figure
`F0`	Specify EvenOdd fill rule
`F1`	Specify NonZero fill rule

One additional concept previously discussed is that of a `Figure` element. Because the `Path` element can have more than one figure, it can create an empty space in the middle of a filled space as shown in Figure 3.13.

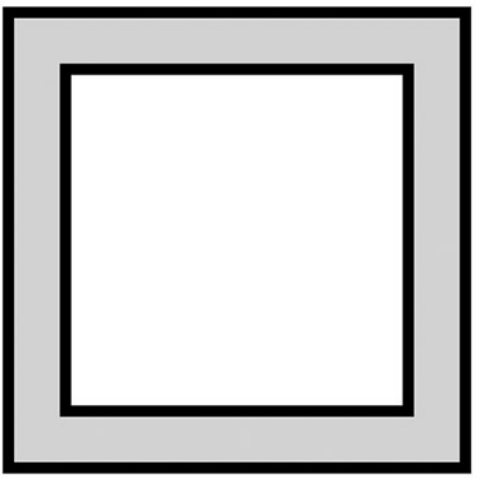

FIGURE 3.13: Path with an empty space in the center

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

PERFORMANCE TIP

These two forms of specifying a path may render identically, but they differ in performance and flexibility. The mini-language form parses faster, consumes less memory, and draws fastest at runtime. The mini-language is the recommended form for static content. However, it is not possible to bind an animation or change a segment property of a path generated with the mini-language because Silverlight does not create the path, figure, or segment API objects.

Fill Rules

The previous section explained how to use geometry to specify an outline of an area to fill. However, an outline does not uniquely specify the inside

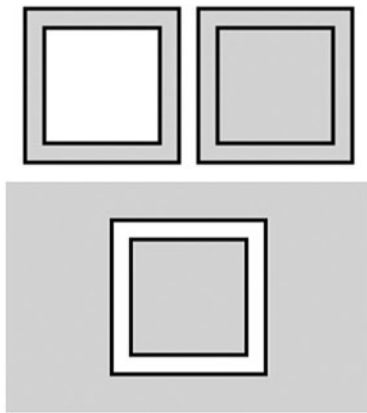

FIGURE 3.14: Different fills for the same outline

and outside of the shape. For example, the outline in Figure 3.13 could generate any of the rendering results shown in Figure 3.14. A fill rule is a method to distinguish the inside of a shape from the outside of the shape.

One approach to specifying what is inside a shape is to cast a horizontal line through the shape and count the number of edges crossed from left to right. If the number of edges crossed is even, classify the horizontal line segment as outside the shape. If the number of edges is odd, classify that segment as inside the shape. This fill rule is the `EvenOdd` rule and is the default fill mode for `Path` elements. To specify the fill mode explicitly, you can specify the `FillRule` property on geometry or use `F0` for `EvenOdd` from the path mini-language:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F0 = EvenOdd -->

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F0
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

An alternative rule is the NonZero rule, which considers the order points are specified in the input. If an edge goes up in the y direction, assign that edge a + 1 winding number. If an edge goes down in the y direction, assign that edge a – 1 winding number. The NonZero rule defines the interior of the shape to be all those segments where the sum of winding numbers from the left to the current segment is not zero. For example, if you specify the geometry shown in Figure 3.14 with the point order in the following markup, it would result in the winding numbers and filled segments shown in Figure 3.15.

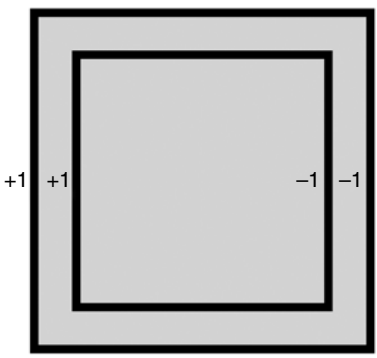

Figure 3.15: Winding mode numbers resulting in a filled center space

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F1 = NonZero -->

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F1
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

If you specify the points in the following order, the shape would render differently as shown in Figure 3.16.

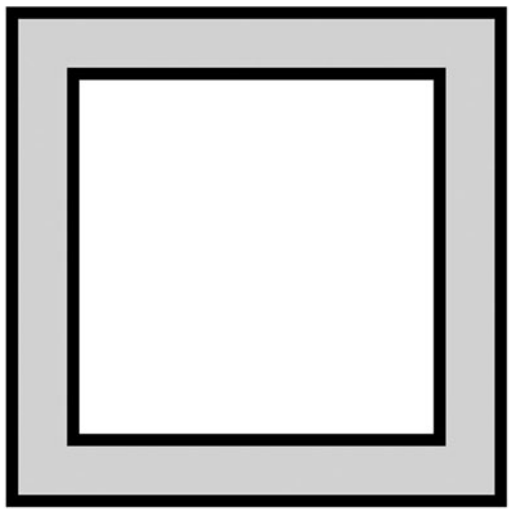

FIGURE 3.16: Different fill as a result of a different point order

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F1 = NonZero -->

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F1
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 400,100 400,400 100,400 100,100 z"
    />

</Canvas>
```

PERFORMANCE TIP

The `NonZero` rule is more complicated than the `EvenOdd` rule and does render slower. For most vector graphics fills, the `EvenOdd` rule gives the desired result.

Images

In addition to the vector graphics elements previously discussed, the other fundamental graphics primitive is the `Image` element. To display an image,

you can use the `Image` element with a reference to a URI to produce the result shown in Figure 3.17.

Figure 3.17: Image element example

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Image Source="silverlight.png"/>
</Canvas>
```

The `Source` property can reference any image in JPG or PNG format. However, if the JPG or PNG contains DPI (dots per inch) information, Silverlight ignores this information because it is usually not accurate for display purposes. All references to URIs in XAML are relative to the location of the XAML file. For example, if the XAML file is in a XAP, Silverlight searches for silverlight.png in the XAP. If the XAML file is a resourced in a managed assembly, Silverlight searches for silverlight.png in that same assembly.

If you do not specify the `Width` and `Height` properties of an image, Silverlight draws the image at its natural size, which results in a pixel perfect rendering of the original image data.

Brushes

All of the previous examples filled the `Path` element pixels with a single color. Silverlight also supports filling arbitrary shapes with image brushes, linear gradient brushes, and radial gradient brushes. A brush is a mathematical function that maps an (x,y) position to a color. For example, `SolidColorBrush` is a simple function that ignores its position and always outputs the same color. This section describes the brushes available in Silverlight and includes the function used to map from screen position to color.

Technical Insight

As shown in the previous example, the `Source` property of an `Image` element can be set to a URI. Many references to the same `Source` URI cause Silverlight to download the image once and use it multiple times. If you remove all references to `Image` elements for a specific URI, Silverlight removes the image from the memory cache and a future reference causes Silverlight to initiate another download. Future downloads may be serviced from the browser cache or go over the network if the image is no longer in the browser cache.

Solid Color Brush

`SolidColorBrush` returns a single color for all screen positions. When you specify a `Fill` or `Stroke` property value, Silverlight creates a solid color brush for the corresponding shape fill or pen stroke respectively:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="Blue"
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

</Canvas>
```

Alternatively, you can specify a brush with the expanded XAML syntax:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        StrokeThickness="10"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    >
        <Rectangle.Fill>
            <SolidColorBrush Color="Blue"/>
        </Rectangle.Fill>
        <Rectangle.Stroke>
```

```
            <SolidColorBrush Color="Black"/>
        </Rectangle.Stroke>
    </Rectangle>

</Canvas>
```

The previous examples specified a named color. You can also specify a color explicitly by providing a hex string of the form #aarrggbb, which represents a hex alpha channel value, red channel value, green channel value, and blue channel value. For example, opaque green would be #ff00ff00.

From C#, you can specify a color by creating an instance of the `Color` class:

```
Color green = Color.FromArgb(0xff, 0x0, 0xff, 0x0);
```

The alpha channel specifies the degree of transparency where `0x0` indicates completely transparent, `0xff` indicates an opaque color, and intermediate values indicate partial transparency. A brush with a transparent color will blend its color to the background color using the following formula:

$$\text{Color\_destination} = (\text{alpha} * \text{Color\_source} + (0\text{xff} - \text{alpha}) * \text{Color\_destination})/256$$

Silverlight will pass colors specified in hex format to the Web browser without a color space conversion. Typically, the browser will interpret the color as a standard RGB color space color (sRGB) that is, an 8-bit per channel 2.2 implied gamma space color. However, the visible color may vary with operating systems, Web browser, the monitor, and the operating system color profile. An alternative form of specifying colors is the floating point wide gamut linear RGB color space (scRGB):

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="sc#1, 1.0, 0.0, 0.0"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

</Canvas>
```

Silverlight converts scRGB colors to sRGB internally for blending. Consequently, specifying colors in native sRGB is desirable to avoid extra color conversion steps.

Gradient Brushes

Gradient brushes define a set of colors and positions along a virtual gradient line. The function that maps screen position to color first maps the screen position to a point along the gradient line and then interpolates a color based on the two nearest points.

Consider the following use of a `LinearGradientBrush`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
        <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

The preceding linear gradient brush produces the fill shown in Figure 3.18. A linear gradient brush maps a screen position to the point on the line

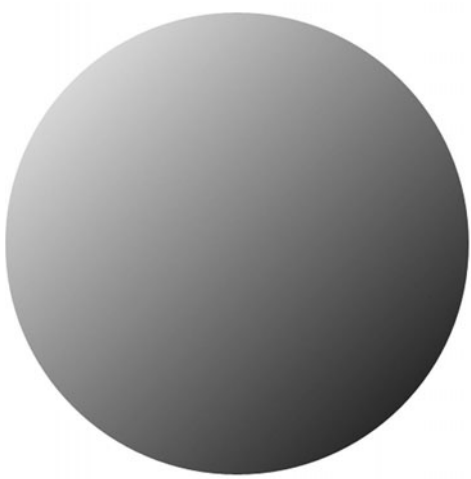

FIGURE 3.18: Linear gradient brush

closest to that position. The brush then interpolates a color based on the two nearest points specified along the line as shown in Figure 3.18.

Alternatively, you can specify a radial gradient fill using `RadialGradientBrush` that takes the distance from the screen position to the center of the radial gradient and maps that distance to the specified gradient line of colors and positions. For example, the following XAML generates the rendering shown in Figure 3.19.

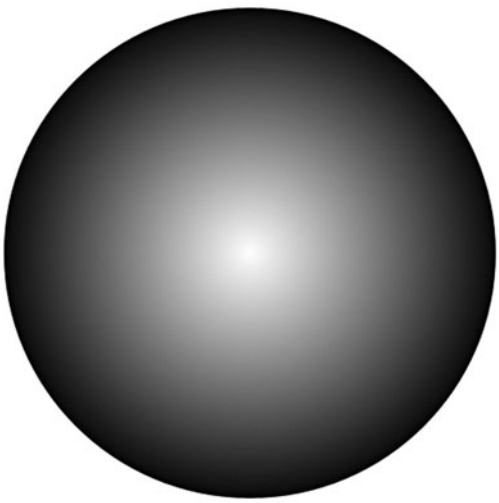

Figure 3.19: Radial gradient brush

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
        <Ellipse.Fill>
            <RadialGradientBrush>
                <RadialGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

Another feature of `RadialGradientBrush` is the capability to move the point that maps to the start of our gradient line. In particular, in our previous example, the center of the radial gradient circle mapped to the start of our gradient line and the radius of the gradient circle mapped to the end of our gradient line. With this pattern, you always get a uniform ellipse. You can move the center using the `GradientOrigin` property to get the result shown in Figure 3.20.

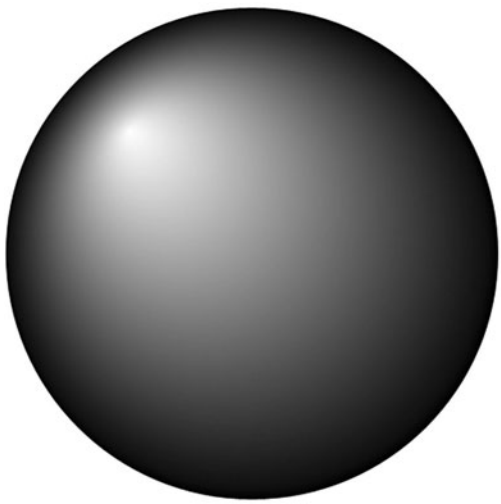

Figure 3.20: Focal gradient brush

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
        <Ellipse.Fill>
            <RadialGradientBrush GradientOrigin="0.25 0.25">
                <RadialGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

One other feature of linear and radial gradients is the capability to specify the behavior when the display position maps to some position outside the range of the gradient line. The `SpreadMethod` property defines that behavior. The `Pad` mode repeats the closest point when off the line, the `Reflect` mode mirrors to a point on the line, and the `Repeat` mode simply takes the position modulo the length of the line as shown in Figure 3.21.

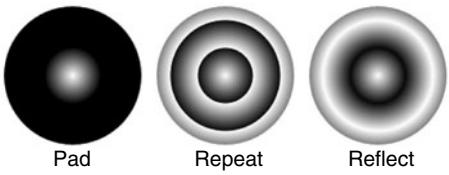

FIGURE 3.21: SpreadMethod example

Image Brushes

The role of the image brush is to map a screen position to a pixel in the specified image. For example, the following XAML would result in the image brush rendering shown in Figure 3.22.

FIGURE 3.22: ImageBrush example

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
```

```
        Width="450"
        Height="450"
        Stroke="Black"
        StrokeThickness="10"
    >
        <Ellipse.Fill>
            <ImageBrush ImageSource="silverlight.png"/>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

Strokes

The previous section showed how to use a brush to color the fill of a shape. You can also use a brush to add color to the outline of a shape by setting the stroke properties. For example, the following XAML generates the output shown in Figure 3.23.

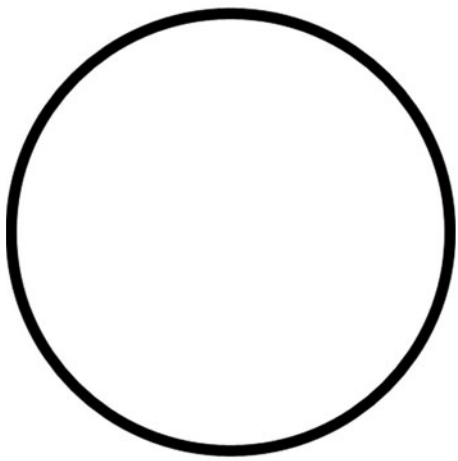

FIGURE 3.23: Sample stroke applied to an ellipse

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Ellipse
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />
</Canvas>
```

Stroke

A stroke transforms geometry to a widened form that describes the shape outline instead of the shape fill. Silverlight fills the widened geometry with exactly the same rendering rules as the main shape fill. For example, Figure 3.24 shows an example of a widened ellipse.

The widening process expands the original geometry by half the stroke thickness to form an outer outline. The widening process also shrinks the original geometry by half the stroke thickness to form an inner outline. The outer and inner outlines combine to form two figures Silverlight fills to produce the resulting stroke.

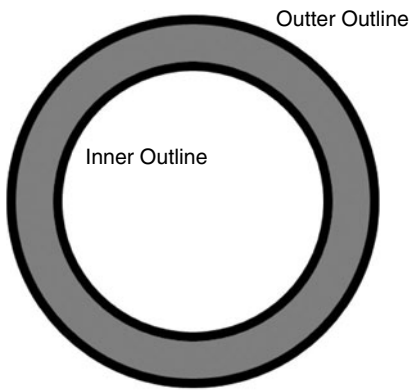

FIGURE 3.24: The widening process applied to an ellipse

Technical Insight

One side effect of the widening process is that local self-intersection can occur. For example, the process of widening a triangle generates several self-intersections as shown in Figure 3.25. One option is to run a loop removal algorithm to remove these intersections before rasterization. However, by simply filling the new geometry with the `NonZero` fill rule, these self intersections are not visible to the end user.

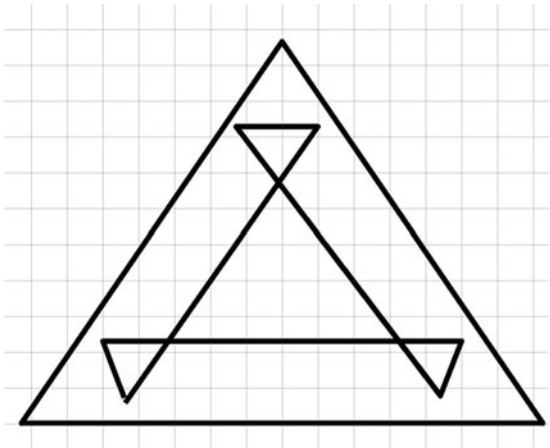

FIGURE 3.25: The widening process applied to a triangle

Dashes

To add dashes to your strokes, specify an array of distances alternating between the dash filled distance and the gap distance. For example, the simple dash array in the following XAML generates the output shown in Figure 3.26.

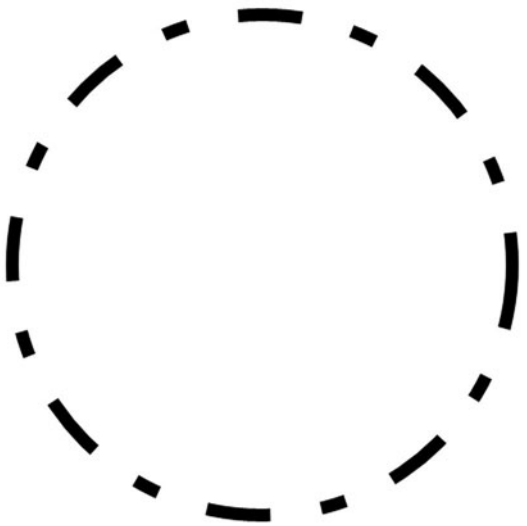

FIGURE 3.26: StrokeDashArray example of long and short dashes

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Ellipse
        Stroke="Black"
        StrokeThickness="10"
        StrokeDashArray="5, 4, 2, 4"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

Technical Insight

Dashes are also a geometry modifier built on top of the stroke geometry modifier. When you specify a `StrokeDashArray`, Silverlight takes the output of the pen and subdivides it into smaller geometries. Large numbers of dashes can result in significant slowdowns in rendering speed and therefore you should use them sparingly.

Canvas

Every example shown so far has had a single root `Canvas` element with a set of `Shape` elements contained within it. In addition to providing a convenient container, the `Canvas` element also enables you to modify the rendering primitives it contains as a group. In particular, the `Canvas` element enables the following:

- Naming groups of elements
- Grouping shapes so that you can add or remove the group with a single operation
- Applying a transform to the group of elements
- Clipping a group of elements
- Apply an opacity or opacity mask effect to a group of elements

Transforms, clipping, and opacity effects are available on both individual shapes and the `Canvas` element.

PERFORMANCE TIP

For individual shapes, it is faster to express clipping or opacity as a different geometry or a different brush color. For example, draw a `Path` with an `ImageBrush` to achieve the same result as applying a clip to an `Image` element. Similarly, you can add opacity to the brush color alpha channel instead of adding `Opacity` to the shape.

Transforms

A transform enables you to position, rotate, scale, or skew a shape or group of shapes. To transform a group of primitives, you can set the `RenderTransform` on the `Canvas` element as exemplified in the following listing to achieve the result shown in Figure 3.27.

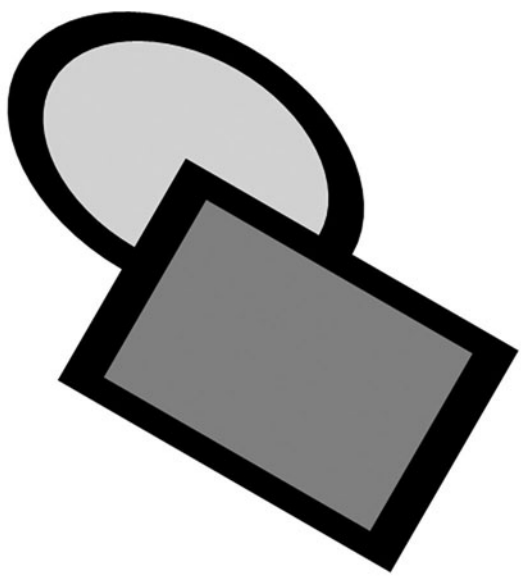

FIGURE 3.27: RenderTransform example of overlapping a rectangle over an ellipse

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1.5"/>
            <RotateTransform Angle="30"/>
            <TranslateTransform X="100" Y="-10"/>
        </TransformGroup>
    </Canvas.RenderTransform>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

As shown in the previous example, you can use a list of `ScaleTransform`, `TranslateTransform`, and `RotateTransform` elements in a `TransformGroup` element. Alternatively, you can specify an explicit matrix with a `MatrixTransform`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.RenderTransform>
        <TransformGroup>
            <MatrixTransform Matrix="
                1.30,   0.75,
               -0.50,   0.87,
              100.00, -10.00"
            />
        </TransformGroup>
    </Canvas.RenderTransform>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
```

```
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

3D Transforms (New in Silverlight 3)

In Silverlight 3, you can set the `Projection` property to a `PlaneProjection` to rotate a group of elements in 3D as shown in Figure 3.28.

FIGURE 3.28: 3D projection example

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.Projection>
        <PlaneProjection RotationY="-60" CenterOfRotationY="50" />
    </Canvas.Projection>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
        Canvas.Top="50"
    />
```

```
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

Each projection logically has its own camera. To position more than one object relative to the same perspective camera, position them all in the same place and use the `GlobalOffsetX`, `GlobalOffsetY`, and `GlobalOffsetZ` properties to move in the 3D world as shown in Figure 3.29.

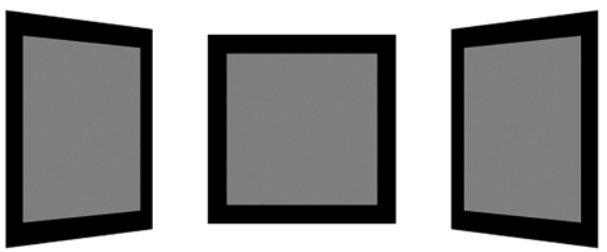

Figure 3.29: Position three rectangles in the same 3D projection camera

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

  <Rectangle
    Fill="Gray"
    Stroke="Black"
    StrokeThickness="20"
    Canvas.Left="200"
    Canvas.Top="100"
    Width="200"
    Height="200"
  >
    <Rectangle.Projection>
        <PlaneProjection
           GlobalOffsetX="-200"
           RotationY="-60"
           CenterOfRotationY="50"
        />
    </Rectangle.Projection>
  </Rectangle>
```

```
    <Rectangle
      Fill="Gray"
      Stroke="Black"
      StrokeThickness="20"
      Canvas.Left="200"
      Canvas.Top="100"
      Width="200"
      Height="200"
    >
      <Rectangle.Projection>
          <PlaneProjection  GlobalOffsetZ="-150"/>
      </Rectangle.Projection>
    </Rectangle>

    <Rectangle
      Fill="Gray"
      Stroke="Black"
      StrokeThickness="20"
      Canvas.Left="200"
      Canvas.Top="100"
      Width="200"
      Height="200"
    >
      <Rectangle.Projection>
          <PlaneProjection
             GlobalOffsetX="200"
             RotationY="60"
             CenterOfRotationY="50"
           />
      </Rectangle.Projection>
    </Rectangle>

</Canvas>
```

The global offset properties apply after the rotation property. You can also use the `LocalOffsetX`, `LocalOffsetY`, and `LocalOffsetZ` properties on the `PlaneProjection` object to apply an offset before the rotation.

Clipping

Clipping is the process of restricting the display area to a specified shape. To clip an element, set the `Clip` property as shown Figure 3.30 and in the following listing:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.Clip>
        <EllipseGeometry
            Center="100,200"
```

```
            RadiusX="150"
            RadiusY="150"
        />
    </Canvas.Clip>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />
</Canvas>
```

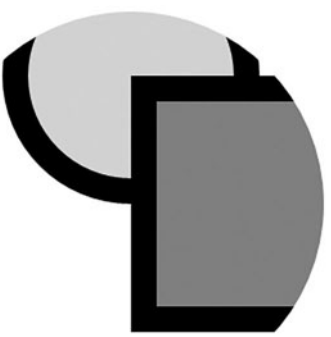

FIGURE 3.30: Clipping example

PERFORMANCE TIP

A clipping operation is semantically equivalent to intersecting two geometries. Clipping a group of elements or a single shape does come with a significant performance penalty. You should avoid clipping when possible.

Opacity

Setting opacity on a brush or setting a transparent color on a brush introduces alpha blending. In particular, if a brush contains a transparent color, the brush blends its color with the content underneath using the following formula:

$$\text{Color\_destination} = \text{alpha} * \text{Color\_source} + (1 - \text{alpha}) * \text{Color\_destination}$$

The other form of opacity is setting the `Opacity` property on a `Canvas`. This operation is not equivalent to changing the opacity of each of the shapes within the `Canvas` element as demonstrated by Figure 3.31.

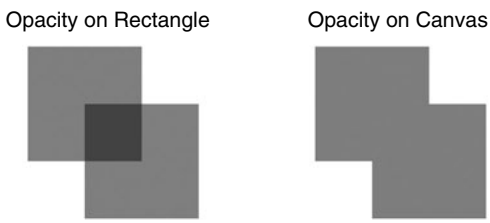

FIGURE 3.31: Canvas Opacity versus per path Opacity

PERFORMANCE TIP

Setting `Opacity` on a `Canvas` element resolves occlusion first and then blends content. This process is significantly slower at runtime than blending individual primitives. If possible, you should set opacity on a brush, brush color, or a `Path` element for maximum performance.

OpacityMask

The `OpacityMask` property on a `UIElement` provides a mechanism to blend brush per pixel alpha information with the content of a `UIElement`. For example, the following XAML would produce the result shown in Figure 3.32.

FIGURE 3.32: OpacityMask example

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.OpacityMask>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Transparent" Offset="0"/>
                <GradientStop Color="White" Offset="1"/>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Canvas.OpacityMask>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

`OpacityMask` is slow at runtime. In some cases, it is faster to draw content on top that blends to the background instead of using the `OpacityMask`. For example, you can achieve the effect in Figure 3.32 with the following XAML:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
            Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

    <!-- simulate opacity mask effect with a rectangle on top -->

    <Rectangle Width="300" Height="300">
        <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Transparent" Offset="1"/>
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>

</Canvas>
```

Under the Hood

Previous sections have discussed the graphics principles and the graphics API elements. This section goes "under the hood" to describe how Silverlight draws XAML content and displays it in the browser window.

Understanding this process will help you understand the Silverlight runtime performance characteristics. Furthermore, you will understand the problems solved by the runtime and the problems your application must solve.

In particular, this section discusses the following:

- The draw loop process that takes changes to the graph of objects and draws it to an off screen back buffer
- The rasterization process that converts vector graphics primitives to pixels in an offscreen back buffer
- Performance optimizations such as incremental redraw, occlusion culling, and multi-core
- How the offscreen back buffer gets displayed in the browser window

Draw Loop

Silverlight draws at a regular timer interval set by the `MaxFrameRate` property. On each tick of the timer, Silverlight does the following:

1. Checks for any changes to the properties of our graph of `Canvas` and `Shape` elements. If no changes exist, Silverlight does no further work for this timer tick.
2. Performs any pending layout operations. Chapter 7, "Layout," will discuss these layout operations further.
3. Gathers rendering changes and prepares to rasterize them.
4. Incrementally rasterizes the changes for the current timer tick. The graphics state at the current timer tick is the current frame.
5. Notifies the browser that a frame (or an incremental delta to an existing frame) is complete for display.

Rasterization

After the draw loop has identified which elements need to be redrawn, Silverlight converts those elements to a set of pixels in our offscreen

PERFORMANCE TIP

One property of the draw loop is that nothing draws immediately after you make a change to the element tree. Consequently, profiling tools do not associate the cost of a drawing operation with the function that added those drawing primitives. To tune your performance, you should measure the maximum frame rate of your application during development. In particular, set the `MaxFrameRate` property to some value that is beyond what Silverlight can achieve and turn on the frame rate display as shown in the following JavaScript:

```
function loadHandler()
{
   plugin = document.getElementById("MySilverlightPlugin");
   plugin.settings.EnableFramerateCounter = true;
   plugin.settings.MaxFrameRate = 10000;
}
```

During development, watch for content that drops the frame rate significantly, and consider specifying that content in an alternative form.

back buffer. The previous discussion of shapes described how to specify path outlines and a method of specifying the inside and the outside of the shape. However, the geometry describes an abstract infinite resolution outline of a shape and a screen has a finite number of pixels to color. Rasterization is the process of converting from a path outline to discrete pixels. This section describes how rasterization is accomplished.

The simplest method to convert geometry to pixels is a process called *sampling*. The sampling process uses a discrete number of sample points to convert from the infinite shape description to pixels. For example, consider the simple sample pattern consisting of a uniform grid of sample points with one sample point per pixel. If the sample point is contained within the geometry, light up the pixel. If the sample point is not contained within the geometry, do not light the pixel. For example, the circle specified by the following XAML would light the pixels shown in Figure 3.33.

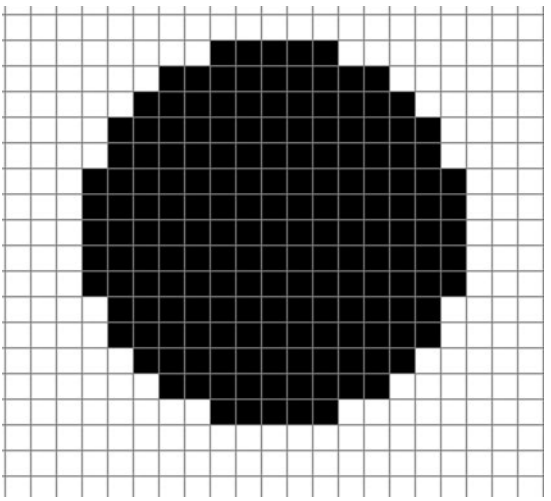

FIGURE 3.33: Sampling a circle

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="Black"
        Width="15"
        Height="15"
    />

</Canvas>
```

You may have noticed that the integer coordinates were located at the top left of the pixel and the sample points were in the center of a pixel. This convention enables a symmetrical curved surface specified on integer coordinates to produce a symmetrical rasterization. If the sample points were on integer coordinates instead, the ellipse would lose symmetry as shown in Figure 3.34.

The rasterization shown in Figure 3.33 appears to have jagged edges. This jagged appearance is the consequence of aliasing. Aliasing is the loss of information that results from converting from a continuous curve to a discrete set of samples. Anti-aliasing is a term that refers to a technique that attempts to minimize aliasing artifacts.

The Silverlight anti-aliasing technique consists of sampling multiple times per pixel and applying a box filter to produce the final pixel color. Silverlight conceptually samples 64 times per pixel as shown in Figure 3.35.

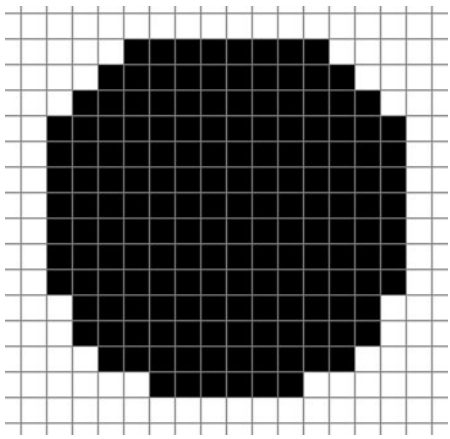

FIGURE 3.34: Sampling a circle with integer sample point coordinates

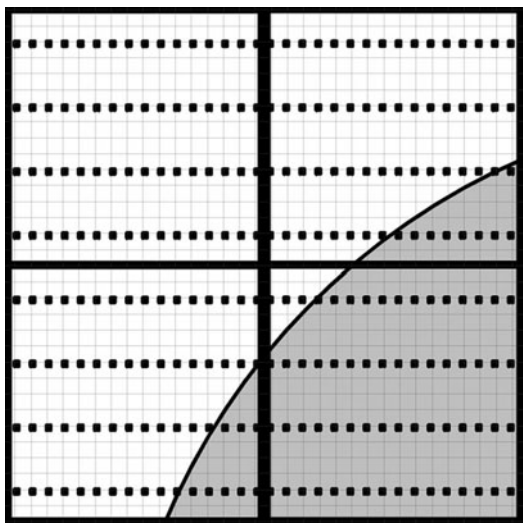

FIGURE 3.35: Anti-aliasing sampling pattern

The box filter averages the contribution of all samples within a rectangle bordering the pixel to produce a final pixel color. If some partial number of samples is in the box, Silverlight applies transparency to blend smoothly with what is underneath the geometry as shown in Figure 3.36. This anti-aliasing technique produces a smooth transition from inside the shape to outside the shape along edges.

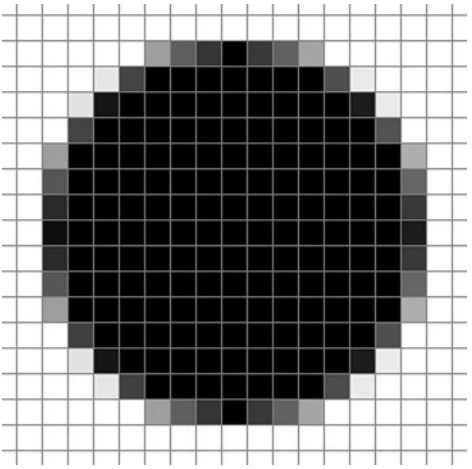

Figure 3.36: Anti-aliased rasterization

Technical Insight

You may be wondering why there are 16 samples per pixel in the x direction and only 4 samples per pixel in the y direction. The reason for picking this sample pattern is that horizontal resolution is critical to being able to render text clearly. Furthermore, horizontal resolution is computationally cheap and vertical resolution is slower. The 16x4 sampling pattern balances image quality and speed.

Instead of a box pattern, it is also possible to accumulate samples in a circular pattern, weight samples unevenly, or even have a sample pattern that extends far beyond a single pixel in size. In fact, all of these other algorithms generate better image quality than a box filter but typically render more slowly. The Silverlight high-resolution box filter is a choice made to achieve good rendering performance with reasonable image quality.

One artifact of anti-aliasing is a visible seam that sometimes results from drawing two adjacent shapes. For example, the following two rectangles that meet in the middle of a pixel generate a seam:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Rectangle
        Fill="Black"
        Width="100.5"
        Height="100.5"
    />
    <Rectangle
        Fill="Black"
        Canvas.Left="100.5"
        Width="100.5"
        Height="100.5"
    />
</Canvas>
```

The previous XAML results in the rasterization shown in Figure 3.37. Notice the gap between the two rectangles. The rectangles joined perfectly in the input XAML, so why is there a seam in the rendered result?

These seams are a result of the rasterization rules described in this section. Consider the rasterization process applied to a light gray pixel *X* shown in Figure 3.37. Rectangle *A* is covering exactly half the samples for pixel *X*. Silverlight consequently draws that pixel of Rectangle *A* with 0.5 anti-aliasing alpha. Alpha is a term that refers to the transparency used to blend colors with a formula such as

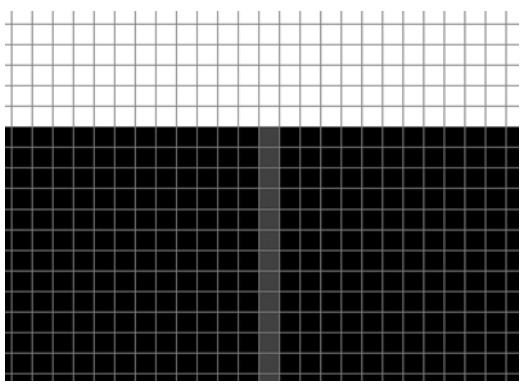

FIGURE 3.37: Anti-aliasing seam example

Color_destination = alpha * Color_source + (1 – alpha) * Color_destination

In our example, *alpha = 0.5, Color_source = Black*, and *Color_destination = White*. Blending along the edge of rectangle *A* results in a destination color of

0.5 * Black + (1 – 0.5) * White = 0.5 * White

Rectangle *B* also covers half its sample points. Silverlight also blends pixel *X* of rectangle *B* with *alpha = 0.5* to a background color of *0.5 * White*. Consequently, the resulting color is

0.5 * Black + (1 – 0.5) * (0.5 White) = 0.25 White.

The final pixel color has one quarter of the background color showing through as a visible seam.

Technical Insight

This result is an artifact of sampling each primitive independently. An alternative anti-aliasing mode is full screen anti-aliasing that processes all samples from all shapes simultaneously. However, Silverlight does not currently use full screen anti-aliasing because it results in slower runtime performance.

To avoid these seams, you should snap edges to pixel boundaries as shown in Figure 3.38. Snapping also produces a sharper edge between the

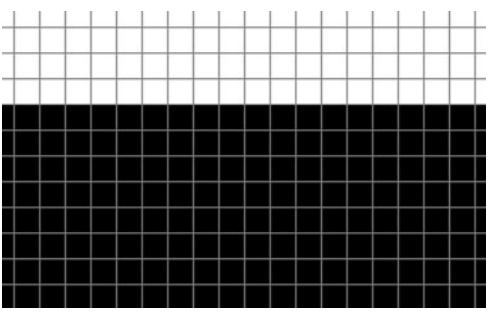

FIGURE 3.38: Pixel snapped rasterization

two shapes. However, pixel snapping only removes seams if you align the shapes edges with the *x*-axis or the *y*-axis. For the rotated edges shown in Figure 3.39, snapping does not remove the artifact. For rotated edges, the common technique to avoid this seam is to overlap the edges so that the background is no longer visible.

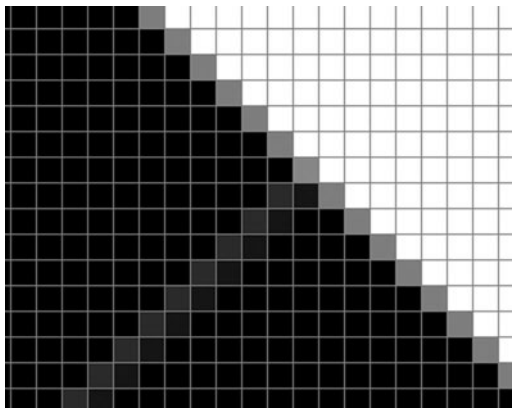

FIGURE 3.39: Seams with a rotated edge

Bilinear Filtering

The previous section discussed how Silverlight converts an arbitrary geometry to a set of pixels to fill. Silverlight then colors the filled pixels based on the brush specified. This process is straightforward for solid color brushes and gradient brushes. However, with image brushes, Silverlight must map from the destination pixels to the original image data, which may be at a different resolution. This section describes the mapping function used to achieve the image data stretch shown in Figure 3.40.

Nearest neighbor is a simple image scaling function that transforms the destination pixel to an image bitmap position and picks the nearest pixel color. Nearest neighbor sampling generates ugly aliasing artifacts when the image is displayed with a scale or rotation as shown in Figure 3.41. You will notice jagged lines if you look at the picture closely.

Silverlight uses nearest neighbor sampling in the special case where the brush image data maps exactly onto centers of pixels. For rotated, scaled, or non-integer translated images, bilinear filtering is used to produce the result shown in Figure 3.40.

FIGURE 3.40: Image with bilinear filtering

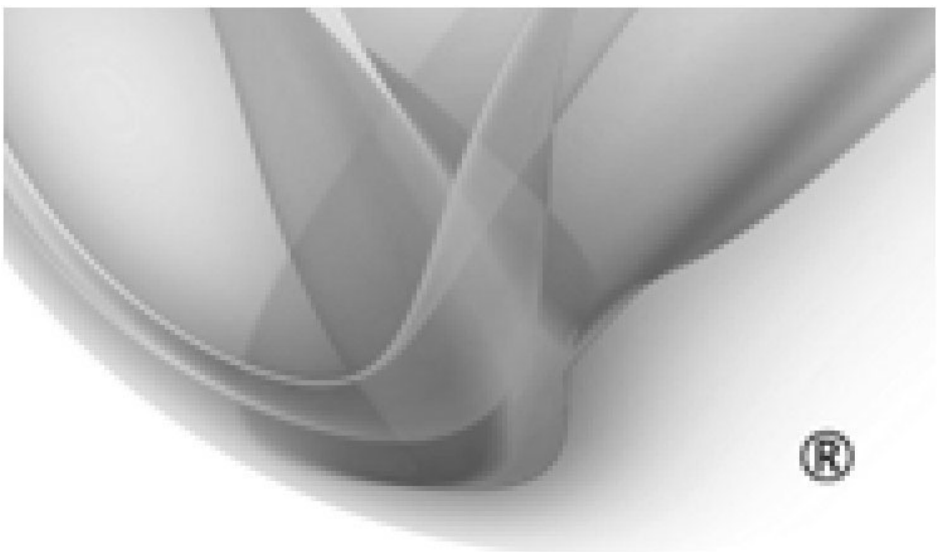

FIGURE 3.41: Image with nearest neighbor

Bilinear filtering maps the screen position to a position (u,v) in image space. The bilinear filtering process then interpolates a color from pixels *(floor(u), floor(v))*, *(floor(u) + 1, floor(v))*, *(floor(u), floor(v) + 1)*, and *(floor(u) + 1, floor(v)+1)*. Figure 3.42 illustrates this process. Bilinear filtering generates good results for scales that are within a factor of two of the original image size. Figure 3.43 demonstrates the results of scaling an image in two sizes within reasonable limits.

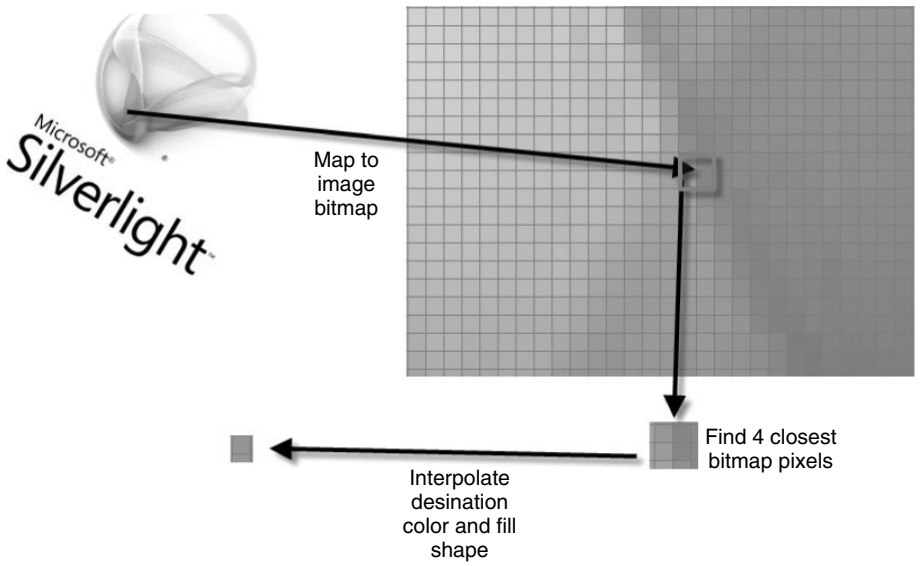

FIGURE 3.42: The bilinear filtering process

FIGURE 3.43: Image scaling within good limits

With bilinear filtering, if you scale up an image significantly, it becomes blurry. Conversely, if you scale down an image significantly, it looks aliased. Figure 3.44 shows examples of both these artifacts.

Figure 3.44: Image scaling extremes

New in Silverlight 3

There are a number of techniques for scaling down an image to produce a result better than Figure 3.44. However, these techniques can slow down your animations. Silverlight 3 adds support for mip-mapping that converts your image to a set of smaller images at various sizes using a better algorithm.

For example, if you have a 128x128 image, Silverlight also generates copies at 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1 resolutions resized with high quality. When displaying the image at a particular scale, Silverlight chooses the closest resolution to the display size or even uses multiple sizes at once when displaying in 3D.

This conversion happens only once per image and Silverlight caches the resulting images. Consequently, this approach uses 33% more memory to store images to achieve better image quality.

Incremental Redraw

In addition to drawing static objects for a single frame, Silverlight must constantly redraw objects as they are changing. If an object moves from one

position to another, it would be wasteful to redraw all the pixels on the screen. Instead, Silverlight marks the old position as needing a redraw and marks the new position as also needing a redraw. To simplify this marking algorithm, Silverlight uses the bounding box of a shape instead of the tight shape bounds.

For example, suppose the shape shown in the following XAML moves from position 0,0 to position 100,100. Figure 3.45 shows the area that is redrawn.

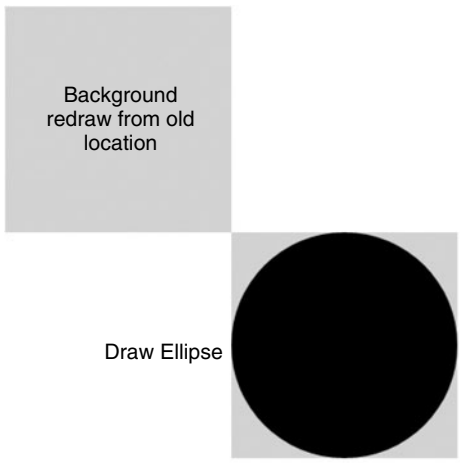

FIGURE 3.45: Incremental redraw regions

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Ellipse
        Fill="Black"
        Width="100"
        Height="100”
    />
</Canvas>
```

To view a visualization of these incremental redraw regions in an application, use the following JavaScript:

```
function loadHandler()
{
    plugin = document.getElementById("MySilverlightPlugin");
    plugin.settings.EnableRedrawRegions = true;
}
```

This visualization blends a transparent color on top of any content drawn and cycles to a different color each frame. Consequently, any content that is flashing represents content that Silverlight is constantly redrawing. Any content that stabilizes on a single color has not changed for several frames.

Occlusion Culling

The most expensive operation in the draw loop is the rasterization process, which writes each of the destination pixels. For example, a full screen animation can consist of processing several hundred million pixels per second. Each of these pixels applies at least one brush operation. If there are overlapping brushes, the computational requirements can multiply by a factor of 3 to 10.

As the graph of elements gets more complicated, it may no longer render at the desired frame rate. To optimize the rasterization process, Silverlight avoids brush operations for completely occluded brush pixels. For example, if you draw a full screen background and an almost full screen image, Silverlight computes all the image pixels and only those background pixels not covered by the image itself. For complicated graphs of elements, this optimization can produce a 3–10x speedup.

> **PERFORMANCE TIP**
>
> Occlusion culling only applies to brush pixel color optimizations. If a complicated geometry is behind a big opaque rectangle, the rasterizer walks the geometry before it realizes that the pixel operations are not necessary. Consequently, it is still important to remove hidden content from the element tree for maximum performance.

Multi-Core Rendering

Silverlight takes advantage of multiple CPU cores to produce faster rendering throughput. In particular, Silverlight subdivides a frame into a set of horizontal bands and distributes the rasterization of those bands across CPU cores as shown in Figure 3.46. Currently, only the frame rasterization step, effects, mip-map generation, and media operations run in parallel

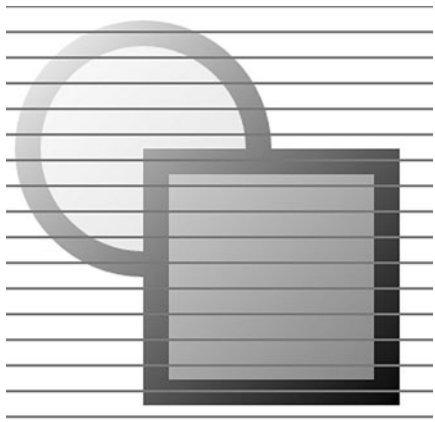

FIGURE 3.46: Dividing a scene for multi-core rendering

across CPU cores. Systems such as layout, control templating, application user code, and animation all run on a single thread. Consequently, you can determine if your application is rasterization bound by simply setting your framerate to 10000 frames per second and measuring your CPU usage percentage. If you achieve almost 100% CPU usage on a dual core machine, you are almost entirely rasterization bound. If you achieve 70% CPU usage on a dual core machine (at 10000 frames per second), that means 30% of the work is not running in parallel.

How Content Gets to the Screen

As previously discussed, the draw loop first draws a frame to an offscreen back buffer and then it notifies the browser that the frame is ready for display. With `windowless=false` mode, Silverlight content goes to the screen without browser intervention on most operating systems. With `windowless=true`, the browser copies the offscreen frame to its display area. This extra step is both slow and can result in visual tearing effects in a number of browsers. The worst mode of operation is when `windowless=true` is specified with a transparent color for the background of the control. The transparent color causes the Web browser to redraw the content underneath the control each time any control content has changed.

PERFORMANCE TIP

You should avoid using both a transparent background and `windowless=true` if possible.

Where Are We?

This chapter described the following:

- The graphics system design principles
- The elements for displaying graphics
- The problems the Silverlight runtime solves "under the hood" and the problems your application must solve

In addition, you have learned a number of important performance optimization techniques for use with your application.

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z