About the Author

Matthew MacDonald is an author, educator, and Microsoft MVP. He's the author of more than a dozen books about .NET programming, including Pro Silverlight 3 in C# (Apress, 2009), Pro ASP.NET 3.5 in C# (Apress, 2007), and the previous edition of this book, Pro WPF in C# 2008 (Apress, 2008). He lives in Toronto with his wife and two daughters.

About the Technical Reviewer

■ **Fabio Claudio Ferracchiati** is a prolific writer on cutting-edge technologies. Fabio has contributed to more than a dozen books on .NET, C#, Visual Basic, and ASP.NET. He is a .NET Microsoft Certified Solution Developer (MCSD) and lives in Rome, Italy. You can read his blog at `www.ferracchiati.com`.

Acknowledgments

No author can complete a book without a small army of helpful individuals. I'm deeply indebted to the whole Apress team, including Anne Collett, who shepherded this third edition through production, Kim Wimpsett and Marilyn Smith, who speedily performed the copy edit, and many other individuals who worked behind the scenes indexing pages, drawing figures, and proofreading the final copy. I also owe a special thanks to Gary Cornell, who always offers invaluable advice about projects and the publishing world.

Fabio Claudio Ferracchiati and Christophe Nasarre deserve my sincere thanks for their insightful and timely tech review comments. I'm also thankful for the legions of die-hard bloggers on the various WPF teams, who never fail to shed light on the deepest recesses of WPF. I encourage anyone who wants to learn more about the future of WPF to track them down. Finally, I'd never write any book without the support of my wife and these special individuals: Nora, Razia, Paul, and Hamid. Thanks everyone!

Introduction

When .NET first appeared, it introduced a small avalanche of new technologies. There was a whole new way to write web applications (ASP.NET), a whole new way to connect to databases (ADO.NET), new typesafe languages (C# and VB .NET), and a managed runtime (the CLR). Not least among these new technologies was Windows Forms, a library of classes for building Windows applications.

Although Windows Forms is a mature and full-featured toolkit, it's hardwired to essential bits of Windows plumbing that haven't changed much in the past ten years. Most significantly, Windows Forms relies on the Windows API to create the visual appearance of standard user interface elements such as buttons, text boxes, check boxes, and so on. As a result, these ingredients are essentially uncustomizable.

For example, if you want to create a stylish glow button you need to create a custom control and paint every aspect of the button (in all its different states) using a lower-level drawing model. Even worse, ordinary windows are carved up into distinct regions, with each control getting its own piece of real estate. As a result, there's no good way for the painting in one control (for example, the glow effect behind a button) to spread into the area owned by another control. And don't even think about introducing animated effects such as spinning text, shimmering buttons, shrinking windows, or live previews because you'll have to paint every detail by hand.

The Windows Presentation Foundation (WPF) changes all this by introducing a model with entirely different plumbing. Although WPF includes the standard controls you're familiar with, it draws every text, border, and background fill *itself*. As a result, WPF can provide much more powerful features that let you alter the way any piece of screen content is rendered. Using these features, you can restyle common controls such as buttons, often without writing any code. Similarly, you can use transformation objects to rotate, stretch, scale, and skew anything in your user interface, and you can even use WPF's baked-in animation system to do it right before the user's eyes. And because the WPF engine renders the content for a window as part of a single operation, it can handle unlimited layers of overlapping controls, even if these controls are irregularly shaped and partially transparent.

Underlying WPF is a powerful infrastructure based on DirectX, the hardware-accelerated graphics API that's commonly used in cutting-edge computer games. This means that you can use rich graphical effects without incurring the performance overhead that you'd suffer with Windows Forms. In fact, you even get advanced features such as support for video files and 3-D content. Using these features (and a good design tool), it's possible to create eye-popping user interfaces and visual effects that would have been all but impossible with Windows Forms.

Although the cutting-edge video, animation, and 3-D features often get the most attention in WPF, it's important to note that you can use WPF to build an ordinary Windows application with standard controls and a straightforward visual appearance. In fact, it's just as easy to use common controls in WPF as it is in Windows Forms. Even better, WPF enhances features that appeal directly to business developers, including a vastly improved data binding model, a set of classes for printing content and managing print queues, and a document feature for displaying large amounts of formatted text. You'll even get a model for building page-based applications that run seamlessly in Internet Explorer and can be launched from a website, all without the usual security warnings and irritating installation prompts.

Overall, WPF combines the best of the old world of Windows development with new innovations for building modern, graphically rich user interfaces. Although Windows Forms applications will continue to live on for years, developers embarking on new Windows development projects should look first to WPF.

About This Book

This book is an in-depth exploration of WPF for professional developers who know the .NET platform, the C# language, and the Visual Studio development environment. Experience with previous versions of WPF is not required, although new features are highlighted with a "What's New" box at the beginning of each chapter for more seasoned WPF developers.

This book provides a complete description of every major WPF feature, from XAML (the markup language used to define WPF user interfaces) to 3-D drawing and animation. Along the way, you'll occasionally work with code that involves other features of the .NET Framework, such as the ADO.NET classes you use to query a database. These features aren't discussed here. Instead, if you want more information about .NET features that aren't specific to WPF, you can refer to one of the many dedicated .NET titles from Apress.

Chapter Overview

This book includes 33 chapters. If you're just starting out with WPF, you'll find it's easiest to read them in order, as later chapters often draw on the techniques demonstrated in earlier chapters.

The following list gives you a quick preview of each chapter:

Chapter 1: Introducing WPF describes the architecture of WPF, its DirectX plumbing, and the new device-independent measurement system that resizes user interfaces automatically.

Chapter 2: XAML describes the XAML standard that you use to define user interfaces. You'll learn why it was created and how it works, and you'll create a basic WPF window using different coding approaches.

Chapter 3: Layout delves into the layout panels that allow you to organize elements in a WPF window. You'll consider different layout strategies, and you'll build some common types of windows.

Chapter 4: Dependency Properties describes how WPF uses dependency properties to provide support for key features such as data binding and animation.

Chapter 5: Routed Events describes how WPF uses event routing to send events bubbling or tunneling through the elements in your user interface. It also describes the basic set of mouse, keyboard, and multitouch events that all WPF elements support.

Chapter 6: Controls considers the controls every Windows developer is familiar with, such as buttons, text boxes, and labels—and their WPF twists.

Chapter 7: The Application introduces the WPF application model. You'll see how to create single-instance and document-based WPF applications.

Chapter 8: Element Binding introduces WPF data binding. You'll see how to bind any type of object to your user interface.

Chapter 9: Commands introduces the WPF command model, which allows you to wire multiple controls to the same logical action.

Chapter 10: Resources describes how resources let you embed binary files in your assembly and reuse important objects throughout your user interface.

Chapter 11: Styles and Behaviors explains the WPF style system, which lets you apply a set of common property values to an entire group of controls.

Chapter 12: Shapes, Brushes, and Transforms introduces the 2-D drawing model in WPF. You'll learn to create shapes, alter elements with transforms, and paint exotic effects with gradients, tiles, and images.

Chapter 13: Geometries and Drawings delves deeper into 2-D drawing. You'll learn to create complex paths that incorporate arcs and curves and how to use complex graphics efficiently.

Chapter 14: Effects and Visuals describes lower-level graphics programming. You'll apply Photoshop-style effects with pixel shaders, build a bitmap by hand, and use WPF's visual layer for optimized drawing.

Chapter 15: Animation Basics explores WPF's animation framework, which lets you integrate dynamic effects into your application using straightforward, declarative markup.

Chapter 16: Advanced Animations explore more sophisticated animation techniques like key-frame animation, path-based animation, and frame-based animation. You'll also consider a detailed example that shows how to create and manage dynamic animations with code.

Chapter 17: Control Templates shows you how you can give any WPF control a dramatic new look (and new behavior) by plugging in a customized template. You'll also see how templates allow you to build a skinnable application.

Chapter 18: Custom Elements explores how you can extend the existing WPF controls and create your own. You'll see several examples, including a template-based color picker, a flippable panel, a custom layout container, and a decorator that performs custom drawing.

Chapter 19: Data Binding shows you how to fetch information from a database, insert it into a custom data objects, and bind these objects to WPF controls. You'll also learn how to improve the performance of huge data-bound lists with virtualization, and catch editing mistakes with validation.

Chapter 20: Formatting Bound Data shows some of the tricks for turning raw data into rich data displays that incorporate pictures, controls, and selection effects.

Chapter 21: Data Views explores how you use the view in a data-bound window to navigate through a list of data items, and to apply filtering, sorting, and grouping.

Chapter 22: Lists, Grids, and Trees gives you a tour of WPF's rich data controls, including the ListView, TreeView, and DataGrid.

Chapter 23: Windows examines how windows work in WPF. You'll also learn how to create irregularly shaped windows and use Vista glass effects. You'll also make the most of Windows 7 features by customizing taskbar jump lists, thumbnails, and icon overlays.

Chapter 24: Pages and Navigation describes how you can build pages in WPF and keep track of navigation history. You'll also see how to build a browser-hosted WPF application that can be launched from a website.

Chapter 25: Menus, Toolbars, and Ribbons considers command-oriented controls such as menus and toolbars. You'll also get a taste of more modern user interface with the freely downloadable Ribbon control.

Chapter 26: Sound and Video describes WPF's media support. You'll see how to control playback for sound and video, and how to throw in synchronized animations and live effects.

Chapter 27: 3-D Drawing explores the support for drawing 3-D shapes in WPF. You'll learn how to create, transform, and animate 3-D objects. You'll even see how to place interactive 2-D controls on 3-D surfaces.

Chapter 28: Documents introduces WPF's rich document support. You'll learn to use flow documents to present large amounts of text in the most readable way possible, and you'll use fixed documents to show print-ready pages. You'll even use the RichTextBox to provide document editing.

Chapter 29: Printing demonstrates WPF's printing model, which lets you draw text and shapes in a print document. You'll also learn how to manage page settings and print queues.

Chapter 30: Interacting with Windows Forms examines how you can combine WPF and Windows Forms content in the same application—and even in the same window.

Chapter 31: Multithreading describes how to create responsive WPF applications that perform time-consuming work in the background.

Chapter 32: The Add-In Model shows you how to create an extensible application that can dynamically discover and load separate components.

Chapter 33: ClickOnce Deployment shows how you can deploy WPF applications using the ClickOnce setup model.

What You Need to Use This Book

In order to *run* a WPF 4 application, your computer must have Windows 7, Windows Vista or Windows XP with Service Pack 2. You also need the .NET Framework 4. In order to *create* a WPF 4 application (and open the sample projects included with this book), you need Visual Studio 2010, which includes the .NET Framework 4.

There's one other option. Instead of using any version of Visual Studio, you can use Expression Blend—a graphically oriented design tool—to build and test WPF applications. Overall, Expression Blend is intended for graphic designers who spend their time creating serious eye candy, while Visual Studio is ideal for code-heavy application programmers. This book assumes you're using Visual Studio. If you'd like to learn more about Expression Blend, you can consult one of many dedicated books on the subject. (Incidentally, to create applications with WPF 4 you need Expression Blend 4, which is in beta at the time of this writing.)

Code Samples and URLs

It's a good idea to check the Apress website or `www.prosetech.com` to download the most recent up-to-date code samples. You'll need to do this to test most of the more sophisticated code examples described in this book because the less significant details are usually left out. This book focuses on the most important sections so that you don't need to wade through needless extra pages to understand a concept.

To download the source code, surf to `www.prosetech.com` and look for the page for this book. You'll also find a list of links that are mentioned in this book, so you can find important tools and examples without needless typing.

Feedback

This book has the ambitious goal of being the best tutorial and reference for programming WPF. Toward that end, your comments and suggestions are extremely helpful. You can send complaints, adulation, and everything in between directly to `apress@prosetech.com`. I can't solve your .NET problems or critique your code, but I will benefit from information about what this book did right and wrong (or what it may have done in an utterly confusing way).

CHAPTER 1

Introducing WPF

The Windows Presentation Foundation (WPF) is a graphical display system for Windows. WPF is designed for .NET, influenced by modern display technologies such as HTML and Flash, and hardware-accelerated. It's also the most radical change to hit Windows user interfaces since Windows 95.

In this chapter, you'll take your first look at the architecture of WPF. You'll learn how it deals with varying screen resolutions, and you'll get a high-level survey of its core assemblies and classes. You'll also consider the new features that have been added to WPF 4.

What's New If you're already an experienced WPF developer, you'll want to skip directly to the "WPF 4" section later in this chapter, which summarizes the changes in the latest release of WPF.

The Evolution of Windows Graphics

It's hard to appreciate how dramatic WPF is without realizing that Windows developers have been using essentially the same display technology for more than 15 years. A standard Windows application relies on two well-worn parts of the Windows operating system to create its user interface:

- **User32.** This provides the familiar Windows look and feel for elements such as windows, buttons, text boxes, and so on.
- **GDI/GDI+.** This provides drawing support for rendering shapes, text, and images at the cost of additional complexity (and often lackluster performance).

Over the years, both technologies have been refined, and the APIs that developers use to interact with them have changed dramatically. But whether you're crafting an application with .NET and Windows Forms or lingering in the past with Visual Basic 6 or MFC-based C++ code, behind the scenes the same parts of the Windows operating system are at work. Newer frameworks simply deliver better wrappers for interacting with User32 and GDI/GDI+. They can provide improvements in efficiency, reduce complexity, and add prebaked features so you don't have to code them yourself; but they can't remove the fundamental limitations of a system component that was designed more than a decade ago.

■ **Note** The basic division of labor between User32 and GDI/GDI+ was introduced more than 15 years ago and was well established in Windows 3.0. Of course, User32 was simply User at that point, because software hadn't yet entered the 32-bit world.

DirectX: The New Graphics Engine

Microsoft created one way around the limitations of the User32 and GDI/GDI+ libraries: *DirectX*. DirectX began as a cobbled-together, error-prone toolkit for creating games on the Windows platform. Its design mandate was speed, and so Microsoft worked closely with video card vendors to give DirectX the hardware acceleration needed for complex textures, special effects such as partial transparency, and three-dimensional graphics.

Over the years since it was first introduced (shortly after Windows 95), DirectX has matured. It's now an integral part of Windows, with support for all modern video cards. However, the programming API for DirectX still reflects its roots as a game developer's toolkit. Because of its raw complexity, DirectX is almost never used in traditional types of Windows applications (such as business software).

WPF changes all this. In WPF, the underlying graphics technology isn't GDI/GDI+. Instead, it's DirectX. Remarkably, WPF applications use DirectX no matter what type of user interface you create. That means that whether you're designing complex three-dimensional graphics (DirectX's forte) or just drawing buttons and plain text, all the drawing work travels through the DirectX pipeline. As a result, even the most mundane business applications can use rich effects such as transparency and anti-aliasing. You also benefit from hardware acceleration, which simply means DirectX hands off as much work as possible to the graphics processing unit (GPU), which is the dedicated processor on the video card.

■ **Note** DirectX is more efficient because it understands higher-level ingredients such as textures and gradients that can be rendered directly by the video card. GDI/GDI+ doesn't, so it needs to convert them to pixel-by-pixel instructions, which are rendered much more slowly by modern video cards.

One component that's still in the picture (to a limited extent) is User32. That's because WPF still relies on User32 for certain services, such as handling and routing input and sorting out which application owns which portion of screen real estate. However, all the drawing is funneled through DirectX.

■ **Note** This is the most significant change in WPF. WPF is *not* a wrapper for GDI/GDI+. Instead, it's a replacement—a separate layer that works through DirectX.

Hardware Acceleration and WPF

You're probably aware that video cards differ in their support for specialized rendering features and optimizations. Fortunately, this isn't a problem, because WPF has the ability to perform everything it does using software calculations rather than relying on built-in support from the video card.

■ **Note** There's one exception to WPF's software support. Because of poor driver support, WPF performs anti-aliasing for 3-D drawings only if you're running your application on Windows Vista or Windows 7 (and you have a native WDDM driver for your video card). That means that if you draw three-dimensional shapes on a Windows XP computer, you'll end up with slightly jagged edges rather than nicely smoothed lines. However, anti-aliasing is always provided for 2-D drawings, regardless of the operating system and driver support.

Having a high-powered video card is not an absolute guarantee that you'll get fast, hardware-accelerated performance in WPF. Software also plays a significant role. For example, WPF can't provide hardware acceleration to video cards that are using out-of-date drivers. (If you're using an older video card, these out-of-date drivers are quite possibly the only ones that were provided in the retail package.) WPF also provides better performance under the Windows Vista and Windows 7 operating systems, where it can take advantage of the Windows Display Driver Model (WDDM). WDDM offers several important enhancements beyond the Windows XP Display Driver Model (XPDM). Most importantly, WDDM allows several GPU operations to be scheduled at once, and it allows video card memory to be paged to normal system memory if you exceed what's available on the video card.

As a general rule of thumb, WPF offers some sort of hardware acceleration to all WDDM drivers and to XPDM drivers that were created after November 2004, which is when Microsoft released new driver development guidelines. Of course, the level of support differs. When the WPF infrastructure first starts up, it evaluates your video card and assigns it a rating from 0 to 2, as described in the sidebar "WPF Tiers."

Part of the promise of WPF is that you don't need to worry about the details and idiosyncrasies of specific hardware. WPF is intelligent enough to use hardware optimizations where possible, but it has a software fallback for everything. So if you run a WPF application on a computer with a legacy video card, the interface will still appear the way you designed it. Of course, the software alternative may be much slower, so you'll find that computers with older video cards won't run rich WPF applications very well, especially ones that incorporate complex animations or other intense graphical effects. In practice, you might choose to scale down complex effects in the user interface, depending on the level of hardware acceleration that's available in the client (as indicated by the RenderCapability.Tier property).

■ **Note** The goal of WPF is to off-load as much of the work as possible on the video card so that complex graphics routines are *render-bound* (limited by the GPU) rather than *processor-bound* (limited by your computer's CPU). That way, you keep the CPU free for other work, you make the best use of your video card, and you are able to take advantage of performance increases in newer video cards as they become available.

WPF Tiers

Video cards differ significantly. When WPF assesses a video card, it considers a number of factors, including the amount of RAM on the video card, support for pixel shaders (built-in routines that calculate per-pixel effects such as transparency), and support for vertex shaders (built-in routines that calculate values at the vertexes of a triangle, such as the shading of a 3-D object). Based on these details, it assigns a *rendering tier* value.

WPF recognizes three rendering tiers:

- **Rendering Tier 0.** The video card will not provide any hardware acceleration. This corresponds to a DirectX version level of less than 7.0.
- **Rendering Tier 1.** The video card can provide partial hardware acceleration. This corresponds to a DirectX version level greater than 7.0 but less than 9.0.
- **Rendering Tier 2.** All features that can be hardware accelerated will be. This corresponds to a DirectX version level greater than or equal to 9.0.

In some situations, you might want to examine the current rendering tier programmatically so you can selectively disable graphics-intensive features on lesser-powered cards. To do so, you need to use the static Tier property of the System.Windows.Media.RenderCapability class. But there's one trick. To extract the tier value from the Tier property, you need to shift it 16 bits, as shown here:

```
int renderingTier = (RenderCapability.Tier >> 16);

if (renderingTier == 0)
{ ... }
else if (renderingTier == 1)
{ ... }
```

This design allows extensibility. In future versions of WPF, the other bits in the Tier property might be used to store information about support for other features, thereby creating subtiers.

For more information about what WPF features are hardware-accelerated for tier 1 and tier 2 and for a list of common tier 1 and tier 2 video cards, refer to `http://msdn.microsoft.com/en-us/library/ms742196(VS.100).aspx`.

WPF: A Higher-Level API

If the only thing WPF offered was hardware acceleration through DirectX, it would be a compelling improvement but not a revolutionary one. But WPF actually includes a basket of high-level services designed for application programmers.

The following are some of the most dramatic changes that WPF ushers into the Windows programming world:

- **A web-like layout model.** Rather than fix controls in place with specific coordinates, WPF emphasizes flexible flow layout that arranges controls based on their content. The result is a user interface that can adapt to show highly dynamic content or different languages.
- **A rich drawing model.** Rather than painting pixels, in WPF you deal with *primitives*—basic shapes, blocks of text, and other graphical ingredients. You also have new features, such as true transparent controls, the ability to stack multiple layers with different opacities, and native 3-D support.
- **A rich text model.** After years of substandard text handling, WPF finally gives Windows applications the ability to display rich, styled text anywhere in a user interface. You can even combine text with lists, floating figures, and other user interface elements. And if you need to display large amounts of text, you can use advanced document display features such as wrapping, columns, and justification to improve readability.
- **Animation as a first-class programming concept.** In WPF, there's no need to use a timer to force a form to repaint itself. Instead, animation is an intrinsic part of the framework. You define animations with declarative tags, and WPF puts them into action automatically.
- **Support for audio and video media.** Previous user interface toolkits, such as Windows Forms, were surprisingly limited when dealing with multimedia. But WPF includes support for playing any audio or video file supported by Windows Media Player, and it allows you to play more than one media file at once. Even more impressively, it gives you the tools to integrate video content into the rest of your user interface, allowing you to pull off exotic tricks such as placing a video window on a spinning 3-D cube.
- **Styles and templates.** Styles allow you to standardize formatting and reuse it throughout your application. Templates allow you to change the way any element is rendered, even a core control such as the button. It's never been easier to build modern skinned interfaces.
- **Commands.** Most users realize that it doesn't matter whether they trigger the Open command through a menu or through a toolbar; the end result is the same. Now that abstraction is available to your code, you can define an application command in one place and link it to multiple controls.
- **Declarative user interface.** Although you can construct a WPF window with code, Visual Studio takes a different approach. It serializes each window's content to a set of XML tags in a XAML document. The advantage is that your user interface is completely separated from your code, and graphic designers can use professional tools to edit your XAML files and refine your application's front end. (*XAML* is short for Extensible Application Markup Language, and it's described in detail in Chapter 2.)

- **Page-based applications.** Using WPF, you can build a browser-like application that lets you move through a collection of pages, complete with forward and back navigation buttons. WPF handles the messy details such as the page history. You can even deploy your project as a browser-based application that runs right inside Internet Explorer.

Windows Forms Lives On

WPF is the platform for the future of Windows user interface development. However, it won't displace Windows Forms overnight. Windows Forms is in many ways the culmination of the previous generation of display technology, which was built on GDI/GDI+ and User32.

So, which platform should you choose when you begin designing a new Windows application? If you're starting from the ground up, WPF is an ideal choice, and it offers the best prospects for future enhancements and longevity. Similarly, if you need one of the features that WPF provides and Windows Forms does not—such as 3-D drawing or page-based applications—it makes sense to make the shift. On the other hand, if you have a considerable investment in a Windows Forms–based business application, there's no need to recode your application for WPF. The Windows Forms platform will continue to be supported for years to come.

Perhaps the best part of the story is that Microsoft has invested considerable effort in building an interoperability layer between WPF and Windows Forms (which plays a similar role to the interoperability layer that allows .NET applications to continue to use legacy COM components). In Chapter 30, you'll learn how to use this support to host Windows Forms controls inside a WPF application, and vice versa. WPF offers similarly robust support for integrating with older Win32-style applications.

DirectX Also Lives On

There's one area where WPF isn't a good fit—when creating applications with demanding real-time graphics, such as complex physics-based simulators or cutting-edge action games. If you want the best possible video performance for these types of applications, you'll need to program at a much lower level and use raw DirectX. You can download the managed .NET libraries for DirectX programming at `http://msdn.microsoft.com/directx`.

■ **Note** As of WPF 3.5 SP1, Microsoft is beginning to break down some of the boundaries between DirectX and WPF. It's now possible to take DirectX content and place it inside a WPF application. In fact, you can even make it into a brush and use it to paint a WPF control, or you can make it into a texture and map it onto a WPF 3-D surface. Although WPF and DirectX integration is beyond the scope of this book, you can learn more from the MSDN documentation, starting at `http://tinyurl.com/y93cpn3`.

Silverlight

Like the .NET Framework, WPF is a *Windows-centric technology*. That means that WPF applications can be used only on computers running the Windows operating system. Browser-based WPF applications are similarly limited—they can run only on Windows computers, although they support both the Internet Explorer and Firefox browsers.

These restrictions won't change—after all, part of Microsoft's goal with WPF is to take advantage of the rich capabilities of Windows computers and its investment in technologies such as DirectX. However, Silverlight is designed to take a subset of the WPF platform, host it in any modern browser using a plug-in (including Firefox, Google Chrome, and Safari), and open it up to other operating systems (such as Linux and Mac OS). This is an ambitious project that's attracted considerable developer interest.

In many ways, Silverlight is based on WPF, and it incorporates many of WPF's conventions (such as the XAML markup you'll learn about in the next chapter). However, Silverlight also leaves out certain feature areas, such as true three-dimensional drawing or rich document display. New features may appear in future Silverlight releases, but the more complex ones might never make the leap.

The ultimate goal of Silverlight is to provide a powerful developer-oriented competitor for Adobe Flash. However, Flash has a key advantage—it's used throughout the Web, and the Flash plug-in is installed just about everywhere. To entice developers to switch to a new, less-established technology, Microsoft will need to make sure Silverlight has next-generation features, rock-solid compatibility, and unrivaled design support.

■ **Note** Silverlight has two potential audiences: web developers who are seeking to create more interactive applications and Windows developers who are seeking to get a broader reach for their applications. To learn more about Silverlight, refer to a dedicated book such as *Pro Silverlight 3 in C#*, or surf to `http://silverlight.net`.

Resolution Independence

Traditional Windows applications are bound by certain assumptions about resolution. Developers usually assume a standard monitor resolution (such as 1024 by 768 pixels), design their windows with that in mind, and try to ensure reasonable resizing behavior for smaller and larger dimensions.

The problem is that the user interface in traditional Windows applications isn't scalable. As a result, if you use a high monitor resolution that crams pixels in more densely, your application windows become smaller and more difficult to read. This is particularly a problem with newer monitors that have high pixel densities and run at correspondingly high resolutions. For example, it's common to find consumer monitors (particularly on laptops) that have pixel densities of 120 dpi or 144 dpi (dots per inch), rather than the more traditional 96 dpi. At their native resolution, these displays pack the pixels in much more tightly, creating eye-squintingly small controls and text.

Ideally, applications would use higher pixel densities to show more detail. For example, a high-resolution monitor could display similarly sized toolbar icons but use the extra pixels to render sharper graphics. That way, you could keep the same basic layout but offer increased clarity and detail. For a variety of reasons, this solution hasn't been possible in the past. Although you can resize graphical content that's drawn with GDI/GDI+, User32 (which generates the visuals for common controls) doesn't support true scaling.

WPF doesn't suffer from this problem because it renders all user interface elements itself, from simple shapes to common controls such as buttons. As a result, if you create a button that's 1 inch wide on your computer monitor, it can remain 1 inch wide on a high-resolution monitor—WPF will simply render it in greater detail and with more pixels.

This is the big picture, but it glosses over a few details. Most importantly, you need to realize that WPF bases its scaling on the *system* DPI setting, not the DPI of your physical display device. This makes perfect sense—after all, if you're displaying your application on a 100-inch projector, you're probably standing several feet back and expecting to see a jumbo-size version of your windows. You don't want