SPECIAL EXCERPT

Microsoft

PREVIEW CONTENT

Complete book available Fall 2010

Programming Windows® Phone 7 Series

Charles Petzold

Introduction

This is a short "draft preview" of a much longer ebook that will be completed and published later this year. That later edition will be brilliantly conceived, exquisitely structured, elegantly written, delightfully witty, and refreshingly free of bugs, but this draft preview is none of that. It is very obviously a work-in-progress that was created under an impossible timeframe while targeting quickly evolving software.

Even with this book's defects and limited scope, I hope it helps get you started in writing great programs for the Windows Phone 7 Series. Visit www.charlespetzold.com/phone for information about this book and later editions.

My Assumptions About You

I assume that you know the basic principles of .NET programming and you have a working familiarity with the C# programming language. If not, you might benefit from reading my free online book *.NET Book Zero: What the C or C++ Programmer Needs to Know about C# and the .NET Framework*, available from my Web site at www.charlespetzold.com/dotnet.

Using This Book

To use this book properly you'll need to download and install the Windows Phone Developer Tools, which includes Visual Studio 2010 Express for Windows Phone and an on-screen Windows Phone Emulator to test your programs in the absence of an actual device.

You'll want to check the release notes for the Windows Phone Developer Tools, but it is my experience that Visual Studio 2010 Express for Windows Phone can be installed on top of an existing installation of the Visual Studio 2010 Release Candidate.

When finalizing these six chapters, I've been working with a Windows Phone Developer Tools package dating from March 5, 2010. Although I pleaded, threatened, whined, and even threw a tantrum, I have not yet held an actual Windows Phone in my hands.

The Windows Phone 7 Series supports multitouch, and working with multitouch is an important part of developing programs for the phone. When using the Windows Phone Emulator, mouse clicks and mouse movement on the PC are used to mimic touch on the emulator.

You can run the Windows Phone Emulator on a PC with a multitouch display under Windows 7, but in my experience, this configuration doesn't offer any real benefits over the mouse.

Apparently touch events on the PC screen are translated to mouse events for the emulator, which are then translated back to touch events for your phone program.

If you're writing an XNA program for the phone, and getting multitouch working well is critical, and you don't have an actual phone quite yet, you might want to consider adapting your program for the Zune HD and testing it there.

The Essential People

This book owes its existence to Dave Edson—an old friend from the early 1990s era of *Microsoft Systems Journal*—who had the brilliant idea that I would be the perfect person to write a tutorial on Windows Phone. Dave arranged for me to attend a technical deep dive on the phone at Microsoft way back in December 2009, and I was hooked. Todd Brix gave the thumbs up on the book, and Anand Iyer coordinated the project with Microsoft Press.

At Microsoft Press, Ben Ryan launched the book and Devon Musgrave had the unenviable job of trying to make my hastily written code and prose resemble an actual book in virtually no time at all. (We go way back: You'll see Ben and Devon's names on the bottom of the copyright page of *Programming Windows*, fifth edition.)

Dave Edson also reviewed chapters and served as conduit to the Windows Phone team to deal with my technical problems and questions. Aaron Stebner provided essential guidance; Michael Klucher reviewed chapters, and Kirti Deshpande, Charlie Kindel, Casey McGee, and Shawn Oster also had important things to tell me. Thanks also to Bonnie Lehenbauer for reviewing one of the chapters at the last minute.

My wife Deirdre Sinnott was a marvel of patience and tolerance over the past two months as she dealt with an author given to sudden mood swings, insane yelling at the computer screen, and the conviction that the difficulty of writing a book relieves one of the responsibility of performing basic household chores.

Alas, I can't blame any of them for bugs or other problems with this book. Those are all mine.

Charles Petzold
New York City
March 10, 2010

Part I

Getting Started

Chapter 1
Phone Hardware + Your Software

Sometimes it becomes apparent that previous approaches to a problem haven't quite worked the way you anticipated. Perhaps you just need to clear away the smoky residue of the past, take a deep breath, and try again with a new attitude and fresh ideas. In golf, it's known as a "mulligan"; in schoolyard sports, it's called a "do-over"; and in the computer industry, we say it's a "reboot."

A reboot is what Microsoft has initiated with its new approach to the mobile phone market. On February 15, 2010, at the Mobile World Congress in Barcelona, Microsoft CEO Steve Ballmer unveiled the Microsoft Windows Phone 7 Series and promised a product introduction in time for year-end holiday shopping. With its clean look, striking fonts, and new organizational paradigms, Windows Phone 7 Series not only represents a break with the Windows Mobile past but also differentiates itself from other smartphones currently in the market.

For programmers, the news from Barcelona was certainly intriguing but hardly illuminating. Exactly how do we write programs for this new Windows Phone 7 Series? Developers detected a few hints but no real facts. The really important stuff wouldn't be disclosed until mid-March at MIX 2010 in Las Vegas.

Silverlight or XNA?

Intelligent speculation about the application platform for the Windows Phone 7 Series has gravitated around two possibilities: Silverlight and XNA.

Since about 2008, programmers have been impatiently awaiting the arrival of a mobile version of Silverlight. Silverlight, a spinoff of the client-based Windows Presentation Foundation (WPF), has already given Web programmers unprecedented power to develop sophisticated user interfaces with a mix of traditional controls, high-quality text, vector graphics, media, animation, and data binding that run on multiple platforms and browsers. Many programmers thought Silverlight would be an excellent platform for writing applications and utilities for smartphones.

XNA—the three letters stand for something like “XNA is Not an Acronym”—is Microsoft’s game platform supporting both 2D sprite-based and 3D graphics with a traditional game-loop architecture. Although XNA is mostly associated with writing games for the Xbox 360 console, developers can also target the PC itself, as well as Microsoft’s classy audio player, the Zune. The 2009 release of the Zune HD particularly seemed to suggest a mobile future built around the device’s revamped graphics and multitouch navigation. For many Zune HD users, the most disappointing feature of the device was its inability to make phone calls!

Either Silverlight or XNA would make good sense as the application platform for the Windows Phone 7 Series, but the decision from Microsoft is:

Both!

The Windows Phone 7 Series supports programs written for either Silverlight or XNA. And this we call “an embarrassment of riches.”

Targeting Windows Phone 7 Series

The Windows Phone 7 Series operating system exposes classes defined by the .NET Compact Framework. All programs for the phone are written in managed code. At the present time, C# is the only supported programming language. Programs are developed in Microsoft Visual Studio 2010 Express for Windows Phone, which includes XNA Game Studio and access to an on-screen phone emulator. You can develop visuals for Silverlight application using Microsoft Expression Blend.

A program for Windows Phone 7 Series must target using either Silverlight or XNA. It would surely be great to mix them up a bit and combine Silverlight and XNA visuals in the same program. Maybe that will be possible in the future, but it’s not possible now except for some cross-library use. Before you create a Visual Studio project, you must decide whether your million-dollar idea is a Silverlight program or an XNA program.

Generally you’ll choose Silverlight for writing programs you might classify as applications or utilities. These programs use the Extensible Application Markup Language (XAML) to define a layout of user-interface controls and panels. Code-behind files can also perform some initialization but are generally relegated to handling events from the controls. Silverlight is great for bringing to the Windows Phone the style of Rich Internet Applications (RIA), including media and the Web.

XNA is primarily for writing high-performance games. For 2D games, you define sprites and backgrounds based around bitmaps; for 3D games you define models in 3D space. The action of the game, which includes moving graphical objects around the screen and polling for user input, is synchronized by the built-int XNA game loop.

The differentiation between Silverlight-based applications and XNA-based games is convenient but not restrictive. You can certainly use Silverlight for writing games and you can even write traditional applications using XNA, although doing so might sometimes be challenging. In this book I'll try to show you some examples—games in Silverlight and utilities in XNA—that push the envelope.

In particular, Silverlight might be ideal for games that are less graphically oriented, or that use vector graphics rather than bitmap graphics, or that are paced by user-time rather than clock-time. A Tetris-type program might work quite well in Silverlight. You'll probably find XNA to be a bit harder to stretch into Silverlight territory, however. Implementing a list box in XNA might be considered "fun" by some programmers but a torture by many others.

Microsoft has been positioning Silverlight as the front end or "face" of the cloud, so cloud services and Windows Azure form an important part of Windows Phone 7 Series development. The Windows Phone is "cloud-ready." Programs are location-aware, have access to maps and other data through Bing and Windows Live, and can interface with social networking sites. Among the available cloud services is Xbox Live, which allows XNA-based programs to participate in online multiplayer games, and can also be accessed by Silverlight applications.

Programs you write for the Windows Phone 7 Series will be sold and deployed through the Windows Phone Marketplace, which provides registration services and certifies that programs meet minimum standards of reliability, efficiency, and good behavior.

The Hardware Chassis

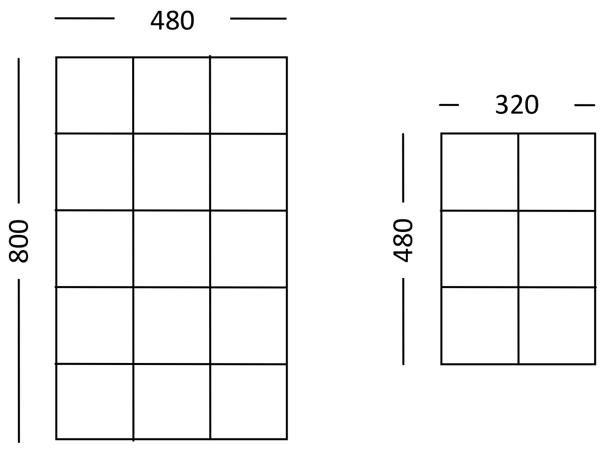
Developers with experience targeting Windows Mobile devices of the past will find significant changes in Microsoft's strategy for the Windows Phone 7 Series. Microsoft has been extremely proactive in defining the hardware specification. There are only two possible screen sizes, and many other hardware features are guaranteed to exist on each device.

For devices that become part of the Windows Phone 7 Series, Microsoft has established a hardware specification often referred to as a "chassis." The front of the phone consists of a multitouch display and three hardware buttons generally positioned in a row below the display. From left to right, these buttons are called Back, Start, and Search:

- **Back** Programs written for the phone are required to use the Back button to exit themselves. In addition, programs can use this button in connection with their own navigation needs, much like the Back button on a Web browser.
- **Start** This button powers up the phone itself, and when the phone is running, takes the user to the start screen; this button is inaccessible to programs running on the phone.
- **Search** Phone programs can ignore this button or use it for any program function related to search.
- The screen can have one of two display sizes: 480 × 800 pixels or 320 × 480 pixels. For the Windows Phone 7 Series, there are no other screen options, so obviously these two screen sizes play a very important role in phone development.

In theory, it's usually considered best to write programs that adapt themselves to any screen size, but that's not always possible, particularly with game development. You will probably find yourself specifically targeting these two screen sizes, even to the extent of having *if*/*else* clauses and different XAML files for layout that is size-dependent.

I will generally refer to these two sizes as the "large" screen and the "small" screen. The greatest common denominator of the horizontal and vertical dimensions of both screens is 160, so you can visualize the two screens as multiples of 160-pixel squares:



I’m showing these screens in portrait mode because that’s usually the way smartphones are designed. The screen of the original Zune is 240 × 320 pixels; the Zune HD is 272 × 480.

Of course, phones can be rotated to put the screen into landscape mode. This is particularly useful for watching movies. Some programs might require the phone to be held in a certain orientation; others might be more adaptable. Generally you’ll want to write your Silverlight applications to adjust themselves to orientation; new events are available specifically for the purpose of detecting orientation change, and some orientation shifts are handled automatically. In contrast, game programmers can usually impose a particular orientation on the user. A solitaire card game probably works much better in landscape mode than portrait mode, for example.

In portrait mode, the small screen is half of an old VGA screen (that is, 640 × 480). In landscape mode, the large screen has a dimension sometimes called WVGA (“wide VGA”). In landscape mode, the small screen has an aspect ratio of 3:2 or 1.5; the large screen has an aspect ratio of 5:3 or 1.66…. Neither of these matches the aspect ratio of television, which for standard definition is 4:3 or 1.33… and for high-definition is 16:9 or 1.77…. The Zune HD screen has an aspect ratio of 16:9.

Like many recent phones and the Zune HD, the Windows Phone 7 Series displays will likely use OLED (“organic light emitting diode”) technology. OLEDs are different from flat displays of the past in that power consumption is directly proportional to the light emitted from the display.

For example, an OLED display consumes less than half the power of an LCD display of the same size, but only when the screen is mostly black. For an all-white screen, an OLED consumes more than three times the power of an LCD.

Battery life is extremely important on mobile devices, so this characteristic of OLED displays has some profound consequences. It means that we will be designing screens for our Windows Phone programs that have mostly black backgrounds—if we care about issues like power consumption, and we should. If the aesthetic of the Windows Phone 7 Series can be summed up in a tiny true phrase, it is this: Black is Beautiful.

Most user input to a Windows Phone program will come through touching the screen with fingers. The screens incorporate capacitance-touch technology, which means they respond to a human fingertip but not to a stylus or other forms of pressure. Windows Phone screens are required to respond to at least four simultaneous touch-points.

A hardware keyboard is optional. Keep in mind that phones can be designed in different ways, so when the keyboard is in use, the screen might be in either portrait mode or landscape mode. A Silverlight program that uses keyboard input *must* respond to orientation events so that the user can both view the screen and use the keyboard without wondering what idiot designed the program sideways. An on-screen keyboard is also provided, known in Windows circles as the Soft Input Panel or SIP.

Neither the hardware keyboard nor the on-screen keyboard is available to XNA programs.

Sensors and Services

The Windows Phone 7 Series is required to contain several other hardware devices—sometimes called sensors—and provide some software services, perhaps through the assistance of hardware, as described here:

- **Wi-Fi** The phone has Wi-Fi for Internet access. Software on the phone includes a version of Internet Explorer.
- **Camera** The phone must have at least a 5-megapixel camera with flash. Programs can register themselves as a Photos Extra Application and appear on a menu to obtain access to photographed images, perhaps for some image processing.

- **Accelerometer** An accelerometer detects acceleration, which in physics is a change in velocity. When the camera is still, the accelerometer responds to gravitation. Programs can obtain a three-dimensional vector that indicates how the camera is oriented with respect to the earth. (This technique will not work if the user is viewing the device while suspended upside-down or in a weightless spacecraft.) An XNA program can use this information to select a suitable orientation of the video display; a Silverlight program can handle explicit orientation events instead. The accelerometer can also detect sharp movements of the phone.

- **Compass** The compass detects orientation relative to magnetic north.

- **Location** If the user so desires, the phone can use multiple strategies for determining where it is geographically located. The phone might incorporate a hardware GPS device or it might use the Web for determining location. Programs running on the phone can obtain geographic coordinates (longitude, latitude, and altitude), civic addresses (including street address, city, state or province, country, and even building and floor, if available). If the phone is moving, course and speed might also be available.

- **Speech** The phone supports both speech synthesis and speech recognition through classes that are also part of .NET 4.0.

- **Vibration** The phone can be vibrated through program control.

- **Push Notifications** Some Web services would normally require the phone to frequently poll the service to obtain updated information. This can drain battery life. To help out, a push notification service has been developed that will allow any required polling to occur outside the phone and for the phone to receive notifications only when data has been updated.

That’s quite a list, but although I haven’t been able to confirm this, a persistent rumor indicates that a Windows Phone device can also be used to make and receive telephone calls.

Continuity and Innovation

I'm no market analyst. I'm just a programmer. Don't ask me if I think the Windows Phone 7 Series will be a commercial success. Market forces are a complete mystery to me. I don't know form factors. I can't judge if the phone is the right size, or well proportioned, or anything else. The visuals look good to me, but I don't trust my instincts about visual design. I don't know if the phone is slick enough for the cool kids and mainstream enough for everyone else.

But as a programmer I can tell you this: coding for Windows Phone 7 Series is a total blast!

The Windows Phone 7 Series has been characterized as representing a severe break with the past. If you compare it with past versions of Windows Mobile, that is certainly true. But the support of Silverlight, XNA, and C# are not breaks with the past, but a balance of continuity and innovation. As young as they are, Silverlight and XNA have already proven themselves as powerful and popular platforms. Many skilled programmers are already working with either one framework or the other—probably not so many with both just yet—and they have expressed their enthusiasm with a wealth of online information and communities. C# has become the favorite language of many programmers (myself included), and developers can use C# to share libraries between their Silverlight and XNA programs as well as those written for other .NET environments.

Adapting one's experience of these frameworks to the small, fixed-size screens of this multitouch input phone is both challenging and fun. I've been coding for Microsoft-based operating systems for over 25 years—and often writing about my experiences—and I've had a great time coding for the phone even though I have not yet actually held one in my hands. I have a strong sense that I won't be alone and that Windows Phone 7 Series will be a break-out product because we programmers will create an extensive array of applications and games that will make it sing.

So enough of the preliminaries: let's get coding.

Chapter 2
Hello, Windows Phone

A typical "hello, world" program that just displays a little bit of text might seem silly to nonprogrammers, but programmers have discovered that such a program serves at least two useful purposes. First, the program provides a way to examine how easy (or ridiculously complex) it is to display a simple text string. Second, it gives the programmer an opportunity to experience the process of creating, compiling, and running a program without a lot of distractions. When developing programs that run on a mobile device, this process is little more complex than customary because you'll be creating and compiling programs on the PC but you'll be deploying and running them on an actual phone or at least an emulator.

This chapter presents programs for both Microsoft Silverlight and Microsoft XNA that display the text "Hello, Windows Phone!" followed by slightly enhanced versions that respond to touch in a very rudimentary manner.

In real-world programming, you'll probably use XNA mostly for games, and Silverlight for programs that you might classify as applications or utilities, although Silverlight is also a fine game platform if you don't need 3D. Following this chapter, the book splits into different parts for Silverlight and XNA. I suspect that some developers will stick with either Silverlight or XNA exclusively and won't even bother learning the other environment. I hope that's not a common attitude. The good news is that Silverlight and XNA are so dissimilar that you can probably bounce back and forth between them without confusion!

Just to make these programs a little more interesting, I want to establish two rules:

- The text will be displayed in the center of the display, or at least in the center of an area of the display allocated for program output.
- The text will be white on a dark background.

As you'll recall, the OLED screens likely to be used on Windows Phone 7 Series devices consume much less power if pixels are mostly dark. The second rule ensures that our very first programs for the phone won't be power pigs!

If you want to play along, you should have the Windows Phone Developer Tools installed, which includes Visual Studio 2010 Express for Windows Phone and the Windows Phone Emulator.

A First Silverlight Phone Program

From the File menu of Visual Studio 2010 Express for Windows Phone, select New Project. In the dialog box, on the left under Installed Templates, choose Visual C# and then Silverlight for Windows Phone. In the middle area, choose Windows Phone Application. Select a location for the project, and enter the project name: SilverlightHelloPhone. You probably don't want to create an unnecessary separate directory for the single-project solution, so uncheck the "Create directory for solution" check box. Click OK.

As the project is created you'll see an image of a large-screen phone in portrait mode: 480 × 800 pixels in size. This is the design view. Although you can interactively pull controls from a toolbox to design the application, I'm going to focus instead on showing you how to write your own markup.

Several files have been created for this SilverlightHelloPhone project. They are listed under the project name in the Solution Explorer over at the right. In the Properties folder are three files that you can usually ignore when you're just creating little sample Silverlight programs for the phone. Only when you're actually in the process of making a real application do these files become important.

However, you might want to open the WMAppManifest.xml file. In the App tag near the top, you'll see the attribute:

```
Title="SilverlightHelloPhone"
```

That's just the project name. Insert some spaces to make it a little friendlier:

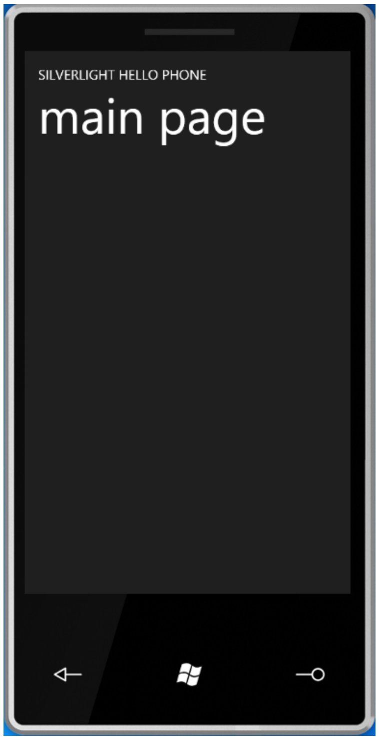
```
Title="Silverlight Hello Phone"
```

This is the name used by the phone and the phone emulator to display the program. If you're really ambitious, you can also edit the ApplicationIcon.png file that the phone uses to visually symbolize the program.

In the standard toolbar under the program's menu, you'll see a drop-down list probably displaying "Windows Phone 7 Emulator." The other choice is "Windows Phone 7 Device." This is how you deploy your program to either the emulator or an actual phone connected to your computer via USB.

Just to see that everything's working OK, press F5 (or select Start Debugging from the Debug menu). Your program will quickly build and in the status bar you'll see the text "Connecting to Windows Phone 7 Emulator…" The first time you use the emulator during a session, this might take a minute or so. If you leave the emulator running between edit/build/run cycles, Visual Studio 2010 Express for Windows Phone doesn't need to establish this connection again.

Soon the phone emulator will appear on the desktop and you'll see the opening screen, followed soon by this little do-nothing Silverlight program as it is deployed and run on the emulator. On the phone you'll see pretty much the same image you saw in the design view. Here's the emulator displayed at about half size on this page:



You can terminate execution of this program and return to to editting the program either though Visual Studio 2010 for Windows Phone (using Shift-F5 or by selecting Stop Debugging from the Debug menu) or by exiting the program by clicking the emulator Back or Start button.

Don't exit the emulator itself by clicking the X at the top of the emulator's attached floating menu (not shown in the screen shot)! Keeping the emulator running will make subsequent deployments go much faster.

While the emulator is still running, it retains all programs deployed to it. If you click the arrow at the upper-right of the Start screen, you'll get a list that will include this program identified by the text "Silverlight Hello Phone". The program will disappear from this list when you exit the emulator.

Back in Visual Studio 2010 Express for Windows Phone, you'll see that the creation of the SilverlightHelloPhone project also resulted in the creation of two pairs of skeleton files: App.xaml and App.xaml.cs, and MainPage.xaml and MainPage.xaml.cs. The App.xaml and MainPage.xaml files are Extensible Application Markup Language (XAML) files, while App.xaml.cs and MainPage.xaml.cs are C# code files. This peculiar naming scheme is meant to imply that the two C# code files are "code-behind" files associated with the two XAML files. They provide code in support of the markup.

I want to give you a little tour of these four files. If you look at the App.xaml.cs file, you'll see a namespace definition that is the same as the project name and a class named *App* that derives from the Silverlight class *Application*. Here's an excerpt showing the general structure:

Silverlight Project: SilverlightHelloPhone   File: App.xaml.cs (excerpt)

```
namespace SilverlightHelloPhone
{
    public partial class App : Application
    {
        public App()
        {
            …             InitializeComponent();
        }    }
}
```

All Silverlight programs contain an *App* class that derives from *Application*; this class performs application-wide initialization, startup, and shutdown chores. You'll notice this class is defined as a *partial* class, meaning that there should be another file that contains additional members of the *App* class.

The project also contains an App.xaml file, which has an overall structure like this:

Silverlight Project: SilverlightHelloPhone   File: App.xaml (excerpt)

```
<Application x:Class="SilverlightHelloPhone.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" … >
…
</Application>
```

You'll recognize this file as XML, but more precisely it is a XAML file, which is an important part of Silverlight programming.

The App.xaml file is often used for storing *resources* that are used throughout the application. These could include color schemes, gradient brushes, styles, and so forth, and a bunch of these resources are already present in App.xamle. The syntax of these resources and how they are used is not exactly an "advanced" feature but nevertheless shouldn't be introduced in Chapter 2. I'll come back to this file in subsequent chapters.

The root element is *Application*, which is the Silverlight class that the *App* class derives from. The first XML namespace declaration ("xmlns") is the standard namespace for Silverlight, and it helps the compiler locate and identify Silverlight classes such as *Application*. As with most XML namespace declarations, this URI doesn't actually point to anything; it's just a URI that Microsoft owns and which it has defined for this purpose.

The second XML namespace declaration is associated with XAML itself, and it allows the file to reference some elements and attributes that are part of XAML itself rather than specifically Silverlight. By convention, this namespace is associated with a prefix of "x" (meaning "XAML").

Among the several attributes supported by XAML and referenced with this "x" prefix is *Class*, which is often pronounced "x class." In this file *x:Class* is assigned the name *SilverlightHelloPhone.App*. This means that a class named *App* in the .NET *SilverlightHelloPhone* namespace derives from the Silverlight *Application* class, the root element. It's the same class definition as in the App.xaml.cs file but just with different syntax.

The App.xaml.cs and App.xaml files really define two halves of the same *App* class. During compilation, Visual Studio parses App.xaml and generates another code file named App.g.cs. The "g" stands for "generated." If you want to look at this file, you can find it in the \obj\Debug subdirectory of the project. The App.g.cs file contains another partial definition of the *App* class, and it contains a method named *InitializeComponent* that is called from the constructor in the App.xaml.cs file.

Towards the top of the App.xaml file you'll also see this markup:

```
<Application.RootVisual>
    <phoneNavigation:PhoneApplicationFrame x:Name="RootFrame" Source="/MainPage.xaml"/>
</Application.RootVisual>
```

The syntax might be a obscure at this point, but it will become clearer in Chapter 3, "Code and XAML." This markup essentially instantiates an object of type *PhoneApplicationFrame* and assigns to its *Source* property a text string referencing MainPage.xaml. (Another XML namespace declaration for "phoneNavigation" is required for *PhoneApplicationFrame* because the class is not a part of standard Silverlight.) This *PhoneApplicationFrame* object is then set to the *RootVisual* property of the *App* object.

This is how the *MainPage* class comes into being. *MainPage* is the second major class in the program and is defined in the second pair of files, MainPage.xaml and MainPage.xaml.cs. In smaller Silverlight programs, it is in these two files that you'll be spending most of your time.

Aside from a long list of using directives, the MainPage.xaml.cs file is very simple:

Silverlight Project: SilverlightHelloPhone   File: MainPage.xaml.cs (excerpt)

```
namespace SilverlightHelloPhone
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
            SupportedOrientations = SupportedPageOrientation.Portrait |
                                    SupportedPageOrientation.Landscape;
        }
    }
}
```

Again, we see another *partial* class definition. This one defines a class named *MainPage* that derives from the Silverlight class *PhoneApplicationPage*. This is the class that defines the visuals you'll actually see on the screen when you run the SilverlightHelloPhone program.

The other half of this *MainPage* class is defined in the MainPage.xaml file:

Silverlight Project: SilverlightHelloPhone   File: MainPage.xaml (excerpt)

```
<phoneNavigation:PhoneApplicationPage
        x:Class="SilverlightHelloPhone.MainPage"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:phoneNavigation="clr-namespace:Microsoft.Phone.Controls;assembly= …
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d" d:DesignHeight="800" d:DesignWidth="480" …>
    <Grid … >
    …
    </Grid>
</phoneNavigation:PhoneApplicationPage>
```

These first three XML namespace declarations are the same as in App.xaml. As in the App.xaml file, an *x:Class* attribute also appears in the root element. Here it indicates that the *MainPage* class in the *SilverlightHelloPhone* namespace derives from the Silverlight

PhoneApplicationPage class. This *PhoneApplicationPage* class requires its own XML namespace declaration because it is not a part of standard Silverlight.

Everything else in the root element is for the benefit of XAML design programs, such as Expression Blend and the designer in Visual Studio 2010 Express for Windows Phone itself. The XML namespace prefix "d" is associated with designer-related attributes such as *DesignHeight* and *DesignWidth*. The namespace declaration "mc" (for "markup compatibility") and the *Ignorable* attribute indicate that these attributes should be ignored by other programs, and during compilation, they are. You can delete these namespace declarations and attributes with no effect on the program as it is compiled and run.

The compilation of the program generates a file name MainPage.g.cs that contains another partial class definition for *MainPage* (you can look at it in the \obj\Debug subdirectory) and the *InitializeComponent* method called from the constructor in MainPage.xaml.cs.

In theory, the App.g.cs and MainPage.g.cs files generated during the build process are solely for internal use by the compiler and can be ignored by the programmer. However, sometimes when a buggy program raises an exception, one of these files comes popping up in to view. It might help your understanding of the problem to have seen these files before they mysteriously appear in front of your face. However, don't try to edit these files to fix the problem! The real problem is probably somewhere in the corresponding XAML file.

The remainder of the MainPage.xaml file is occupied by a couple nested *Grid* elements with a couple *TextBlock* elements. This nesting defines a *visual tree* of elements that is displayed by the program, including the "MY APPLICATION" and "page title" headings you see in the designer and the emulator.

The *Grid* class derives from *Panel* and is one of the primary layout elements of Silverlight. Panels are usually containers that host multiple user-interface elements and visually organize them on the screen; I'll be discussing panels in more detail in Chapter 4, "Presentation and Layout."

The *TextBlock* class derives from *FrameworkElement* and (as the name implies) is used for displaying blocks of text. *Panel* also derives from *FrameworkElement*, and for that reason, these visuals are often referred to collectively as "elements." This is the same word used to describe XML components delimited by start and end tags, so it's a very convenient word in Silverlight programming. Another important class that derives from *FrameworkElement* is *Control*, from which the familiar *Button*, *Slider*, *ListBox*, and others derive.

For now, simply observe that in XAML, Silverlight classes such as *PhoneApplicationPage*, *Grid,* and *TextBlock* become XML elements. Properties of these classes (such as the *Background* property of *Grid* and the *Text* property of *TextBlock*) become XML attributes.

It is the MainPage.xaml file that we need to edit to create a Silverlight application that displays some text in the center of its display. Towards the bottom of the file—in the nested *Grid* tags preceded by the comment "ContentGrid is empty. Place new content here"—insert a *TextBlock* element so that the existing *Grid* tags and the new *TextBlock* look like this:

Silverlight Project: SilverlightHelloPhone   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

TextBlock is the Silverlight class you'll use for displaying short blocks of text—up to a paragraph or so. *Text*, *HorizontalAlignment*, and *VerticalAlignment* are all properties of *TextBlock* that I'll describe in more detail in the next chapter.

While you're editing MainPage.xaml you might also want to fix the other *TextBlock* elements so that they make some sense. Change

```
<TextBlock Text="MY APPLICATION" … />
```

to

```
<TextBlock Text="SILVERLIGHT HELLO PHONE" … />
```

and

```
<TextBlock Text="page title" … />
```

to:

```
<TextBlock Text="main page" … />
```

It doesn't make much sense to have a page title in a Silverlight application with only a single page, and you can delete that second *TextBlock* if you'd like.

The changes you make to this XAML file will be reflected in the design view.

If you'd like the text to be displayed in a different color, you can try setting the Foreground attribute in the TextBlock tag, for example:

```
Foreground="Red"
```

You can put it anywhere in the tag as long as you leave spaces on either side. As you type this attribute, you'll see a list of colors pop up. Silverlight supports the 140 color names supported by many browsers, as well as a bonus 141st color, *Transparent*.

When Silverlight runs in a web browser, usually the background is white, so the default text color is black. When programming for the Windows Phone 7 Series, it's a good idea to keep backgrounds dark, so black text would be problem. For that reason, you'll notice a *Foreground* setting on the root element in MainPage.xaml:

```
Foreground="{StaticResource PhoneForegroundBrush}"
```

This statement references an item in the App.xaml file, which is *White*. I'll discuss the syntax and mechanism of *StaticResource* in a later chapter, and how setting the color in the root element affects the *TextBlock* at the bottom of the XAML file in the next chapter.

Another font-related attribute set on the root element is *FontSize*:

```
FontSize="{StaticResource PhoneFontSizeNormal}"
```

In App.xaml, this is defined as 20. All dimensions in Silverlight are in units of pixels, and the default *FontSize* is 11 pixels. With this default setting, you get a font that from the top of its ascenders to the bottom of its descenders, plus a little breathing room, measures approximately 11 pixels. On a common video display with a resolution of about a hundred pixels to the inch, this is a reasonable size for small but readable text. On modern hand-held devices such as phones, the pixels are packed in much more tightly—closer to about 150 dots per inch—and the 11-pixel text tends to be a little too small.

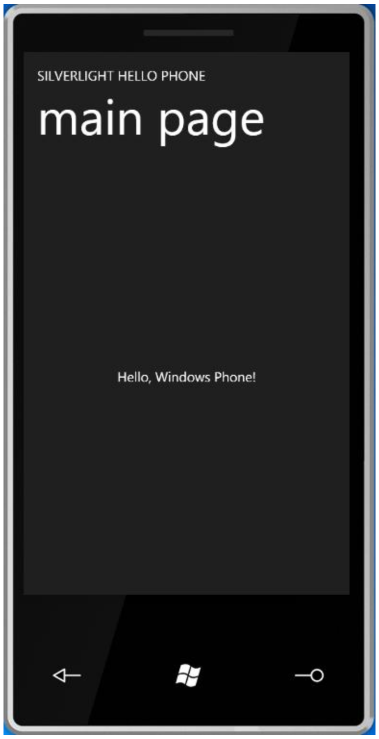
Traditionally, font sizes are expressed in units of *points*. In traditional typography, a point is very close to 1/72nd inch and in digital typography is often assumed to be exactly 1/72nd inch. A font with a size of 72 points measures approximately an inch from the top of its characters to the bottom. (I say "approximately" because the 72 points indicate a typographic design height, and it's really the creator of the font who determines exactly how large the characters of a 72-point font should be.)

Obviously, a 72-point font will have a pixel height dependent on the output device on which the font is rendered. On a 600 DPI printer, for example, the 72-point font will be 600 pixels tall. However, for many years Microsoft Windows has assumed that video displays have a resolution of 96 DPI. Under that assumption, font sizes can be converted from pixels to points by multiplying by ¾, and from points to pixels by multiplying by 4/3.

So, that 11-pixel default *FontSize* in Silverlight might be said to be equivalent to 8.25 points. Obviously this conversion works only for devices that actually have 96 DPI displays and it completely falls apart for higher resolution printers and phones, but that's the conversion that's used in some definitions of font sizes you'll see in the App.xaml file, which is why the 20-pixel font size is claimed to be 15 points.

If you're comfortable thinking of fonts in terms of point sizes, you might want to use a simple rule of thumb: Double the point size for the phone pixel size. For example, for reading text comfortably on paper, you probably wouldn't go below 8 points. On the phone, make that font 16 pixels and you'll be in good shape.

At any rate, you can now compile and run the program if you haven't already done so. Although it's not quite obvious, the *PhoneApplicationFrame* in App.xaml occupies the entire visual display; the *MainPage* (which derives from *PhoneApplicationPage*) occupies the entire area of the frame. The outermost *Grid* in *MainPage* occupies the entire area of the page. This is split into a title area on top and something we might call a "content" area under that. The new *TextBlock* with is centered within that area:



As simple as it is, this program demonstrates some essential concepts of Silverlight programming, including dynamic layout. The XAML file defines a layout of elements in what is called a *visual tree*. These elements are capable of arranging themselves dynamically. By assigning the *HorizontalAlignment* and *VerticalAlignment* properties we can put an element in the center of another element, or in along one of the edges or in one of the corners. *TextBlock* is one of a number of possible elements you can use in a Silverlight program; others include bitmap images, movies, and familiar controls like buttons, sliders, and list boxes.

From the menu that hangs off the emulator, you can turn the "phone" sideways: You'll see that the content of the page shuffles itself around to the new orientation automatically.

Before the end of the chapter, you'll see how the C# code-behind file gets involved in providing event handling for elements in the XAML file and how the code file can then interact with those elements.

An XNA Program for the Phone

While text is often prevalent in Silverlight applications, it usually doesn't show up a whole lot in graphical games. In games, text is usually relegated to describing how the game works or displaying the score, so the very concept of a "hello, world" program doesn't quite fit in with the whole XNA programming paradigm.

In fact, XNA doesn't even have any built-in fonts. You might think that an XNA program running on the phone can make use of the same native fonts as Silverlight programs, but this is not so. Silverlight uses vector-based TrueType fonts and XNA doesn't know anything about such exotic concepts. To XNA, everything is a bitmap, including fonts.

If you wish to use a particular font in your XNA program, that font must be embedded into the executable as a collection of bitmaps for each character. XNA Game Studio makes the actual process of font embedding very easy, but it raises some thorny legal issues. You can't legally distribute an XNA program unless you can also legally distribute the embedded font, and with most of the fonts distributed with Windows itself or Windows applications, this is not the case.

To help you out of this legal quandary, Microsoft licensed some fonts from Ascender Corporation specifically for the purpose of allowing you to embed them in your XNA programs. Here they are:

Kootenay	**Pericles**
Lindsey	Pericles Light
Miramonte	Pescadero
Miramonte Bold	**Pescadero Bold**

Notice that the Pericles font uses small capitals for lower-case letters, so it's probably suitable only for headings.

Let's begin. Bring up Visual Studio 2010 Express for Windows Phone again.E. From the File menu select New Project. On the left of the dialog box, select Visual C# and XNA Game Studio 4.0. In the middle select Windows Phone Game (4.0). Select a location and enter a project name of XnaHelloPhone. XNA projects contain at least two directories, so you'll probably want to make sure "Create directory for solution" has been clicked. Click OK.

As with the Silverlight project the Solution Explorer at the right displays a list of the files created for this new project. Although it isn't essential, you'll probably want to open the WindowsPhoneManifest.xml file in the Properties folder, and in the *App* element, change the *Title* attribute to "XNA Hello Phone." You can also open the GameThumbnail.png file and create a less generic icon for the program.

This XNA program will require a font, which is considered part of the "content" of an XNA program. XNA programs usually contain lots of content, mostly bitmaps and 3D models, but fonts as well. To embed a font into this program, right-click the Content folder (labeled "XnaHelloPhoneContent (Content)" and from the pop-up menu choose Add and New Item. Choose Sprite Font, leave the filename as SpriteFont1.spritefont, and click Add.

The word "sprite" is common in game programming and usually refers to a small bitmap that can be moved very quickly, much like the sprites you might encounter in an enchanted forest. In XNA, even fonts are sprites.

You'll see SpriteFont1.spritefont show up in the file list of the Content directory, and you'll be able to edit a extremely well commented XML file describing the font.

XNA Project: XnaHelloPhone   File: SpriteFont1.spritefont (complete w/o comments)

```
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">
    <FontName>Kootenay</FontName>
    <Size>14</Size>
    <Spacing>0</Spacing>
    <UseKerning>true</UseKerning>
    <Style>Regular</Style>
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Within the *FontName* tags you'll see Kootenay, but you can change that to one of the other fonts I listed earlier. If you want Pericles Light, put the whole name in there, but if you want Miramonte Bold or Pescadero Bold, use just Miramonte or Pescadero, and enter the word Bold between the Style tags. You can use Bold for the other fonts as well, but for the other fonts, bold will be synthesized, while for these two fonts, you'll get the font actually designed for bold.

The Size tags indicate the point size of the font. In XNA you deal with pixel coordinates and dimensions. Although you specify a point size here—remember that in digital typography a point is 1/72nd inch—you'll get a font with a pixel dimension based on a resolution of 96 DPI. The point size of 14 becomes a pixel size of 18-2/3 within your XNA program. The *CharacterRegions* section of the file indicates the ranges of hexadecimal Unicode character encodings you need. The default setting from 0x32 through 0x126 includes all the noncontrol characters of the ASCII character set.

The filename of SpriteFont1.spritefont is not very descriptive. I like to rename it to something that describes the actual font; if you're sticking with the default font settings, you can rename it to Kootenay14.spritefont. If you look at the properties for this file—right-click the filename and select Properties—you'll see an Asset Name that is also the filename without the extension: Kootenay14. This Asset Name is what you use to refer to the font in your program. If you want to confuse yourself, you can change the Asset Name independently of the filename.

In its initial state, the XNAHelloPhone project contains two C# code files: Program.cs and Game1.cs. The first is very simple and, moreover, is actually irrelevant for Windows Phone 7 Series games! A preprocessor directive enables the *Program* class only if a symbol of WINDOWS or XBOX is defined. When compiling Windows Phone programs, the symbol WINDOWS_PHONE is defined instead.

For most small games, you'll be spending all your time in the Game1.cs file. The *Game1* class derives from *Game* and in its pristine state it defines two fields: *graphics* and *spriteBatch*. To those two fields I want to add three more:

XNA Project: XnaHelloPhone   File: Game1.cs (excerpt showing fields)

```
namespace XnaHelloPhone
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        string text = "Hello, Windows Phone!";
        SpriteFont kootenay14;
        Vector2 textPosition;

        …
    }
}
```

These three new fields simply indicate the text that the program will display, the font it will use to display it, and the position of the text. In XNA everything is positioned using pixel coordinates relative to the upper-left corner of the display. The *Vector2* structure has two fields named *X* and *Y* of type *float*. For performance purposes, all floating-point values in XNA are single-precision. (Silverlight is all double-precision.) The *Vector2* structure is often used for two-dimensional points, sizes, and even vectors.

When the game is run on the phone, the Game1 class is instantiated and the *Game1* constructor is executed. This standard code is provided for you:

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    TargetElapsedTime = TimeSpan.FromSeconds(1 / 30.0);
}
```

The first statement initializes the *graphics* field. In the second statement, *Content* is a property of *Game* of type *ContentManager*, and *RootDirectory* is a property of that class. Setting this property to “Content” is consistent with the Content directory that is currently storing the 14-point Kootenay font. The third statement sets a time for the program’s game loop, which governs the pace at which the program updates the video display.

After Game1 is instantiated, the phone calls a method called *Run* on the *Game1* instance, and the base *Game* class initiates the process of starting up the game. One of the first steps is a call to the *Initialize* method, which a *Game* derivative can override. XNA Game Studio generates a skeleton method to which I won’t add anything:

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
protected override void Initialize()
{
    base.Initialize();
}
```

The *Initialize* method is not the place to load the font or other content. That comes a little later when the base class calls the *LoadContent* method.

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Rectangle clientBounds = this.Window.ClientBounds;
    Vector2 textSize = kootenay14.MeasureString(text);
    textPosition =
        new Vector2((int)(clientBounds.X + (clientBounds.Width - textSize.X) / 2),
                    (int)(clientBounds.Y + (clientBounds.Height - textSize.Y) / 2));
}
```

The first statement in this method is provided for you. You'll see shortly how this *spriteBatch* object is used to shoot sprites out to the display.

The other statements are ones I've added, and you'll notice I tend to preface property names like *Content* and *Window* with the keyword *this* to remind myself that they're properties and not a static class. As I've already mentioned, the *Content* property is of type *ContentManager*. The generic *Load* method allows loading content into the program, in this case of type *SpriteFont*. The name in quotation marks is the Asset Name as indicated in the content's properties. This statement stores the result in the *kootenay14* field of type *SpriteFont*.

The third statement requires a little bit of explanation: To center text on the screen, the program needs to know the size of the screen. Commonly for this purpose, XNA programs use the *Viewport* property of the *GraphicsDevice* class, which is accessible through the *GraphicsDevice* property of *Game*. This *Viewport* object provides *Width* and *Height* properties of the screen.

However, on the Windows Phone 7 Series, the top 32 pixels of the display are commonly occupied by the System Tray, so a slightly different strategy is required. You have two choices:

If you really want to use the entire screen and make the System Tray invisible to the user, you are allowed to do so. In the constructor of the *Game1* class, insert the following statement after the *GraphicsDeviceManager* has been instantiated:

```
graphics.IsFullScreen = true;
```

You can then use *Viewport* to obtain the width and height of the full screen.

The second option keeps the System Tray alive. Rather than getting the *Viewport* property of the *GraphicsDevice* class, get the *ClientBounds* property of the *GameWindow* class, which is accessible through the *Window* property of *Game*. This *ClientBounds* property is of type *Rectangle*:

```
Rectangle clientBounds = this.Window.ClientBounds;
```

This *Rectangle* structure includes *Width* and *Height* properties but these indicate the area of the screen occupied by the program. In addition, for full generality, the *Left* and *X* provide the leftmost horizontal coordinate, and *Top* and *Y* provide the topmost. You can also make use of the *Right* and *Bottom* properties. (This is the way it's supposed to work, anyway. ClientBounds does not seem to be working quite correctly in early versions of the Windows Phone 7 Series.)

You also need the size of the text you're displaying. The *SpriteFont* class has a very handy method named *MeasureString* that returns a *Vector2* object with the size of the text in pixels. It is then straightforward to calculate *textPosition*—the point relative to the upper-left corner of the viewport where the upper-left corner of the text string is to be displayed—to enter the text on the program's area of the screen. Notice that I cast the result of the calculation to integers. Because fonts are converted to bitmaps for use by the XNA program, it is best to display them at integer boundaries.

The initialization phase of the program has now concluded, and the real action begins. The program enters the *game loop*. In synchronization with the 30 Hz refresh rate of the video display, two methods in your program are called: *Update* followed by *Draw*. Back and forth: *Update*, *Draw*, *Update*, *Draw*, *Update*, *Draw*…. (It's actually somewhat more complicated than this if the *Update* method requires more than 1/30th of a second to complete, but I'll discuss these timing issues in more detail in a later chapter.)

In the *Draw* method you want to draw on the display. But that's *all* you want to do. If you need to perform some calculations in preparation for drawing, you should do those in the *Update* method. The *Update* method prepares the program for the *Draw* method. Very often an XNA program will be moving sprites around the display based on user input. For the phone, this user input involves fingers touching the screen. All handling of user input should also occur during the *Update* method. You'll see an example later in this chapter.

You should write your *Update* and *Draw* methods so that they execute as quickly as possible. That's rather obvious, I guess, but here's something very important that might not be so obvious: You should avoid code in *Update* and *Draw* that routinely allocates memory from the program's local heap. Eventually the .NET garbage collector will want to reclaim some of this memory and while the garbage collector is doing its job, your game might stutter a bit. Throughout the chapters on XNA programming, you'll see techniques to avoid allocating memory from the heap.

Your *Draw* methods probably won't contain any questionable code; it's usually in the *Update* method where trouble lurks. Avoid any *new* expressions involving classes. These always cause memory allocation. Instantiating a structure is fine, however, because structure instances are stored on the stack and not in the heap. (XNA uses structures rather than classes for many types of objects you'll often use in *Update*.) But heap allocations can also occur without explicit *new* expressions. For example, concatenating two strings creates another string on the heap. If you need to perform string manipulation in *Update*, you should use *StringBuilder*. In fact, XNA provides methods for display text using *StringBuilder* objects.

In XnaHelloPhone, however, the *Update* method is trivial. The text displayed by the program is anchored in one spot. All the necessary calculations have already been performed in the *LoadContent* method. For that reason, the *Update* method will be left simply as it was originally created:

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    base.Update(gameTime);
}
```

The default code uses the static *GamePad* class to check if the Back button has been pressed and uses that to exit the game.

Finally, there is the *Draw* method. The version created for you simply colors the background with a light blue:

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

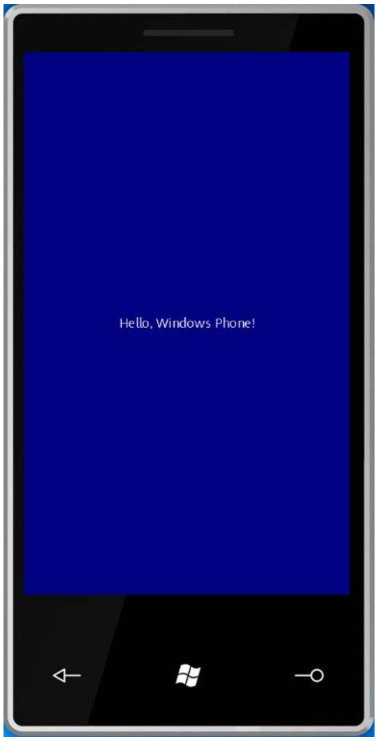
```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}
```

When you're developing an XNA program for the phone or the phone emulator, the appearance of the light blue screen is very comforting because it means the program is actually working. But it's not good color for conserving power. In my revised version, I've compromised by setting the background to a darker blue. As in Silverlight, XNA supports the 140 colors come to be regarded as standard. The text is colored white:

XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, text, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Sprites get out on the display by being bundled into a *SpriteBatch* object, which was created during the call to *LoadContent*. Between calls to *Begin* and *End* there can be multiple calls to *DrawString* to draw text and *Draw* to draw bitmaps. Those are the only options. This particular *DrawString* call references the font, the text to display, the position of the upper-left corner of the text relative to the upper-left corner of the screen, and it specifies the color. And here it is:



Modes of Drawing

You can convince yourself that *Draw* is actually being called 30 times per second with a little additional code. Define a field like so:

```
bool drawTheScreen;
```

Change the *Draw* method to only draw on the screen when *drawTheScreen* is true, and then toggle the value on each call:

```
if (drawTheScreen)
{
    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, text, textPosition, Color.White);
    spriteBatch.End();
}

drawTheScreen ^= true;
```

You'll see the text flicker as it's drawn only during every other frame.

Reflect for a moment how different both Silverlight and XNA are from older Windows programming environments. In Windows API programming, or MFC programming, or even Windows Forms programming, your program draws "on demand," that is, when an area of your window is invalid and needs to be repainted or when your program deliberately invalidates an area of your window to force painting. This may seem similar to the XNA *Draw* method, but it's really not. A conventional Windows program draws much less frequently, and what it does draw goes straight out to the video display where it remains until overwritten.

A Silverlight program often doesn't seem to draw at all! Deep inside of Silverlight is a visual composition layer that operates in a retained graphics mode and organizes all the visual elements into a composite whole. Elements such as *TextBlock* exist as actual entities inside this composition layer. At some point, *TextBlock* is rendering itself, but what it renders is retained along with the rendered output of all the other elements in the visual tree.

In contrast, an XNA program is actively drawing during every frame of the video display. This is conceptually different from older Windows programming environments as well as Silverlight. It is very powerful, but I'm sure you know well what must also come with great power.

In a program such as XnaHelloPhone, the text stays firmly in place. The program really only needs one call to *Draw* to render the display. To conserve power, it is possible for the *Update* method to call the *SuppressDraw* method defined by the *Game* class to inhibit a corresponding call to *Draw*. The *Update* method will still be called 30 times per second because it needs to check for user input, but if the code in *Update* calls *SuppressDraw*, *Draw* won't be called during that cycle of the game loop. If the code in *Update* doesn't call *SuppressDraw*, *Draw* will be called.

I guess it's probably obvious that you just can't drop a naked *SuppressDraw* call into *Update*. That will prevent *Draw* from *ever* being called, and you'll get nothing on the screen. Instead, you probably want to define a Boolean field:

```
bool requiresDraw = true;
```

In *Update*, call the following code:

```
if (!requiresDraw)
    SuppressDraw();
```

In *Draw* set the field to *false*:

```
requiresDraw = false;
```

I'll be using this technique—or one similar to it—to suppress unnecessary calls to *Draw* in programs where there might not be movement every frame. The *Update* method can then set *requiresDraw* to *true* if anything changes that needs to be updated on the screen.

Using Touch in XNA

Let's write a little variation of the first XNA program. This second project is called XnaTapHello. It's much like the first program except when you tap the text with your finger, it changes to a random color, and when you tap the screen outside the text area, the color changes back to white.

As you'll see before the end of this chapter, in a Silverlight program, user input is reported through events. In XNA, the program obtains the current state of user input through polling. One of the purposes of the *Update* method is to check the state of user input and make changes that affect what goes out to the screen during the *Draw* method.

The multitouch input device is referred to in XNA as a *touch panel*, and you use methods in the static *TouchPanel* class to obtain this input. It is possible (although not necessary) to obtain information about the multitouch device itself by calling the static *TouchPanel.GetCapabilities* method. The *TouchPanelCapabilities* object returned from this method has three properties:

- *IsConnected* is *true* if the touch panel is available. For the phone, this will always be *true*.
- *MaximumTouchCount* returns the number of touch points, at least 4 for the phone.
- *HasPressure* indicates if pressure information is available. For the phone, this will always be *true*.

For most purposes, you just need to use the other static method in *TouchPanel*. This one's essential, and if your phone program uses touch input—which is highly likely unless you're writing a game that completely ignores the game player—you'll be calling this method during every call to *Update* after program initialization:

```
TouchCollection touchLocations = TouchPanel.GetState();
```

The *TouchCollection* is a collection of zero or more *TouchLocation* objects. *TouchLocation* has four properties:

- *State* is a member of the *TouchLocationState* enumeration: *Pressed*, *Moved*, *Released*.
- *Position* is a Vector2 indicating the finger position relative to the upper-left corner of the display.
- *Pressure* is a float between 0 and 1.
- *Id* is an integer identifying a particular finger from *Pressed* through *Released*.

If no fingers are touching the screen, the *TouchCollection* will be empty. When a finger first touches the screen, *TouchCollection* contains a single *TouchLocation* object with *State* equal to *Pressed*. On subsequent calls to *TouchPanel.GetState*, the *TouchLocation* object will have *State* equal to *Moved* even if the finger has not physically moved. When the finger is lifted from the screen, the *State* property of the *TouchLocation* object will equal *Released*. On subsequent calls to *TouchPanel.GetState*, the *TouchCollection* will be empty.

One exception: If the finger is tapped and released on the screen very quickly—that is, within a 1/30th of a second—it's possible that the *TouchLocation* object with *State* equal to *Pressed* will be followed with *State* equal to *Released* with no *Moved* states in between.

That's just one finger touching the screen and lifting. In the general case, multiple fingers will be touching, moving, and lifting from the screen independently of each other. You can track particular fingers using the *Id* property. For any particular finger, that *Id* will be the same from *Pressed*, through all the *Move* values, to *Released*.

TouchLocation also has a very handy method called *TryGetPreviousLocation*, which you call like this:

```
TouchLocation previousTouchLocation;
bool success = touchLocation.TryGetPreviousLocation(out previousTouchLocation);
```

Almost always, you will call this method when *touchLocation.State* is *Moved* because you can then obtain the previous location and calculate a difference in location. If *touchLocation.State* equals *Pressed*, then *TryGetPreviousLocation* will return *false* and *previousTouchLocation.State* will equal the enumeration member *TouchLocationState.Invalid*. You'll also get these results if you use the method on a *TouchLocation,* which itself was returned from *TryGetPreviousLocation*.

The program I've proposed changes the text color when the user taps the text string, so the processing of *TouchPanel.GetStates* will be relatively simple. The program will be looking only at *State* values of *Pressed*.

This project is called XnaTapHello. Like the first project, it needs a font, which I'll assume is the same. A few more fields are required:

XNA Project: XnaTapHello   File: Game1.cs (excerpt showing fields)

```
namespace XnaTapHello
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        Random rand = new Random();
        string text = "Hello, Windows Phone!";
        SpriteFont kootenay14;
        Vector2 textSize;
        Vector2 textPosition;
        Color textColor = Color.White;
        bool requiresDraw = true;
        …
    }
}
```

The *LoadContent* method is similar to the version in the first program except that *textSize* is saved as a field because it needs to be accessed in later calculations:

XNA Project: XnaTapHello   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Rectangle clientBounds = this.Window.ClientBounds;
    textSize = kootenay14.MeasureString(text);
    textPosition =
        new Vector2((int)(clientBounds.X + (clientBounds.Width - textSize.X) / 2),
                    (int)(clientBounds.Y + (clientBounds.Height - textSize.Y) / 2));
}
```

As is typical with XNA programs, much of the "action" occurs in the *Update* method. The method calls *TouchPanel.GetStates* and then loops through the collection of *TouchLocation* objects to find only those with *State* equal to *Pressed*. If the *Position* is inside the rectangle occupied by the text string, the *textColor* field is set to a random RGB color value using one of the constructors of the *Color* structure. Otherwise, *textColor* is set to *Color.White*.

XNA Project: XnaTapHello   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    TouchCollection touchLocations = TouchPanel.GetState();

    foreach (TouchLocation touchLocation in touchLocations)
    {
        if (touchLocation.State == TouchLocationState.Pressed)
        {
            Vector2 touchPosition = touchLocation.Position;

            if (touchPosition.X >= textPosition.X &&
                touchPosition.X < textPosition.X + textSize.X &&
                touchPosition.Y >= textPosition.Y &&
                touchPosition.Y < textPosition.Y + textSize.Y)
            {
                textColor = new Color((byte)rand.Next(256),
                                      (byte)rand.Next(256),
                                      (byte)rand.Next(256));
            }
            else
            {
                textColor = Color.White;
            }
            requiresDraw = true;
        }
    }

    if (!requiresDraw)
        SuppressDraw();

    base.Update(gameTime);
}
```

Notice the use of the *requiresDraw* field. It's set to *true* whenever the *textColor* changes, but if it's *false*, *SuppressDraw* is called. The field is set to *true* in the *Draw* method, which by this point shouldn't hold any surprises:

XNA Project: XnaTapHello   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    this.GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, text, textPosition, textColor);
    spriteBatch.End();
    requiresDraw = false;

    base.Draw(gameTime);
}
```

As you play around with this program in the emulator, you'll discover a latency problem. The program takes about a second to respond to touch so be a little patient. If you experiment with a real device, you'll also discover that touch is not quite as deterministic as you might like. The text string is a small target and often you'll tap outside the string. Even when you tap with a single finger, the finger might touch the screen in more than one place. In some cases, the same *foreach* loop in *Update* might set *textColor* more than once!

Handling multitouch input is often challenging, and it's one of the challenges this book will courageously tackle.

Touch Events in Silverlight

If you've experimented with multitouch in the nonphone version of Silverlight 3, you know about the *Touch.FrameReported* event. In Windows Phone 7 Series programming, that event has become obsolete.

Instead, you'll be using a much more sophisticated array of four events: *ManipulationStarted*, *ManipulationInertiaStarting*, *ManipulationDelta*, and *ManipulationCompleted*. These events are so sophisticated in incorporating concepts of velocity and inertia that they really acquire a chapter of their own. In this chapter I'm going to stick with *ManipulationStarted* just to detect contact of a finger on the screen, and I won't bother with what the finger does after that.

The XnaTapHello program needed to arithmetically determine if the coordinates of the finger touching the screen was within the rectangle occupied by the displayed text. In the Silverlight program, that calculation won't be necessary. Elements such as *TextBlock* are actual entities with specific locations on the screen, and these can be automatically hit-tested—that is, determined if they are underneath a particular user-input event such as a finger touch.

Let's see how this works in a project called SilverlightTapHello1. In the definition of the *TextBlock* in the XAML file, I've added a line that associates the *ManipulationStarted* event with the *OnTextBlockManipulationStarted* event handler:

Silverlight Project: SilverlightTapHello1   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

The MainPage.xaml.cs code-behind file creates an instance of the *Random* class as a field and contains the event handler. This is the complete file except for the *using* directives:

Silverlight Project: SilverlightTapHello1   File: MainPage.xaml.cs (excerpt)

```
namespace SilverlightTapHello1
{
    public partial class MainPage : PhoneApplicationPage
    {
        Random rand = new Random();

        public MainPage()
        {
            InitializeComponent();
            SupportedOrientations = SupportedPageOrientation.Portrait |
                                    SupportedPageOrientation.Landscape;
        }

        void OnTextBlockManipulationStarted(object sender,
                                            ManipulationStartedEventArgs args)
        {
            TextBlock txtblk = sender as TextBlock;

            Color clr = Color.FromArgb(255, (byte)rand.Next(256),
                                            (byte)rand.Next(256),
                                            (byte)rand.Next(256));

            txtblk.Foreground = new SolidColorBrush(clr);
        }
    }
}
```

If you let Visual Studio 2010 Express for Windows Phone create the event handler for you when you type in the name of the event in the XAML file, it will create a name like *TextBlock_ManipulationStarted*. I don't like underlines in identifiers, and I like event handlers to begin with the word *On*, so I usually just rename the handler in the C# file, and Visual Studio's's intelligent renaming will also rename it in the XAML file. The code that Visual StudioE generates uses the name *e* for the event arguments; I usually change that to *args*.

Because this event was set on the *TextBlock*, the event handler is called only when the user touches the *TextBlock*. Within the event handler, the *sender* argument is the *TextBlock* object and it can be safely cast to a *TextBlock* for accessing the object. Unlike the *Color* structure in XNA, the Silverlight *Color* structure doesn't have a constructor to set a color from red, green, and blue values, but it does have a static *FromArgb* method that creates a *Color* object based on alpha, red, green, and blue values, where alpha is opacity. Set the alpha channel to 255 to get an opaque color. Although it's not obvious at all in the XAML files, the *Foreground* property is actually of type *Brush*, an abstract class from which *SolidColorBrush* descends.

If you run this program, you'll see that it works but only partially. If you touch the *TextBlock*, you'll indeed change the text to a random color. But if you touch outside the *TextBlock*, the text does *not* go back to white. The problem is that we're looking at *ManipulationStarted* events only when the user touches the *TextBlock*!

A program that functions correctly according to my original specification needs to get touch event occurring *anywhere* on the page. A handler for the *ManipulationStarted* event needs to be installed on *MainPage* rather than just on the *TextBlock*.

Although that's certainly possible, there's actually an easier way. The *UIElement* class—from which *FrameworkElement*, *Control*, *UserControl*, *Page*, *PhoneApplicationPage*, and *MainPage* successively derive—not only defines all the *Manipulation* events but also exposes protected virtual methods corresponding to those events. You don't need to install a handler for the *ManipulationStarted* event on *MainPage*; instead you can override the *OnManipulationStarted* virtual method.

This approach is implemented in the SilverlightTapHello2 project. The XAML file doesn't refer to any events but gives the *TextBlock* a name so that it can be referred to in code:

Silverlight Project: SilverlightTapHello2   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Although *Name* is a property of *TextBlock*, the property plays a very special role. If you compile the program at this point and look at the MainPage.g.cs—the code file that the compiler generates based on the MainPage.xaml file, you'll see a bunch of fields in the *MainPage* class:

```
internal System.Windows.Controls.Grid LayoutRoot;
internal System.Windows.Controls.Grid TitleGrid;
internal System.Windows.Controls.TextBlock textBlockPageTitle;
internal System.Windows.Controls.TextBlock textBlockListTitle;
internal System.Windows.Controls.Grid ContentGrid;
internal System.Windows.Controls.TextBlock txtblk;
```

These are all names assigned to elements in the XAML file. These names become fields in the generated partial *MainPage* class. These fields are assigned from code in the *InitializeComponent* method also in MainPage.g.cs:

```
this.LayoutRoot = ((System.Windows.Controls.Grid)(this.FindName("LayoutRoot")));
this.TitleGrid = ((System.Windows.Controls.Grid)(this.FindName("TitleGrid")));
this.textBlockPageTitle =
((System.Windows.Controls.TextBlock)(this.FindName("textBlockPageTitle")));
this.textBlockListTitle =
((System.Windows.Controls.TextBlock)(this.FindName("textBlockListTitle")));
this.ContentGrid = ((System.Windows.Controls.Grid)(this.FindName("ContentGrid")));
this.txtblk = ((System.Windows.Controls.TextBlock)(this.FindName("txtblk")));
```

This means that anytime after the constructor in MainPage.xaml.cs calls *InitializeComponent*, any code in the class can reference those *Grid* and *TextBlock* elements in the XAML file. This is one of the two primary ways in which code and XAML interact. You've already seen the other, where an element in XAML fires an event handled in code.

You may be curious why Visual Studio 2010 Express for Windows Phone creates XAML files where *Grid* and TextBlock are assigned names using the *x:Name* attribute and I gave the *TextBlock* a name using *Name*. All classes that derive from *FrameworkElement* (which is the sole class that inherits from *UIElement*) have a *Name* property. But if *Name* is not available, XAML itself supports a *Name* property that you reference as *x:Name*. Some programmers don't like the idea of using *Name* for some objects in XAML but *x:Name* for others, so they have chosen to use *x:Name* consistently. I use *Name* when it's available. But they're really functionally identical.

In the MainPage.xaml.cs file, the *MainPage* class has the same single field as the previous program but overrides the *OnManipulationStarted* method:

Silverlight Project: SilverlightTapHello1   File: MainPage.xaml.cs (excerpt)

```
namespace SilverlightTapHello2
{
    public partial class MainPage : PhoneApplicationPage
    {
        Random rand = new Random();

        public MainPage()
        {
            InitializeComponent();
            SupportedOrientations = SupportedPageOrientation.Portrait |
                                    SupportedPageOrientation.Landscape;
        }

        protected override void OnManipulationStarted(
                                        ManipulationStartedEventArgs args)
        {
            Color clr = Colors.White;

            if (args.OriginalSource == txtblk)
            {
                clr = Color.FromArgb(255, (byte)rand.Next(256),
                                          (byte)rand.Next(256),
                                          (byte)rand.Next(256));
            }

            txtblk.Foreground = new SolidColorBrush(clr);
            base.OnManipulationStarted(args);
        }
    }
}
```

In the *ManipulationStartedEventArgs* a property named *OriginalSource* indicates where this event began—in other words, the topmost element that the user tapped. If this equals the *txtblk* object, the method creates a random color to set to the *Foreground* property.

Although the XNA and Silverlight programs are structured quite differently and the Silverlight program has some additional titles, you'll find the programs to be just about indistinguishable in actual use. And this is how it should be. Different frameworks are for the convenience of the programmer and shouldn't affect how the user experiences the program.

Some Odd Behavior?

In all the Silverlight programs in this chapter, I've centered the *TextBlock* within its container (the *Grid*) using these attributes:

```
HorizontalAlignment="Center"
VerticalAlignment="Center"
```

These two properties are defined by *FrameworkElement*, and they are an important component of dynamic layout in Silverlight. In the SilverlightTapHello2 program, try setting these two properties to the following values:

```
HorizontalAlignment="Left"
VerticalAlignment="Top"
```

In the designer, the *TextBlock* moves to the upper-left corner of the *Grid* that occupies the bulk of the screen. If you run the program, it still works the same: Touch the text and it changes to a random color; touch somewhere outside the text and it changes back to white.

Now delete the *HorizontalAlignment* and *VerticalAlignment* attributes entirely. The text remains positioned in the upper-left corner of the *Grid*. But now try running the program. If you touch *anywhere* within the large area below the *TextBlock*, the text will change to a random color, and only by touching the title area above the text can you change it back to white. Apparently the program has stopped working correctly.

Why do you suppose that is?

Part II
Silverlight

Chapter 3
Code and XAML

As you've seen, a Silverlight program is generally a mix of code and XAML. In general, you'll use XAML for laying out the visuals of your application, and you'll use code for event handling, including all user-input events and all events generated by controls as a result of processing user-input events.

Much of the object creation and initialization performed in XAML would traditionally be done in the constructor of a page or window class. This might make XAML seem just a tiny part of the application, but it turns out to be much more than that. As the name suggests, XAML is totally compliant XML, so it's instantly toolable—machine writable and machine readable as well as human writable and human readable.

Although XAML is usually concerned with object creation and initialization, certain features of Silverlight provide much more than object initialization would seem to imply. One of these features is data binding, which involves connections between controls, or between controls and underlying data, so that properties are automatically updated without the need for explicit event handlers. Entire animations can also be defined in XAML file.

Although XAML is sometimes referred to as a "declarative language," it is certainly not a complete programming language. You can't perform arithmetic in any generalized manner in XAML, and you can't dynamically create objects in XAML.

Experienced programmers encountering XAML for the first time are sometimes resistant to it. I know I was. Everything that we value in a programming language such as C#—such as required declarations, strong typing, array-bounds checking, tracing abilities for debugging—largely goes away when everything is reduced to XML text strings. Over the years, however, I've gotten very comfortable with XAML, and I find it very liberating in using XAML for the visuals of the application. In particular I like how the parent-child relationship of controls on the surface of a window is mimicked by the parent-child structure inherent in XML. I also like the ability to experiment with XAML—even just in the Visual Studio designer.

Everything you need to do in Silverlight can be allocated among these three categories:

- Stuff you can do in either code or XAML
- Stuff you can do only in code (e.g., event handling and methods)
- Stuff you can do only in XAML (e.g., templates)

In both code and XAML you can instantiate classes and structures, and set the properties of these objects. A class or structure instantiated in XAML must be defined as public (of course), but it must also have a parameterless constructor. When XAML instantiates the class, it has no way of passing anything to the constructor. In XAML you can associate a particular event with an event handler, but the event handler itself must be implemented in code. You can't make method calls in XAML because, again, there's no way to pass arguments to the method.

If you want, you can write pretty much your entire application in code, but there is a very important type of job that *must* be done in XAML, and this is the construction of templates. You use templates in two ways: First, to visually display data using a collection of elements and controls, and secondly, to redefine the visual appearance of a control while maintaining its functionality. You can write alternatives to templates in code, but you can't write the templates themselves.

After some experience with Silverlight programming, you might decide that you want to use a design program such as Expression Blend to generate XAML for you. But I urge you—speaking programmer to programmer—to learn how to write XAML by hand. At the very least you need to know how to read the XAML that design programs will generate for you.

One of the very nice features of XAML is that you can experiment with it in a very interactive manner, and by experimenting with XAML you can learn a lot about Silverlight. Programming tools are designed specifically for experimenting with XAML. These programs take advantage of a static method named *XamlReader.Load* that can convert XAML text into an object at runtime. Later in this book I'll show you how to use that method and you'll see a phone application that lets you experiment with XAML right on the phone!

Until then, however, you can experiment with XAML in the Visual Studio designer. Generally the designer responds promptly and accurately to changes you make in the XAML. Only when things get a bit complex will you actually need to build and deploy the application to see what it's doing.

A TextBlock in Code

Before we get immersed in experimenting with XAML, however, I must issue a warning: As you get accustomed to using XAML exclusively for certain common chores, it's important not to forget how to write C#!

You'll recall the XAML version of the *TextBlock* in the *Grid* from Chapter 2:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Elements in XAML such as *TextBlock* are actually classes. Attributes of these elements (such as *Text*, *HorizontalAlignment*, and *VerticalAlignment* are properties of the class. Let's see how easy it is to write a *TextBlock* in code, and to also use code to insert the *TextBlock* into the XAML *Grid*.

The new project is called TextBlockInCode. In the newly created MainPage.xaml file, the *Grid* has been assigned the name "ContentGrid." That name will be useful for referring to the *Grid* in code. As you saw in Chapter 2, assigning *Name* or *x:Name* causes a field to be created in the class, which you can then reference in code.

The code-behind file for *MainPage* creates a *TextBlock* in its constructor and adds that to the *Grid*:

Silverlight Project: TextBlockInCode   File: MainPage.xaml.cs (excerpt)

```
namespace TextBlockInCode
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
            SupportedOrientations = SupportedPageOrientation.Portrait |
                                    SupportedPageOrientation.Landscape;
            TextBlock txtblk = new TextBlock();
            txtblk.Text = "Hello, Windows Phone!";
            txtblk.HorizontalAlignment = HorizontalAlignment.Center;
            txtblk.VerticalAlignment = VerticalAlignment.Center;

            ContentGrid.Children.Add(txtblk);
        }
    }
}
```

The constructor first creates the *TextBlock*, then sets its properties, and finally adds the *TextBlock* to *ContentGrid*. You don't need to perform the steps precisely in this order: You can add the *TextBlock* to *ContentGrid* first and then set the *TextBlock* properties.

You can also take advantage of a feature introduced in C# 3.0 to instantiate a class and define its properties in a block:

```
TextBlock txtblk = new TextBlock
{
    Text = "Hello, Windows Phone!",
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
clientGrid.Children.Add(txtblk);
```

That makes the code look a *little* more like the XAML, but you can still see that XAML provides several shortcuts. The *HorizontalAlignment* and *VerticalAlignment* properties must be set to members of the *HorizontalAlignment* and *VerticalAlignment* enumerations, respectively. In XAML, you need only specify the member name.

Just looking at the XAML, it is not so obvious that the *Grid* has a property named *Children*, and that this property is a collection, and nesting the *TextBlock* inside the *Grid* effectively adds the *TextBlock* to the *Children* collection. The process of adding the *TextBlock* to the *Grid* must be more explicit in code.

A XAML-less Silverlight Program?

Sometimes it's interesting to see how much you can actually delete from a program and still persuade it to run. In the TextBlockInCode project, you can delete everything in MainPage.xaml from the first *Grid* tag to the last *Grid* tag, leaving only the *PhoneApplicationPage* start and end tags. In the MainPage.xaml.cs code file you can delete

```
ContentGrid.Children.Add(txtblk);
```

and replace it with:

```
this.Content = txtblk;
```

This code sets the *Content* property of *PhoneApplicationPage* to the *txtblk*. The program will still compile and run except there will be no title information displayed on the page.

One difference between the *Grid* and the *PhoneApplicationPage* is that the *Grid* can have multiple children in its *Children* collection while the *PhoneApplicationPage* can only support a single child with its *Content* property, although that single child can contain nested children. In the standard MainPage.xaml file created by Visual Studio, the *Content* property of *PhoneApplicationPage* is set to the outermost *Grid*.

If you're in the adventurous mode, you can continue deleting XAML in the program until you're down to no XAML at all. The AllCodeSilverlightApp project has no XAML files. The MainPage.cs file defines a *MainPage* class that derives from *PhoneApplicationPage*. The constructor instantiates the *TextBlock* and sets it to its *Content* property:

Silverlight Project: AllCodeSilverlightApp   File: MainPage.cs (complete)

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using Microsoft.Phone.Controls;

namespace AllCodeSilverlightApp
{
    public class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            TextBlock txtblk = new TextBlock
            {
                Text = "Hello, Windows Phone!",
                Foreground = new SolidColorBrush(Colors.White),
                FontSize = 24,
                HorizontalAlignment = HorizontalAlignment.Center,
                VerticalAlignment = VerticalAlignment.Center
            };
            this.Content = txtblk;
        }
    }
}
```

Normally the MainPage.xaml file provides appropriate values of a *Foreground* color and a *FontSize* property; these now have to be provided explicitly. The *Foreground* property is of type *Brush* and in code you need to explicitly create a *SolidColorBrush* object based on the static *Colors.White* property. (The *Colors* class contains only 14 of the 141 possible colors your can reference in XAML.)

The *App* class derives from *Application*, as usual. This constructor instantiates a *PhoneApplicationFrame* and a *MainPage*. The frame is set to the *RootVisual* property of *App*, and the page is set to the *Content* property of the frame:

Silverlight Project: AllCodeSilverlightApp   File: MainPage.cs (complete)

```
using System.Windows;
using Microsoft.Phone.Controls;

namespace AllCodeSilverlightApp
{
    public class App : Application
    {
        public App()
        {
            PhoneApplicationFrame frame = new PhoneApplicationFrame();
            this.RootVisual = frame;

            MainPage page = new MainPage();
            frame.Content = page;
        }
    }
}
```

I wanted to implement this entire program in the constructor of the *App* class except that the *Content* property of *PhoneApplicationPage* is protected. For that reason, *PhoneApplicationPage* needs to be derived from instead of just instantiated.

Property Inheritance

To experiment with some XAML, it's convenient to create a project specifically for that purpose. Let's call the project XamlExperiment, and put a *TextBlock* in the last *Grid* in MainPage.xaml:

Silverlight Project: XamlExperiment   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!" />
</Grid>
```

The text shows up in the upper-left corner of the page's client area. Let's make the text italic. You can do that by setting the *FontStyle* property in the *TextBlock*:

```
<TextBlock Text="Hello, Windows Phone!"
           FontStyle="Italic" />
```

Alternatively, you can put that *FontStyle* attribute in the *PhoneApplicationPage* tag:

```
<phoneNavigation:PhoneApplicationPage FontStyle="Italic"
```

This *FontStyle* attribute can actually go anywhere in the *PhoneApplicationPage* tag. Notice that setting it in this tag also affects the *TextBlock* down at the bottom of the page. This is a feature known as *property inheritance*. Certain properties—not many more than *Foreground* and the font-related properties *FontFamily*, *FontSize*, *FontStyle*, *FontWeight*, and *FontStretch*—propagate through the visual tree. This is how the *TextBlock* gets the *FontFamily*, *FontSize*, and *Foreground* properties set on the *PhoneApplicationPage*.

You can visualize property inheritance beginning at the *PhoneApplicationPage* object. The *FontStyle* is set on that object and then it's inherited by the outermost *Grid*, and then the inner *Grid* objects, and finally by the *TextBlock*. This is a good theory. The problem with this theory is that *Grid* doesn't have a *FontStyle* property! If you try setting *FontStyle* in a *Grid* element, Visual Studio will tell you what's wrong with an error message. Property inheritance is somewhat more sophisticated than a simple handing off from parent to child, and it is one of the features of Silverlight that is intimately connected with the role of *dependency properties*, which you'll learn about in the Infrastructure chapter.

While keeping the *FontStyle* property setting to *Italic* in the *PhoneApplicationPage* tag, add a *FontStyle* setting to the *TextBlock*:

```
<TextBlock Text="Hello, Windows Phone!"
           FontStyle="Normal" />
```

Now the text goes back to normal. Obviously the *FontStyle* setting on the *TextBlock*—which is referred to as a *local value* or a *local setting*—has precedence over property inheritance. A little reflection will convince you that this behavior is as it should be. Both property inheritance and the local setting have precedence over the default value. We can express this relationship in a simple chart:

Local Settings have precedence over

Property Inheritance, which has precedence over

Default Values

This chart will grow in size as we examine all the ways in which properties can be set.

Alignment and Margins

I mentioned that you can put multiple children in a *Grid.* Generally you'll put these children in specific rows and columns of the *Grid* (as you'll see in a later chapter), but you can also put multiple children in the same *Grid* cell. In the standard MainPage.xaml file, the outermost *Grid* has two rows, but both inner *Grid* elements only have one cell, and the first contains the two *TextBlock* elements that display the titles.

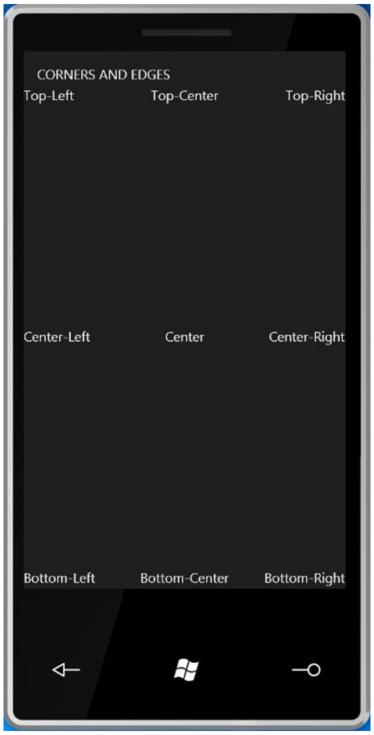
The CornersAndEdges project fills up the client-area *Grid* with nine *TextBlock* elements to demonstrate the use of *HorizontalAlignment* and *VerticalAlignment*. The *Text* properties identify the alignment settings:

Silverlight Project: CornersAndEdges   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Top-Left"
               VerticalAlignment="Top"
               HorizontalAlignment="Left" />

    <TextBlock Text="Top-Center"
               VerticalAlignment="Top"
               HorizontalAlignment="Center" />

    <TextBlock Text="Top-Right"
```

```
               VerticalAlignment="Top"
               HorizontalAlignment="Right" />

    <TextBlock Text="Center-Left"
               VerticalAlignment="Center"
               HorizontalAlignment="Left" />

    <TextBlock Text="Center"
               VerticalAlignment="Center"
               HorizontalAlignment="Center" />

    <TextBlock Text="Center-Right"
               VerticalAlignment="Center"
               HorizontalAlignment="Right" />

    <TextBlock Text="Bottom-Left"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Left" />

    <TextBlock Text="Bottom-Center"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Center" />

    <TextBlock Text="Bottom-Right"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Right" />
</Grid>
```

And here’s what it looks like:



At the very end of Chapter 2, I suggested removing *HorizontalAlignment* and *VerticalAlignment* from the *TextBlock* to see what happened. The *TextBlock* was positioned in the upper-left corner of the client area, but an experiment with touch seemed to suggest that the *TextBlock* was actually occupying the entire area of that inner *Grid*.

And that's precisely the case. The defaults for *HorizontalAlignment* and *VerticalAlignment* aren't *Left* and *Top*. The defaults are called *Stretch*. What this means is that by default the *TextBlock* fills the interior of its container, although obviously the text itself doesn't. If *TextBlock* had a *Background* property, you could easily verify that it fills the *Grid*; without that *Background* property a demonstration based on hit-testing was necessary.

The *HorizontalAlignment* and *VerticalAlignment* properties are very important in the layout system in Silverlight. So is *Margin*. Back in XamlExperiment, you can try putting the *TextBlock* in the upper-left corner but also assign the *Margin* property:

```
<TextBlock Text="Hello, Windows Phone!"
           HorizontalAlignment="Left"
```

```
                   VerticalAlignment="Top"
                   Margin="100" />
```

Now there’s a 100-pixel breathing room between the *TextBlock* and the left and top edges of the client area. The *Margin* property is of type *Thickness*, a structure that has four properties named *Left*, *Top*, *Right*, and *Bottom*. If you only specify one number in XAML, that’s used for all four sides. You can also specify two numbers like this:

```
Margin="100 200"
```

The first applies to the left and right; the second to the top and bottom. With four numbers

```
Margin="100 200 50 300"
```

they're in the order left, top, right, and bottom. Watch out: If the margins are too large, the text or parts of the text will disappear. Silverlight preserves those margins even at the expense of truncating the element.

If you set both *HorizontalAlignment* and *VerticalAlignment* to *Center*, and set *Margin* to four different numbers, you’ll notice that the text is no longer visually centered in the client area. Silverlight bases the centering on the size of the element including the margins.

TextBlock also has a *Padding* property:

```
        <TextBlock Text="Hello, Windows Phone!"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top"
                   Padding="100 200" />
```

Padding is also of type *Thickness*, and when used with the *TextBlock*, *Padding* is visually indistinguishable from *Margin*. But they are definitely different: *Margin* is space on the outside of the *TextBlock*; *Padding* is space inside the *TextBlock* not occupied by the text itself. If you were using *TextBlock* for touch events, it would respond to touch in the *Padding* area but not the *Margin* area.

The *Margin* property is defined by *FrameworkElement*; in real-life Silverlight programming, almost everything has a *Margin* property set to prevent the elements from being jammed up against each other. The *Padding* property is more rare; it’s defined only by *TextBlock*, *Border*, and *Control*.

You might try to use *Margin* to position multiple *TextBlock* elements within a single *Grid*. This is how the two titles in the standard MainPage.xaml file are positioned. But it's not a good idea. It can be confusing, and there are better ways of accomplishing the job.

What's crucial to realize is what we're *not* doing. We're not explicitly setting the *Width* and *Height* of the *TextBlock* like in some antique programming environment:

```
<TextBlock Text="Hello, Windows Phone!"
           HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Width="100"
           Height="50" />
```

You're second guessing the size of the *TextBlock* without knowing as much about the element as the *TextBlock* itself. In some cases, setting *Width* and *Height* is appropriate, but not here.

The *Width* and *Height* properties are of type *double*, and the default values are those special floating-point values called Not a Number or NaN. If you need to get the actual width and height of an element as it's rendered on the screen, you can access the properties named *ActualWidth* and *ActualHeight*. These values are only available when the element has been rendered. Some useful events are also available for obtaining information like this. The *Loaded* event is fired when visuals are layed out on the screen; *SizeChanged* is supported by elements to indicate when they've changed size; LayoutUpdated is useful when you want notification that a layout cycle has occurred.

Property-Element Syntax

Back in XamlExperiment, let's set the *TextBlock* attributes to these values:

```
<TextBlock Text="Hello, Windows Phone!"
           FontSize="36"
           Foreground="Red" />
```

Because this is XML, we can separate the *TextBlock* tag into a start tag and end tag with nothing in between:

```
<TextBlock Text="Hello, Windows Phone!"
           FontSize="36"
           Foreground="Red">
</TextBlock>
```

But you can also do something that will appear quite strange initially. You can remove the *FontSize* attribute from the start tag and set it like this:

```
<TextBlock Text="Hello, Windows Phone!"
           Foreground="Red">
    <TextBlock.FontSize>
        36
    </TextBlock.FontSize>
</TextBlock>
```

Now the *TextBlock* has a child element called *TextBlock.FontSize*, and within the *TextBlock.FontSize* tags is the value.

This is called *property-element* syntax, and it's an extremely important part of XAML. The introduction of property-element syntax also allows nailing down some terminology that unites .NET and XML. This single *TextBlock* element now contains three types of identifiers:

- *TextBlock* is an *object element*—a .NET object based on an XML element.
- *Text* and *Foreground* are *property attributes*—.NET properties set with XML attributes.
- *FontSize* is now a *property element*—a .NET property expressed as an XML element.

When I first saw the property-element syntax, I wondered if it was some kind of XML extension. Of course it's not. The period is a legal character for XML tags, so in terms of nested XML tags, these are perfectly legitimate. That they happen to consist of a class name and a property name is something known only to XAML parsers (machine and human alike).

One restriction, however: It is illegal for anything else to appear in a property element tag:

```
<TextBlock Text="Hello, Windows Phone!"
           Foreground="Red">
    <TextBlock.FontSize absolutely nothing else goes in here!>
        36
    </TextBlock.FontSize>
</TextBlock>
```

Also, you can't have both a property attribute and a property element for the same property, like this:

```
<TextBlock Text="Hello, Windows Phone!"
           FontSize="36"
           Foreground="Red">
    <TextBlock.FontSize>
        36
    </TextBlock.FontSize>
</TextBlock>
```

This is an error because the *FontSize* property is set twice.

If you look towards the top of MainPage.xaml, you'll see another property element:

```
<Grid.RowDefinitions>
```

RowDefinitions is a property of *Grid*. In App.xaml, you'll see a few more, including these:

```
<Application.RootVisual>
<Application.Resources>
```

Both *RootVisual* and *Resources* are properties of *Application*.

Colors and Brushes

Back in XamlExperiment, let's return the *TextBlock* to its pristine condition:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!" />
</Grid>
```

The text shows up as white because the Foreground is effectively set to white on the root element in MainPage.xaml. You can display traditional black on white (and in the process consume more power on OLED screens) if you set the *Background* property of the *Grid* and the *Foreground* property of the TextBlock:

```
<Grid x:Name="ContentGrid" Grid.Row="1" Background="White">
    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Black" />
</Grid>
```

The *Grid* has a *Background* property but no *Foreground* property. The *TextBlock* has a *Foreground* property but no *Background* property. The *Foreground* property is inheritable through the visual tree, and it may sometimes seem that the *Background* property is as well, but it is not. The default value of *Background* is *null*, which makes the background transparent.

When the background is transparent, the parent background shows through, and that makes it seem as if the property is inherited.

A *Background* property set to *null* is visually the same as a *Background* property set to *Transparent*, but the two settings affect hit-testing differently, which affects how the element responds to touch. A *Grid* with its *Background* set to the default value of *null* cannot detect touch input! If you want a *Grid* to have no background color on its own but still respond to touch, set *Background* to *Transparent*. You can also do the reverse: You can make an element with a non-*null* background unresponsive to touch by setting the *IsHitTestVisible* property to *false*.

For the next few experiments, set the *Background* of the *Grid* to *Blue* and the *Foreground* of the *TextBlock* to *Red*:

```
<Grid x:Name="ContentGrid" Background="Blue" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Besides the standard colors, you can write the color as a string of red, green, and blue one-byte hexadecimal values ranging from 00 to FF. For example:

```
Foreground="#FF0000
```

That’s also red. You can alternatively specify *four* two-digit hexadecimal numbers where the first one is an alpha value indicating transparency: The value 00 is completely transparent, FF is opaque, and values in between are partially transparent. Try this value:

```
Foreground="#80FF0000
```

The text will appear a somewhat faded magenta because the blue background shows through.

If you preface the pound sign with the letters *sc* you can use values between 0 and 1 for the red, blue, and green components:

```
Foreground="sc# 1 0 0"
```

You can also precede the three numbers with an alpha value between 0 and 1.

These two methods of specifying color are not equivalent, as you can verify by putting these two *TextBlocks* in the same *Grid*:

```
<Grid x:Name="ContentGrid" Background="Blue" Grid.Row="1">
    <TextBlock Text="RGB COLOR"
               HorizontalAlignment="Left"
               Foreground="#808080" />

    <TextBlock Text="scRGB COLOR"
               HorizontalAlignment="Right"
               Foreground="sc# 0.5 0.5 0.5" />
</Grid>
```

Both color specifications seem to suggest medium gray, except that the one on the right is much lighter than the one on the left.

The colors you get with the hexadecimal specification are probably most familiar. The one-byte values of red, green, and blue are directly proportional to the voltages sent to the pixels of the video display. Although the light intensity of video displays is not linear with respect to voltage, the human eye is not linear with respect to light intensity either. These two non-linearities cancel each other out (approximately) so the text on the left appears somewhat medium.

With the scRGB color space, you specify values between 0 and 1 that are proportional to light intensity, so the non-linearity of the human eye causes the color to seem to be off. If you really want a medium gray in scRGB you need values much lower than 0.5, such as:

```
Foreground="sc# 0.2 0.2 0.2"
```

Let's go back to one *TextBlock* in the *Grid*:

```
<Grid x:Name="ContentGrid" Background="Blue" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Just as I did earlier with the *FontSize* property, you can break the *Foreground* property out as a property element:

```
<TextBlock Text="Hello, Windows Phone!">
    <TextBlock.Foreground>
        Red
    </TextBlock.Foreground>
</TextBlock>
```

When I created a *TextBlock* in the all-code Silverlight program I had to set the *Foreground* property to an object of type *SolidColorBrush*. When you specify a *Foreground* property in

XAML, that *SolidColorBrush* is actually being created for you behind the scenes. You can also explicitly create the *SolidColorBrush* in XAML:

```
<TextBlock Text="Hello, Windows Phone!">
    <TextBlock.Foreground>
        <SolidColorBrush Color="Red" />
    </TextBlock.Foreground>
</TextBlock>
```

You can also break out the *Color* property as a property element:

```
<TextBlock Text="Hello, Windows Phone!">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                Red
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

And you can go even further:

```
<TextBlock Text="Hello, Windows Phone!">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                <Color>
                    <Color.A>
                        255
                    </Color.A>
                    <Color.R>
                        #FF
                    </Color.R>
                </Color>
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Notice that the *A* property of the *Color* structure needs to be explicitly set because the default value is 0, which means transparent.

The extensive use of property elements might not make much sense for simple colors and *SolidColorBrush*, but the technique becomes essential when you need to use XAML to set a property with a value that can't be expressed as a simple text string—for example, when you want to use a gradient brush rather than a *SolidColorBrush*.

Let’s begin with a simple solid *TextBlock* but with the *Background* property of the Grid broken out as a property element:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <SolidColorBrush Color="Blue" />
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Remove that *SolidColorBrush* and replace it with a *LinearGradientBrush*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

The *LinearGradientBrush* has a property of type *GradientStops*, so let’s add property element tags for the *GradientStops* property:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

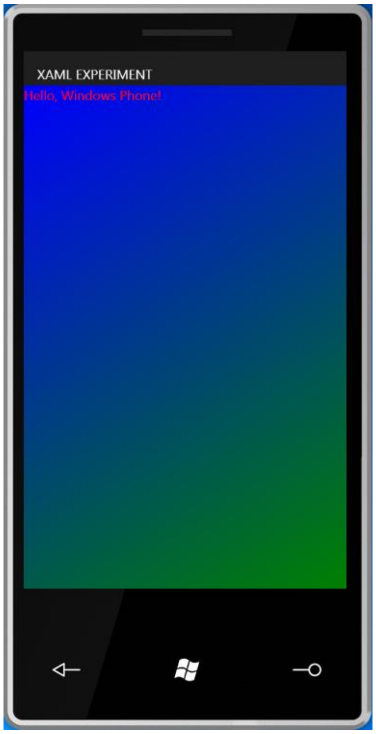
The *GradientStops* property is of type *GradientStopCollection*, so let’s add tags for that:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>
    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Now let’s put a couple *GradientStop* objects in there. The *GradientStop* has properties named *Offset* and *Color*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

And that is sufficient. This is how, with the help of property elements, you create a gradient brush in markup. It looks like this:



The *Offset* values range from 0 to 1 and they are relative to the element being colored with the brush. You can use more than two:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="0.5" Color="White" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Conceptually the brush knows the size of the area that it’s coloring and adjusts itself accordingly.

By default the gradient starts at the upper-left corner and goes to the lower-right corner, but that's only because of the default settings of the *StartPoint* and *EndPoint* properties of *LinearGradientBrush*. As the names suggest, these are coordinate points relative to the upper-left corner of the element being colored. For *StartPoint* the default value is the point (0, 0), meaning the upper-left, and for *EndPoint* (1, 1), the lower-right. If you change them to (0, 0) and (0, 1), for example, the gradient goes from top to bottom:

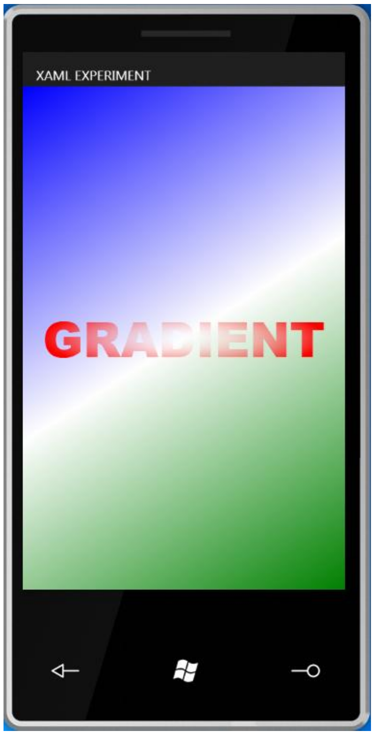
```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="0.5" Color="White" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone!"
               Foreground="Red" />
</Grid>
```

Each point is just two numbers separated by space or a comma. There are also properties that determine what happens outside the range of the lowest and highest *Offset* values if they don't go from 0 to 1.

LinearGradientBrush derives from *GradientBrush*. Another class that derives from *GradientBrush* is *RadialGradientBrush*. Here's markup for a larger *TextBlock* with a *RadialGradientBrush* set to its *Foreground* property:

```
<TextBlock Text="GRADIENT"
           FontFamily="Arial Black"
           FontSize="72"
           HorizontalAlignment="Center"
           VerticalAlignment="Center">
    <TextBlock.Foreground>
        <RadialGradientBrush>
            <RadialGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Transparent" />
                    <GradientStop Offset="1" Color="Red" />
                </GradientStopCollection>
            </RadialGradientBrush.GradientStops>
        </RadialGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

And here’s what the combination looks like:



Content and Content Properties

Everyone knows that XML can be a little “wordy.” However, the markup I’ve shown you with the gradient brushes is a little wordier than it needs to be. Let’s look at the *RadialGradientBrush* I defined for the *TextBlock*:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
            <GradientStopCollection>
                <GradientStop Offset="0" Color="Transparent" />
                <GradientStop Offset="1" Color="Red" />
            </GradientStopCollection>
        </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
</TextBlock.Foreground>
```

First, if you have at least one item in a collection, you can eliminate the tags for the collection itself. This means that the tags for the *GradientStopCollection* can be removed:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
            <GradientStop Offset="0" Color="Transparent" />
            <GradientStop Offset="1" Color="Red" />
        </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
</TextBlock.Foreground>
```

Moreover, many classes that you use in XAML have something called a *ContentProperty* attribute. This word "attribute" has different meanings in .NET and XML; here I'm talking about the .NET attribute, which refers some additional information that is associated with a class or a member of that class. If you look at the documentation for the *GradientBrush* class—the class from which both *LinearGradientBrush* and *RadialGradientBrush* derive—you'll see that the class was defined with an attribute of type *ContentPropertyAttribute*:

```
[ContentPropertyAttribute("GradientStops", true)]
public abstract class GradientBrush : Brush
```

This attribute indicates one property of the class that is assumed to be the content of that class, and for which the property-element tags are not required. For *GradientBrush* (and its descendents) that one property is *GradientStops*. This means that the *RadialGradientBrush.GradientStops* tags can be removed from the markup:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <GradientStop Offset="0" Color="Transparent" />
        <GradientStop Offset="1" Color="Red" />
    </RadialGradientBrush>
</TextBlock.Foreground>
```

Now it's not quite as wordy but it's still comprehensible. The two *GradientStop* objects are the content of the *RadialGradientBrush* class.

Earlier in this chapter I created a *TextBlock* in code and added it to the *Children* collection of the *Grid*. In XAML, we see no reference to this *Children* collection. That's because the *ContentProperty* attribute of *Panel*—the class from which *Grid* derives—defines the *Children* property as the content of the *Panel*:

```
[ContentPropertyAttribute("Children", true)]
public abstract class Panel : FrameworkElement
```

If you want to get more explicit in your markup, you can include a property element for the *Children* property:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Children>
        <TextBlock Text="Hello, Windows Phone!" />
    </Grid.Children>
</Grid>
```

Similarly, *PhoneApplicationPage* derives from *UserControl*, which also has a *ContentProperty* attribute:

```
[ContentPropertyAttribute("Content", true)]
public class UserControl : Control
```

The *ContentProperty* attribute of *UserControl* is the *Content* property. (That sentence makes more sense when you see it on the page rather than when you read it out load!)

Suppose you want to put two *TextBlock* elements in a *Grid*, and you want the *Grid* to have a *LinearGradientBrush* for its *Background*. It's legal to put the *Background* property element first within the *Grid* tags followed by the two *TextBlock* elements:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />

    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />
</Grid>
```

It's also legal to put the two *TextBlock* elements first and the *Background* property element last:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />
    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />

    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

But putting the *Background* property element between the two *TextBlock* elements simply won't work:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />

    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />
</Grid>
```

The precise problem with this syntax is revealed when you put in the missing property elements for the *Children* property of the *Grid*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Children>
        <TextBlock Text="TextBlock #1"
                   HorizontalAlignment="Left" />
    </Grid.Children>

    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>

    <Grid.Children>
        <TextBlock Text="TextBlock #2"
                   HorizontalAlignment="Right" />
    </Grid.Children>
</Grid>
```

Now it's obvious that the *Children* property is being set twice—and that's clearly illegal.

TextBlock Properties and Inlines

The *TextBlock* element has five font-related properties: *FontFamily*, *FontSize*, *FontStretch*, *FontStyle*, *FontWeight*.

In XAML you can set *FontFamily* to a string. (In code you need to create an instance of the *FontFamily* class.) The default is called "Portable User Interface". On the phone emulator, this default font seems to map to Segoe UI, a font that Microsoft uses extensively. In addition, you can set *FontFamily* to one of the following:

Arial

Arial Black

Comic Sans MS

Courier New

Georgia

Lucida Sans Unicode

Tahoma

Times New Roman

Trebuchet MS

Verdana

▶🎁🚲🛡ⓘ●■? (Webdings)

These are the only *FontFamily* names that I've discovered to be implemented on the phone. If you misspell a name that you assign to *FontFamily*, nothing bad will happen; you'll just get the default.

Judging from the App.xaml file that Visual Studio generates, there should be a whole bunch of different fonts beginning with the name Segoe WP, the WP standing for Windows Phone. Although these fonts obviously seem to be tailored to the phone, I have not been able to verify that they are actually on the phone emulator that I used to write this chapter. When the Segoe WP fonts become accessible to the phone, you can choose from a nice progression of stroke weight:

Segoe WP Black

Segoe WP Bold

Segoe WP Semibold

Segoe WP

Segoe WP SemiLight

Segoe WP Light

Until that time, I also remain curious how the *FontWeight* property will interact with *FontFamily* selection. You can set *FontWeight* to one of the static members of the *FontWeights* (notice the plural) class. These members return instance of the *FontWeight* (singular) structure. Commonly you set *FontWeight* to either *Normal* or *Bold*, and boldface is synthesized if there's not a specific boldface font of that family available. But *FontWeights* also includes members named *Black*, *SemiBold*, and *Light* (but not *SemiLight*).

As you saw earlier, you can set *FontType* to either *Normal* or *Italic*. In theory, you can set *FontStretch* to values such as *Condensed* and *Expanded* but I've never seen it work in Silverlight. *TextBlock* also has a *TextDecorations* property. Although this property seems to be very generalized, in Silverlight there is only one option:

```
TextDecorations="Underline"
```

The *TextBlock* property I've used most, of course, is *Text* itself. The string you set to the *Text* property can include embedded Unicode characters in the standard XML format, for example:

```
Text="&#x03C0; is approximately 3.14159"
```

If the *Text* property is set to a very long string, you might not be able to see all of it. You can insert the codes for carriage return or line feed characters (&#x000D; or &#x000A;) or you can set

```
TextWrapping="Wrap"
```

and *TextAlignment* to *Left*, *Right*, or *Center* (but not *Justify*). You can also set the text as a content of the *TextBlock* element:

```
<TextBlock>
    This is some text.
</TextBlock>
```

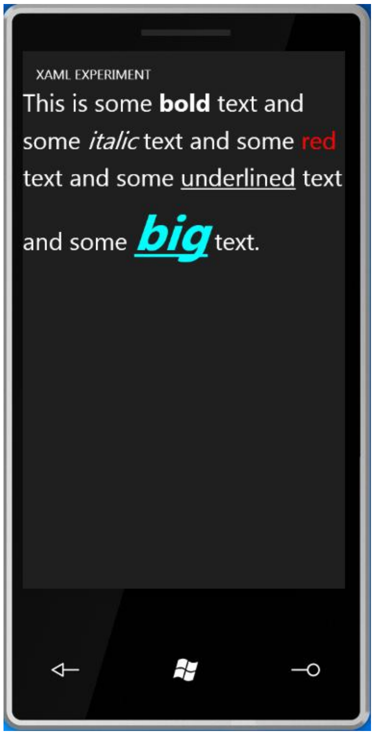
However, you might be surprised to learn that the *ContentProperty* attribute of *TextBlock* is not the *Text* property but another property named *Inlines*. This property is of type *InlineCollection*—a collection of objects of type *Inline*, namely *LineBreak* and *Run*. These make *TextBlock* much more versatile. The use of *LineBreak* is simple:

```
<TextBlock>
    This is some text<LineBreak />This is some more text.
</TextBlock>
```

Run is interesting because it too has *FontFamily*, *FontSize*, *FontStretch*, *FontStyle*, *FontWeight*, *Foreground* and *TextDecorations* properties, so you can make your text very fancy:

```
<TextBlock FontSize="36"
           TextWrapping="Wrap">
    This is
    some <Run FontWeight="Bold">bold</Run> text and
    some <Run FontStyle="Italic">italic</Run> text and
    some <Run Foreground="Red">red</Run> text and
    some <Run TextDecorations="Underline">underlined</Run> text
    and some <Run FontWeight="Bold"
                  FontStyle="Italic"
                  Foreground="Cyan"
                  FontSize="72"
                  TextDecorations="Underline">big</Run> text.
</TextBlock>
```

In the Visual Studio design view, you might see the text within the *Run* tags not properly separated from the text outside the *Run* tags. This is an error. When you actually run the program in the emulator, it looks fine:



This is some **bold** text and some *italic* text and some red text and some underlined text and some ***big*** text.

These are vector-based TrueType fonts, and the actual vectors are scaled to the desired font size before the font characters are rasterized, so regardless how big the characters get, they still seem smooth.

Although you might think of a *TextBlock* as sufficient for a paragraph of text, it doesn't provide all the features that a proper *Paragraph* class provides, such as first-line text indenting or a hanging first line where the rest of the paragraph is indented. I don't know of a way to accomplish the second feat, but the first one is actually fairly easy, as I'll demonstrate in the next chapter.

Chapter 4

Presentation and Layout

Although *TextBlock* is surely one of the most important elements supported by Silverlight, the previous chapters haven't even shown a good generalized way to display multiple *TextBlock* elements in your application! This chapter will stick its toes into the crucial field of *Panel* elements that provide the basis of Silverlight's dynamic layout system. Along the way, I'll show how to supplement text with images and simple graphical shapes, and I'll describe some additional properties you can apply to all these elements.

Transforms

Until the advent of the Windows Presentation Foundation and Silverlight, transforms were mostly the tools of the graphics mavens. Mathematically speaking, transforms apply a simple formula to all the coordinates of a visual object and cause that object to be shifted to a different location, or change size, or be rotated.

In Silverlight, you can apply transforms to any object that descends from *UIElement*, and that includes text, bitmap images, movies, and all controls. The property defined by *UIElement* that makes transforms possible is *RenderTransform*, which you set to an object of type *Transform*. *Transform* is an abstract class, but it is the parent class to seven non-abstract classes:

- *TranslateTransform* to shift location
- *ScaleTransform* to increase or decrease size
- *RotateTransform* to rotate around a point
- *SkewTransform* to shift in one dimension based on another dimension
- *MatrixTransform* to express transforms with a standard matrix
- *TransformGroup* to combine multiple transforms
- *CompositeTransform* to specify a series of transforms in a fixed order

The whole subject of transforms can be quite complex, particularly when transforms are combined, so I'm really only going to show the basics here. Very often, transforms are used in combination with animations. Animating a transform is the most efficient way an animation can be applied to a visual object.

Suppose you have a *TextBlock* and you want to make it twice as big. That's easy: Just double the *FontSize*. Now suppose you want to make the text twice as wide but three times taller. The *FontSize* won't help you there. You need to break out the *RenderTransform* property as a property element and set a *ScaleTransform* to it:

```
<TextBlock … >
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="2" ScaleY="3" />
    </TextBlock.RenderTransform>
</TextBlock>
```

Most commonly, you'll set the *RenderTransform* property of an object of type *TranslateTransform*, *ScaleTransform*, or *RotateTransform*. If you know what you're doing, you can combine multiple transforms in a *TransformGroup*. In two dimensions, transforms are expressed as 3×3 matrices, and combining transforms is equivalent to matrix multiplication. It is well known that matrix multiplication is not commutative, so the order that transforms are applied makes a difference in the overall effect.

Although *TransformGroup* is normally an advanced option, I have nevertheless used *TransformGroup* in a little project named TransformExperiment that allows you to play with several kinds of transforms. It begins with all the properties set to their default values;

Silverlight Project: TransformExperiment   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Transform Experiment"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleX="1" ScaleY="1"
                                CenterX="0" CenterY="0" />
                <SkewTransform AngleX="0" AngleY="0"
                               CenterX="0" CenterY="0" />
                <RotateTransform Angle="0"
                                 CenterX="0" CenterY="0" />
                <TranslateTransform X="0" Y="0" />
            </TransformGroup>
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

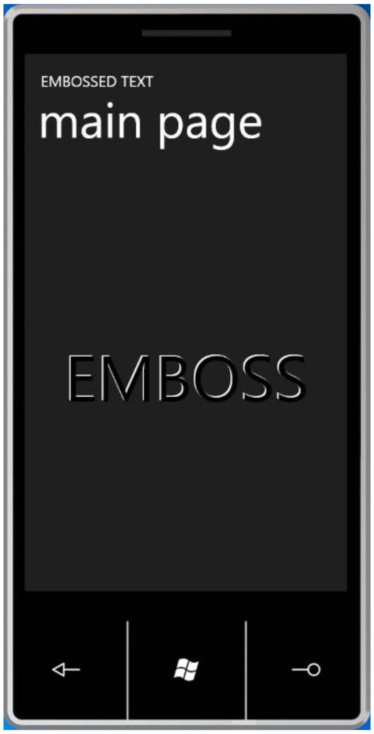
You can experiment with this program right in Visual Studio. At first you'll want to try out each type of transform independently of the others. Although it's at the bottom of the group, try *TranslateTransform* first. By setting the *X* property you can shift the text right or (with negative values) to the left. The *Y* property makes the text go down or up. Set *Y* equal to –400 or so and the text goes up into the title area!

TranslateTransform is useful for making drop shadows. and effects where the text seems embossed or engraved. Simply put two *TextBlock* elements in the same location with the same text, and all the same text properties, but different *Foreground* properties. Without any transforms, the second *TextBlock* sits on top of the first *TextBlock*. On one or the other, apply a small *ScaleTransform* and magic results. The EmbossedText project demonstrates this technique. Here are two *TextBlock* elements in the same *Grid*:

Silverlight Project: EmbossedText   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="EMBOSS"
               Foreground="White"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <TextBlock Text="EMBOSS"
               Foreground="Black"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.RenderTransform>
            <TranslateTransform X="2" Y="2" />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

The *TextBlock* underneath is white, and the one on top is black like the background but shifted a bit to let the white one show through a bit:



Generally this technique is applied to black text on a white background, but it looks pretty good with this color scheme as well.

Back in the TransformExperiment project, set the *TranslateTransform* back to the default values of 0, and experiment a bit with the *ScaleX* and *ScaleY* properties of the *ScaleTransform*. The default values are both 1. Larger values make the text larger in the horizontal and vertical directions; values smaller than 1 shrink the text. You can even use negative values to flip the text around its horizontal or vertical axes.

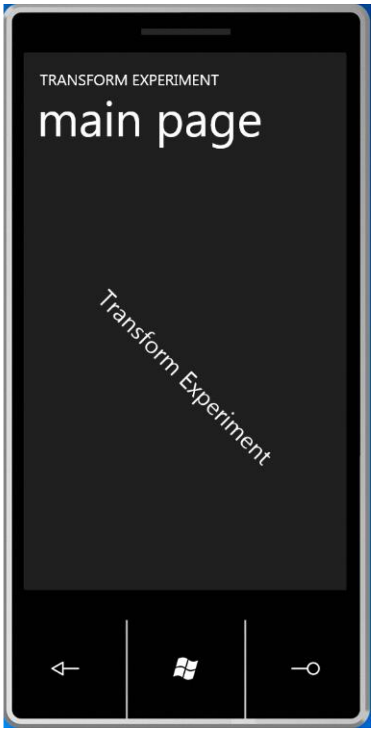
All scaling is relative to the upper-left corner of the text. In other words, as the text gets larger or smaller, the upper-left corner of the text remains in place. This might be a little hard to see because the upper-left corner that remains in place is actually a little *above* the horizontal stroke of the first 'T' in the text string, in the area reserved for diacritics such as accent marks and heavy-metal umlauts.

Suppose you want to scale the text relative to its *center*. That's the purpose of the *CenterX* and *CenterY* properties of the *ScaleTransform*. You can estimate the size of the text (or obtain it in code using the *ActualWidth* and *ActualHeight* properties of the *TextBlock*), divide the values by 2 and set *CenterX* and *CenterY* to the results. For the text string in TransformExperiment, try 96 and 13, respectively. Now the scaling is relative to the center.

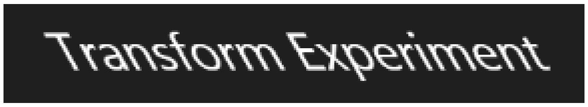
But there's a much easier way: *TextBlock* itself has a *RenderTansformOrigin* property that it inherits from *UIElement*. This property is a point in *relative coordinates* where (0, 0) is the upper-left corner, (1, 1) is the lower-right corner, and (0.5, 0.5) is the center. Set *CenterX* and *CenterY* back to 0, and set *RenderTransformOrigin* in the *TextBlock* like so:

```
RenderTransformOrigin="0.5 0.5"
```

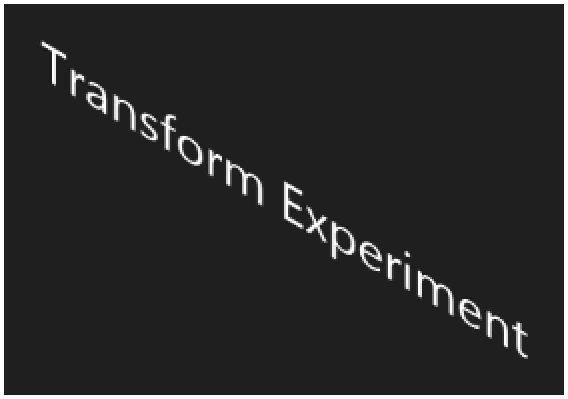
Leave *RenderTransformOrigin* at this value when you set the *ScaleX* and *ScaleY* properties of *ScaleTransform* back to the default values of 1, and play around with *RotateTransform*. As with scaling, rotation is always relative to a point. You can use *CenterX* and *CenterY* to set that point in absolute coordinates relative to the object being rotated, or you can use *RenderTransformOrigin* to use relative coordinates. The *Angle* property is in degrees, and positive angles rotate clockwise. Here's rotation of 45 degrees around the center.



The *SkewTransform* is hard to describe but easy to demonstrate. Here's the result when *AngleX* is set to 30 degrees:



For increasing *Y* coordinates, *X* coordinates are shifted to the right. Use a negative angle to simulate oblique (italic-like) text. Setting *AngleY* causes vertical shifting based on increasing X coordinates. Here's *AngleY* set to 30 degrees:



All the transforms that derive from *Transform* are categorized as affine ("non infinity") transforms. A rectangle will never be transformed into anything other than a parallelogram.

It's easy to convince yourself that the order of the transforms makes a difference. For example, in TransformExperiment on the *ScaleTransform* set *ScaleX* and *ScaleY* to 4, and on the *TranslateTransform* set *X* and *Y* to 100. The text is being scaled by a factor of 4 and then translated 100 pixels. Now cut and paste the markup to move the *TranslateTransform* above the *ScaleTransform*. Now the text is first translated by 100 pixels and scaled, but the scaling applies to the original translation factors as well, so the text is effectively translated by 400 pixels.

If you have a need to combine transforms in the original order that I had them in TransformExperiment—the order scale, skew, rotate, translate—you can use *CompositeTransform* to set them all.

Let's make a clock. It won't be a digital clock, but it won't be entirely an analog clock either. That's why I call it HybridClock. The hour, minute, and second hands are actually *TextBlock* objects that are rotated around the center. Here's the XAML:

Silverlight Project: HybridClock   File: MainPage.xaml (excerpt)

```
<Grid Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="referenceText"
               Text="THE SECONDS ARE 99"
               Foreground="Transparent" />

    <TextBlock Name="hourHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Name="minuteHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Name="secondHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Of the four *TextBlock* elements in the same *Grid*, the first is transparent and used solely for measuring by the code part of the program for measurement. The other three *TextBlock* elements are colored white through property inheritance, and have default *CompositeTransform* objects attached to their *RenderTransform* properties. The code-behind file defines a few fields that will be used throughout the program. The constructor sets a handler for the *Loaded* event:

Silverlight Project: HybridClock   File: MainPage.xaml.cs (excerpt)

```
namespace HybridClock
{
    public partial class MainPage : PhoneApplicationPage
    {
        Point gridCenter;
        Size textSize;
        double scale;

        public MainPage()
        {
            InitializeComponent();
            Loaded += OnMainPageLoaded;
        }

        void OnMainPageLoaded(object sender, RoutedEventArgs args)
        {
            gridCenter = new Point(ContentGrid.ActualWidth / 2,
                                   ContentGrid.ActualHeight / 2);

            textSize = new Size(referenceText.ActualWidth,
                                referenceText.ActualHeight);

            scale = gridCenter.X / textSize.Width;

            DispatcherTimer tmr = new DispatcherTimer();
            tmr.Interval = TimeSpan.FromSeconds(1);
            tmr.Tick += OnTimerTick;
            tmr.Start();
        }
        …
    }
}
```

The *Loaded* event occurs just once after the visual tree has been constructed. HybridClock takes this opportunity to determine the center of the *ContentGrid*, and the size of the *TextBlock* named *referenceText*. From these two items the program can calculate a scaling factor that will expand the *referenceText* so it is exactly as wide as half the smallest dimension of the *Grid*, and the other TextBlock elements proportionally.

The *Loaded* handler then creates a *DispatcherTimer* and sets it for a one-second tick. Although the *DispatcherTimer* is in the *System.Windows.Threading* namespace, the callback occurs in the same thread that created the timer. This is useful for referencing user-interface objects.

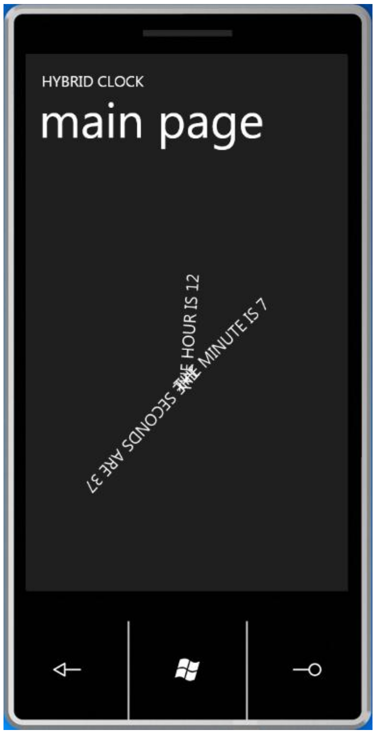
The timer callback obtains the current time and calculates the angles for the second, minute, and hour hands relative to their high-noon positions. Each hand gets a call to *SetupHand* to do all the remaining work.

Silverlight Project: HybridClock   File: MainPage.xaml.cs (excerpt)

```
void OnTimerTick(object sender, EventArgs e)
{
    DateTime dt = DateTime.Now;
    double angle = 6 * dt.Second;
    SetupHand(secondHand, "THE SECONDS ARE " + dt.Second, angle);
    angle = 6 * dt.Minute + angle / 60;
    SetupHand(minuteHand, "THE MINUTE IS " + dt.Minute, angle);
    angle = 30 * (dt.Hour % 12) + angle / 12;
    SetupHand(hourHand, "THE HOUR IS " + (((dt.Hour + 11) % 12) + 1), angle);
}

void SetupHand(TextBlock txtblk, string text, double angle)
{
    txtblk.Text = text;
    CompositeTransform xform = txtblk.RenderTransform as CompositeTransform;
    xform.CenterX = textSize.Height / 2;
    xform.CenterY = textSize.Height / 2;
    xform.ScaleX = scale;
    xform.ScaleY = scale;
    xform.Rotation = angle - 90;
    xform.TranslateX = gridCenter.X - textSize.Height / 2;
    xform.TranslateY = gridCenter.Y - textSize.Height / 2;
}
```

The *CompositeTransform* must perform several chores. The translation part must move the *TextBlock* elements so the beginning of the text is positioned in the center of the *Grid*. But I don't want the upper-left corner of the text to be positioned in the center. I want a point that is offset by that corner by half the *textSize.Height*. That explains the *TranslateX* and *TranslateY* properties. Recall that in the *CompositeTransform* the translation is applied last; that's why I put these properties at the bottom of the method, even though the order that these properties are set is irrelevant.

Both *ScaleX* and *ScaleY* are set to the scaling factor calculated earlier. The *angle* parameter passed to the method is relative to the high-noon position, but the *TextBlock* elements are positioned at 3:00. That why the *Rotation* angle offsets the *angle* parameter by –90 degrees. Both scaling and rotation are relative to *CenterX* and *CenterY*, which is a point at the left end of the text, but offset from the upper-left corner by half the text height. Here's the clock in action:



Windows Phone also supports the *Projection* transform introduced in Silverlight 3, but it's almost entirely used in connection with animations, so I'll hold off on *Projection* until then.

The Border Element

The *TextBlock* doesn't include any kind of border that you can draw around the text. Fortunately Silverlight has a *Border* element that you can use to enclose a *TextBlock* or any other type of element. The *Border* has a property named *Child* of type *UIElement*, which means you can only put one element in a *Border*; however, the element you put in the *Border* can be a panel, and you can then add multiple elements to that panel.

If you run the XamlExperiment program from the last chapter, you can put a *TextBlock* in a *Border* like so:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25">
        <Border.Child>
            <TextBlock Text="Hello, Windows Phone!" />
        </Border.Child>
    </Border>
</Grid>
```

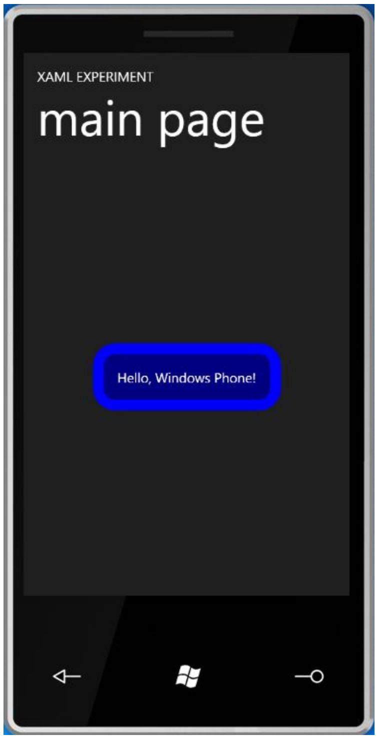
The *Child* property is the *ContentProperty* attribute of *Border* so the *Border.Child* tags are not required. Without setting any *HorizontalAlignment* and *VerticalAlignment* properties, the *Border* element occupies the entire area of the *Grid*, and the *TextBlock* occupies the entire area of the *Border*, even through the text itself sits at the upper-left corner. You can center the *TextBlock* within the *Border*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25">
        <TextBlock Text="Hello, Windows Phone!"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />
    </Border>
</Grid>
```

Or, you can center the *Border* within the *Grid*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
        <TextBlock Text="Hello, Windows Phone!" />
    </Border>
</Grid>
```

At this point, the *Border* contracts in size to become only large enough to fit the *TextBlock*. You can also set the *HorizontalAlignment* and *VerticalAlignment* properties of the *TextBlock* but they would now have no effect. You can give the *TextBlock* a little breathing room inside the border by either setting the *Margin* property of the *TextBlock*, or the *Padding* property of the *Border*:



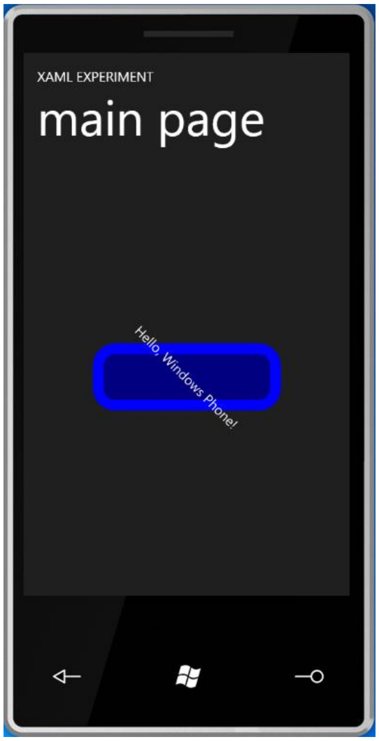
And now we have an attractive *Border* surrounding the *TextBlock*. The *BorderThickness* property is of type *Thickness*, the same structure used for *Margin* or *Padding*, so you can potentially have four different thicknesses for the four sides. The *CornerRadius* property is of type *CornerRadius*, a structure that also lets you specify four different values for the four corners. The *Background* and *BorderBrush* properties are of type *Brush*, so you can use gradient brushes.

What happens if you set a *RenderTransform* on the *TextBlock*? Try this:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Padding="20">
        <TextBlock Text="Hello, Windows Phone!"
                   RenderTransformOrigin="0.5 0.5">
            <TextBlock.RenderTransform>
```

```
                <RotateTransform Angle="45" />
            </TextBlock.RenderTransform>
        </TextBlock>
    </Border>
</Grid>
```

Here’s what you get:



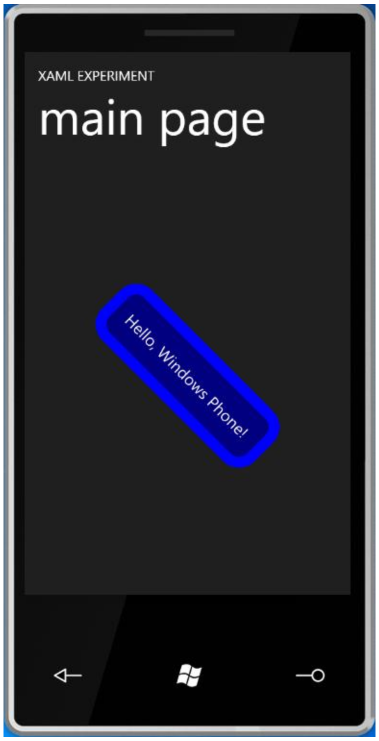
The *RenderTransform* property is called a *render* transform for a reason: It only affects rendering. It does affect how the element is perceived in the layout system. The Windows Presentation Foundation has a second property named *LayoutTransform* that does affect layout. If you were coding in WPF and set the *LayoutTransform* in this case, the Border would expand to fit the rotated text. But Silverlight does not yet have a *LayoutTransform* and, yes, it is sometimes sorely missed.

Your spirits might perk up, however, when you try moving the *RenderTransform* (and *RenderTransformOrigin*) from the *TextBlock* to the *Border*, like this:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Border Background="Navy"
```

```
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Padding="20"
            RenderTransformOrigin="0.5 0.5">
        <Border.RenderTransform>
            <RotateTransform Angle="45" />
        </Border.RenderTransform>

        <TextBlock Text="Hello, Windows Phone!" />
    </Border>
</Grid>
```

Transforms affect not only the element to which they are applied, but all child elements as this screen shot makes clear:



This means that you can apply transforms to whole sections of the visual tree, and within that transformed visual tree you can have additional compounding transforms.

StackPanel and ScrollViewer

The *Border* class defines a property named *Child* of type *UIElement*. The *Panel* class defines a property named *Children* of type *UIElementCollection*. Big difference!

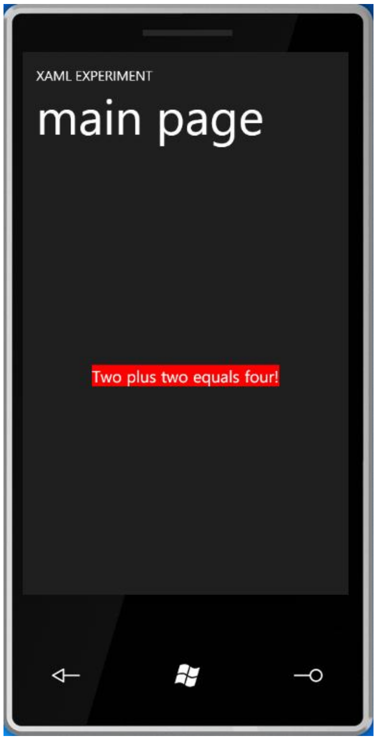
The *Border* doesn't have a whole lot of decision making to perform concerning that child. The child element is inside the *Border*, and that's about it. But a panel can host multiple children in a variety of ways. Perhaps it arranges the children in a stack, or a grid, or perhaps it docks the children on its edges, or arranges them in a circle.

For this reason, the *Panel* class itself is abstract. Silverlight provides three panels you can use on the phone; later in this book I'll show you how to write your own. The three basic panels are *StackPanel*, which is probably the simplest kind of panel, *Grid*, which is the most sophisticated and powerful, and *Canvas*, which should mostly be ignored except for some special purposes.

StackPanel arranges its children in a stack, either horizontally or vertically. Try this in XamlExperiment:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                Background="Red">
        <TextBlock FontSize="24" Text="Two " />
        <TextBlock FontSize="24" Text="plus " />
        <TextBlock FontSize="24" Text="two " />
        <TextBlock FontSize="24" Text="equals " />
        <TextBlock FontSize="24" Text="four!" />
    </StackPanel>
</Grid>
```

The only property *StackPanel* defines on its own is *Orientation*, which you set to a member of the *Orientation* enumeration, either *Horizontal* or *Vertical*. The default is *Vertical*. Here the *StackPanel* is aligned in the center of the *Grid* and has five children, which it arranges in a nice little row:



As you can see by its red background, this *StackPanel* is only as large as it needs to be to fit its children. It might seem rather silly to concatenate text in this way, but it's actually a very useful technique. Sometimes a program has some fixed text defined in XAML, mixed with some variable text from code or a data binding. The *StackPanel* does a nice job of piecing it together without any extraneous spacing.

The *StackPanel* is used most often in a vertical orientation. Each element gets only as much vertical space as it needs. Still, however, it could be that the screen is not large enough to fit all the elements. In that case, you can put the *StackPanel* in a *ScrollViewer*, a control that determines how large its content needs to be, and provides a scrollbar or two if there's insufficient space. By default, the vertical scrollbar is visible and the horizontal scrollbar is hidden, but you can change that with the *VerticalScrollBarVisibility* and *HorizontalScrollBarVisibility* properties.

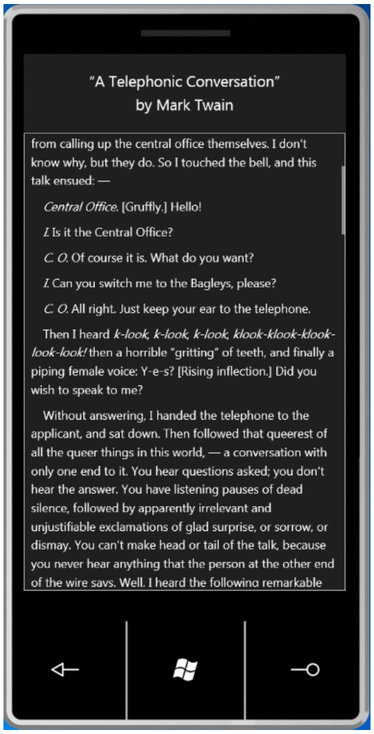
The next program is an ebook reader. Well, not exactly an *ebook* reader. It's more like an *eshort* reader, and I guess it's not very versatile: It displays a little humor piece written by Mark Twain in 1880 and believed to be the first description of the experience of listening to a person talk on the telephone without hearing the other side of the conversation. The woman talking on the telephone is Mark Twain's wife, Olivia.

Silverlight Project: TelephonicConversation   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <ScrollViewer FontSize="18"
                  Padding="5">
        <StackPanel>
            <TextBlock TextWrapping="Wrap" Margin="5">
                &#x2003;I consider that a conversation by telephone – when you are
                simply sitting by and not taking any part in that conversation –
                is one of the solemnest curiosities of this modern life.
                Yesterday I was writing a deep article on a sublime philosophical
                subject while such a conversation was going on in the
                room. I notice that one can always write best when somebody
                is talking through a telephone close by. Well, the thing began
                in this way. A member of our household came in and asked
                me to have our house put into communication with Mr. Bagley's,
                down town. I have observed, in many cities, that the sex
                always shrink from calling up the central office themselves. I
                don't know why, but they do. So I touched the bell, and this
                talk ensued: –
            </TextBlock>
            <TextBlock TextWrapping="Wrap" Margin="5">
                &#x2003;<Run FontStyle="Italic">Central Office.</Run>
                [Gruffly.] Hello!
            </TextBlock>
            …
            <TextBlock TextWrapping="Wrap" Margin="5">
                &#x2003;A… man delivers a single brutal "Good-by," and that is the
                end of it. Not so with the gentle sex, – I say it in their praise;
                they cannot abide abruptness.
            </TextBlock>
            <TextBlock Margin="5" TextAlignment="Right">
                – <Run FontStyle="Italic">Atlantic Monthly</Run>, June 1880
            </TextBlock>
        </StackPanel>
    </ScrollViewer>
</Grid>
```

I defined the *FontSize* property I wanted for the text right in the *ScrollViewer*. (Although *Panel* and its derivatives don't have font-related properties, *Control* and its derivatives, including *ScrollViewer*, do.) The *FontSize* property is inherited through the visual tree so it applies to each of the *TextBlock* elements. *ScrollViewer* is also given a little *Padding* value so the *StackPanel* doesn’t go quite to the edges; in addition, each *TextBlock* gets a *Margin* property. Each paragraph has a composite margin on both the left and right sides of 10 pixels, and 10 pixels separate each *TextBlock*.

I also put a Unicode character &#x2003; at the beginning of each paragraph. This is the Unicode em-space and effectively indents the first line. *ScrollViewer* responds to touch, so you can easily scroll through and read the whole story.



“A Telephonic Conversation”
by Mark Twain

from calling up the central office themselves. I don’t know why, but they do. So I touched the bell, and this talk ensued: —

Central Office. [Gruffly.] Hello!

I. Is it the Central Office?

C. O. Of course it is. What do you want?

I. Can you switch me to the Bagleys, please?

C. O. All right. Just keep your ear to the telephone.

Then I heard *k-look, k-look, k-look, klook-klook-klook-look-look!* then a horrible "gritting" of teeth, and finally a piping female voice: Y-e-s? [Rising inflection.] Did you wish to speak to me?

Without answering, I handed the telephone to the applicant, and sat down. Then followed that queerest of all the queer things in this world, — a conversation with only one end to it. You hear questions asked; you don’t hear the answer. You have listening pauses of dead silence, followed by apparently irrelevant and unjustifiable exclamations of glad surprise, or sorrow, or dismay. You can’t make head or tail of the talk, because you never hear anything that the person at the other end of the wire says. Well, I heard the following remarkable

The Mechanism of Layout

I want you to perform a little experiment. Go into the XAML file of the TelephonicConversation project and insert the following setting into the *ScrollViewer* tag:

```
HorizontalScrollBarVisibility="Visible"
```

Almost immediately you'll see a startling change: All the *TextBlock* elements become long single lines of text with no wrapping. What happened? How does setting a property on the *ScrollViewer* have an effect like that on the various *TextBlock* elements?

Getting a good feel for the layout system is one of the most important Silverlight programming skills you can acquire. The layout system is very powerful, but for the uninitiated, it can also seem quite strange.

Layout in Silverlight is a two-pass process starting at the top of the visual tree and working down. In a Silverlight phone application, it begins with the *PhoneApplicationFrame*, then the *PhoneApplicationPage*, then most likely a *Grid*. In Telephonic Conversation, the process continues into the *ScrollViewer*, which probably contains its own *Border*, and then eventually the *StackPanel*, and finally the *TextBlock* elements. These *TextBlock* elements have no children so that's the end of the line.

During the first pass, every element in the tree is responsible for querying its children to obtain their desired size. In the second pass, elements are responsible for arranging their children relative to their surface. The arrangement can be trivial or complex. For example, a *Border* has only one child and need only take account of its own *BorderThickness* to determine where to position that child. But *Panel* derivatives must arrange their children in unique ways.

When a parent queries the size of its children, it effectively says "Here's an available size for you. How big do you want to be?" and each child calculates its desired size. If that child itself has children, then it must determine its own size by querying its children's sizes, until the process gets down to elements like *TextBlock* that have no children. A *TextBlock*, for example, might be displaying a long piece of text and might have its *TextWrapping* property set to *Wrap*. In that case, the *TextBlock* looks at the *Width* property of that available size and determines where lines should break. It then knows how much vertical space it needs to display all the text. The *TextBlock* then calculates the size it wants to be.

Here's the catch: Sometimes when a parent presents its children with an available size, either the *Width* or *Height* or both could be set to the special floating-point value *Double.PositiveInfinity*. However, the child cannot respond by claiming an infinite desired size. The desired size of a child must be finite.

As you’ve seen, elements such as the *Border* and *TextBlock* have default *HorizontalAlignment* and *VerticalAlignment* properties that cause them to fill the interior of their parents. But this only happens when these elements are given a finite available size. If the elements are given an infinite available size, they must report a desired size that is only sufficient for themselves (and their children) and no more. For this reason, it often happens than an element offered a finite available size will be larger than an element offered an infinite available size!

The overall available size on the phone is the finite size of the display, and *PhoneApplicationFrame* and *PhoneApplicationPage* will use that size. The standard *Grid* in the MainPage.xaml created by Visual Studio divvies that size up among its children. But consider the first *StackPanel* I showed you with an *Orientation* of *Horizontal*. That *StackPanel* makes a size available to its children with a *Height* equal to its own height but a *Width* of infinity. Each *TextBlock* in that *StackPanel* calculates a desired size based on the width of its own text string.

A *StackPanel* with an *Orientation* of *Vertical* offers an available size to each of its children with a *Width* based on its own size and a *Height* of infinity. A *TextBlock* with *TextWrapping* set to *Wrap* can use that finite width to calculate a desired height. But the *StackPanel* could end up with an accumulated height of its children greater than its own available height.

That’s where the *ScrollViewer* comes into play. The *ScrollViewer* has a default *VerticalScrollBarVisibility* setting of *Visible*. Regardless of the finite size of the *ScrollViewer*, to its child (in this example, a vertical *StackPanel*) it offers an available size with *Width* equal to its own width but a *Height* of infinity. The *StackPanel* also offers to its children a width equal to its own width (which equals the width of the *ScrollViewer*) and a *Height* of infinity. Each *TextBlock* determines how large it needs to be. The *StackPanel* accumulates those heights and sets its own desired height. The *ScrollViewer* than uses that desired height and its own actual height for the scroll logic.

When you set the *HorizontalScrollBarVisibility* property of *ScrollViewer* to *Visible*, then the *ScrollViewer* gives its child (the vertical *StackPanel*) an available *Width* of infinity. The vertical *StackPanel* passes that on to its own children, and that’s why the *TextBlock* elements no longer wrap text.

You can put a vertical *StackPanel* in a vertical *ScrollViewer* (that is, a *ScrollViewer* with *VerticalScrollBarVisibility* set to *Visible* but a *HorizontalScrollBarVisibility* set to *Hidden*) and it will work fine. But you can't put a vertical *ScrollViewer* in a vertical *StackPanel* and expect it to work right. The vertical *ScrollViewer* will be given an available height of infinity by the parent *StackPanel* and then report a desired size based on the size of its child. The *StackPanel* will give the *ScrollViewer* that desired size and the *ScrollViewer* will have nothing to do.

Obviously you'll get accustomed to this layout system over time; the real conceptual breakthroughs come when you derive a class from *Panel* and see how it works from the inside.

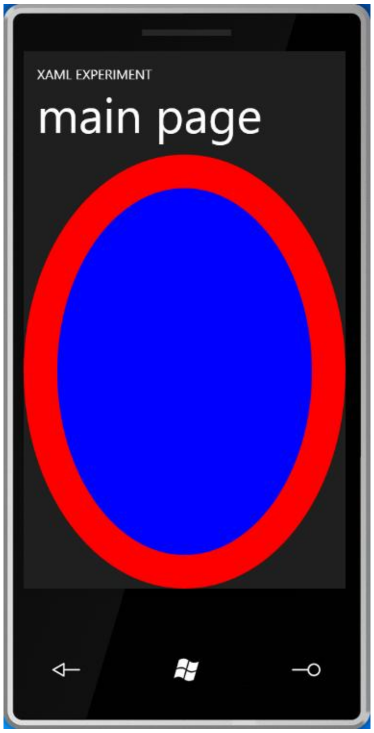
Simple Shapes

The elements you'll use to display vector graphics are part of the *System.Windows.Shapes* namespace, and I'll discuss those classes in a later chapter. However, two of the classes—*Ellipse* and *Rectangle*—are a little different from the others in that you can use them without specifying any coordinate points.

Go back to XamlExperiment again and insert this *Ellipse* element into *ContentGrid*:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Ellipse Fill="Blue"
             Stroke="Red"
             StrokeThickness="50" />
</Grid>
```

You'll see a blue ellipse with a red outline fill the *Grid*:



Now try setting *HorizontalAlignment* and *VerticalAlignment* to *Center*. The *Ellipse* disappears. What happened?

The *Ellipse* has no intrinsic minimum size. When allowed to, it will assume the size of its container, but if it's forced to become small, it will become as small as possible, which is nothing at all. If you put an *Ellipse* or *Rectangle* in a *StackPanel*, it will also shrink into nothingness. This is one case where explicitly setting *Width* and *Height* properties is appropriate.

Images and Media

A Silverlight program can also display bitmap images and play movies. In conventional Silverlight programs that run on the Web, these images can be stored on the Web somewhere—either on the same site as the Silverlight application itself or on some other site—and referenced with URLs. You can do that in a Silverlight phone application as well, but your application is not always guaranteed to have Web access. For this reason, any image that your program requires should be part of the executable itself. Here's how to do it.

In a Visual Studio project, right-click the project name and choose Add and then New Folder. Name the folder Media or Assets or Images or whatever you want. (This step is not strictly necessary but it makes for a tidier project.) Then, right-click the folder name and choose Add and Existing Item. Select an image file. Only JPEG and PNG files are supported by Silverlight!

From the Add button choose either Add or Add as Link. If you choose Add, a copy will be made and the file will be physically copied into a subdirectory of the project. If you choose Add as Link, only a file reference will be retained with the project but the file will still be copied into the executable.

The final step: Right-click the image filename and display Properties. For Build Action select Resource.

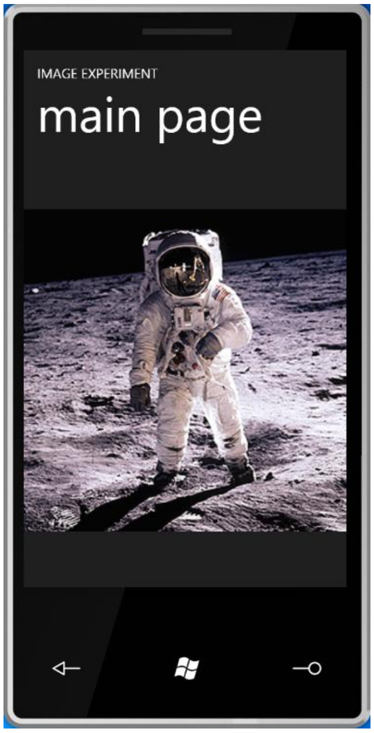
That's what I did in the ImageExperiment program. I created a directory named Media and added a file named BuzzAldrinOnTheMoon.png, which is the famous photograph taken with a Hasselblad camera by Neil Armstrong on July 21st, 1969. The photo is 288 pixels square.

The file is referenced in the MainPage.xaml file like this:

Silverlight Project: **ImageExperiment**   File: **MainPage.xaml** (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Image Source="Media/BuzzAldrinOnTheMoon.png" />
</Grid>
```

Here's how it appears:

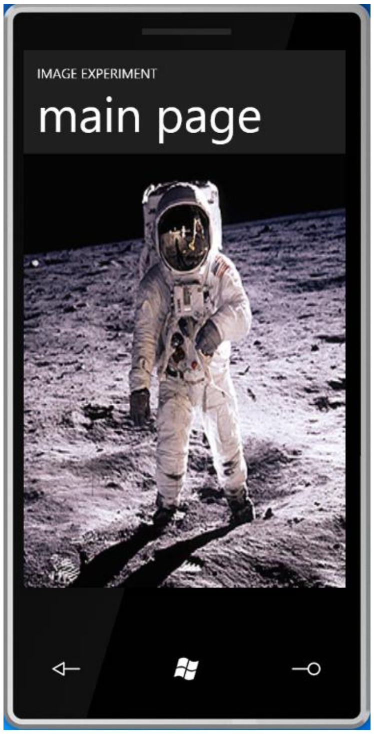


By default, the bitmap expands to the size of its container while maintaining the correct aspect ratio. Depending on the dimensions and aspect ratio of the container, the image is centered either horizontally or vertically. Of course you can change that behavior with the *HorizontalAlignment* and *VerticalAlignment* properties.

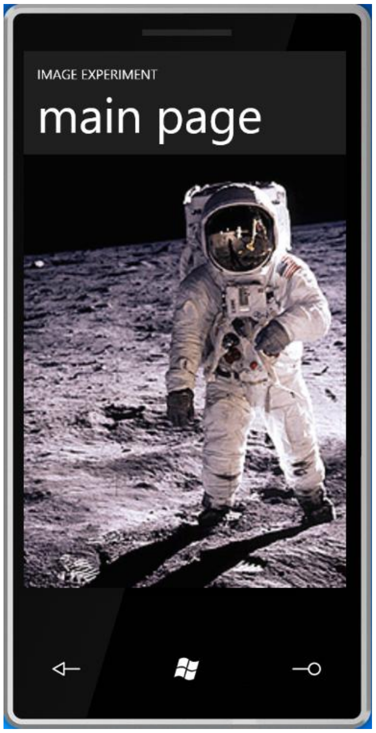
The stretching behavior is governed by a property defined by the *Image* element named *Stretch*, which is set to a member of the *Stretch* enumeration. The default value is *Uniform*, which you can set explicitly like this:

```
<Image Source="Media/BuzzAldrinOnTheMoon.png"
       Stretch="Uniform" />
```

The term “uniform” here means equally in both directions so the image is not distorted. You can also set *Stretch* to *Fill* to make the image fill its container by stretching unequally.

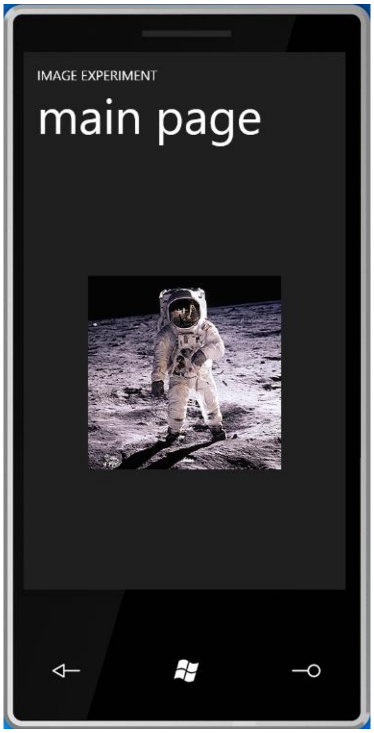


A compromise is *UniformToFill*:



Now the *Image* both fills the container and stretches uniformly to preserve the aspect ratio. How can both goals be accomplished? Well, in general the only way that can happen is by cropping the image. You can govern which edge gets cropped with the *HorizontalAlignment* and *VerticalAlignment* properties. What setting you use really depends on the particular image. For this one, I'd set *HorizontalAlignment* set to *Center* and *VerticalAlignment* set to *Top*.

The fourth option is *None* for no stretching. Now the image is displayed in its native size of 288 pixels square:

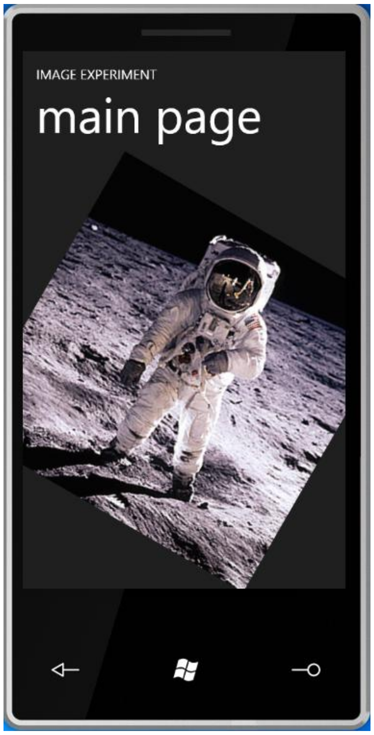


If you want to display the image in a particular size at the correct aspect ratio, you can set either an explicit *Width* or *Height* property. If you want to stretch non-uniformly to a particular dimension, specify both *Width* and *Height* and set *Stretch* to *Fill*.

You can set transforms on the *Image* element with the same ease that you set them on *TextBlock* elements:

```
<Image Source="Media/BuzzAldrinOnTheMoon.png"
       RenderTransformOrigin="0.5 0.5">
    <Image.RenderTransform>
        <RotateTransform Angle="30" />
    </Image.RenderTransform>
</Image>
```

Here it is:



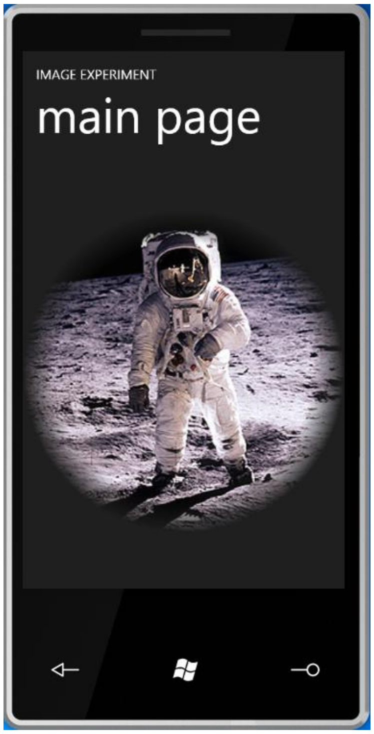
Modes of Opacity

UIElement defines an *Opacity* property that you can set to a value between 0 and 1 to make an element (and its children) more or less transparent. But a somewhat more interesting property is *OpacityMask*, which can "fade out" part of an element. You set the *OpacityMask* to an object of type *Brush*; most often you'll use one of the two *GradientBrush* derivatives. The actual color of the brush is ignored. Only the alpha channel is used to govern the opacity of the element.

For example, you can apply a *RadialGradientBrush* to the OpacityMask property of an *Image* element:

```
<Image Source="Media/BuzzAldrinOnTheMoon.png">
    <Image.OpacityMask>
        <RadialGradientBrush>
            <GradientStop Offset="0" Color="White" />
            <GradientStop Offset="0.8" Color="White" />
            <GradientStop Offset="1" Color="Transparent" />
        </RadialGradientBrush>
    </Image.OpacityMask>
</Image>
```

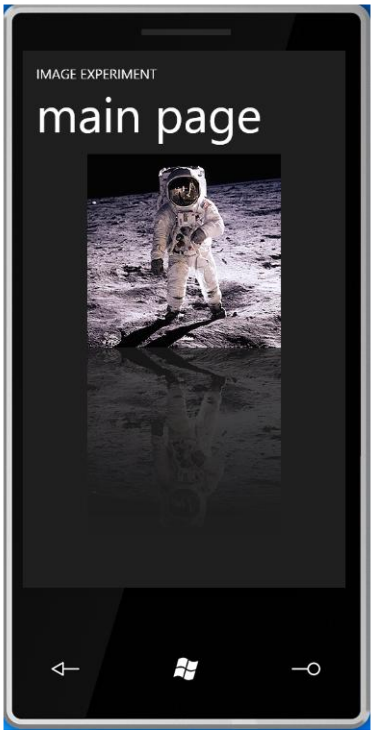
Notice that the *RadialGradientBrush* is opaque in the center, and continues to be opaque until a radius of 0.8, at which point the gradient goes to fully transparent at the edge of the circle. Here's the result, a very nice effect that looks much fancier than the few lines of XAML would seem to imply:



Here's a popular technique that uses two identical elements but one of them gets both a *ScaleTransform* to flip it upside down, and an *OpacityMask* to make it fade out:

```
<Image Source="Media/BuzzAldrinOnTheMoon.png"
       Stretch="None"
       VerticalAlignment="Top" />
<Image Source="Media/BuzzAldrinOnTheMoon.png"
       Stretch="None"
       VerticalAlignment="Top"
       RenderTransformOrigin="0.5 1">
    <Image.RenderTransform>
        <ScaleTransform ScaleY="-1" />
    </Image.RenderTransform>
    <Image.OpacityMask>
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <GradientStop Offset="0" Color="#00000000" />
            <GradientStop Offset="1" Color="#40000000" />
        </LinearGradientBrush>
    </Image.OpacityMask>
</Image>
```

There are two *Image* elements here, both the same size and aligned at the top and center. Normally the second one would be positioned on top of the other. But the second one has a *RenderTransform* set to a *ScaleTransform* that flips the image around the horizontal axis. The *RenderTransformOrigin* is set at the bottom of the element, which means the image flips around its bottom edge. Then a *LinearGradientBrush* is applied to the *OpacityMask* property to make the reflected image fade out:



Notice that the *GradientStop* values apply to the unreflected image, so that full transparency (the #00000000 value) seems to be at the top of the picture and then is reflected to the bottom of the composite display. It is often little touches like these that make a program’s visuals pop out just a little more and endear themselves to the user.

Part III

XNA

12:38
2
Phone
People
3
Text
20
Outlook
Pictures
XBOX
LIVE
Games
Me

Chapter 5
Principles of Movement

Much of the core of an XNA program is dedicated to moving sprites around the screen. Sometimes these sprites move under user control; at other times they move on their own volition as if animated by some internal vital force. Instead of moving real sprites, you can use instead move some text, and text is what I'll be sticking with for this entire chapter. The concepts and strategies involved in moving text around the screen are the same as those in moving sprites.

A particular text string seems to move around the screen when it's given a different position in the *DrawString* method during subsequent calls of the *Draw* method in *Game*. In Chapter 2, you'll recall, the *textPosition* variable was simply assigned a static value during the *LoadContent* method. This code puts the text in the center of the screen:

```
Rectangle clientBounds = this.Window.ClientBounds;
Vector2 textSize = kootenay14.MeasureString(text);
textPosition = new Vector2((int)(clientBounds.X + (clientBounds.Width - textSize.X) / 2),
                           (int)(clientBounds.Y + (clientBounds.Height - textSize.Y) / 2));
```

Most of the programs in this chapter recalculate *textPosition* during every call to *Update* so the text is drawn in a different location during the *Draw* method. Usually nothing fancy will be happening; the text will simply be moved from the top of the screen down to the bottom, and then back up to the top, and down again. Lather, rinse, repeat.

I'm going to begin with a rather "naïve" approach to moving text, and then refine it. If you're not accustomed to thinking in terms of vectors or parametric equations, my refinements will seem to make the program more complex, but you'll see that the program actually becomes simpler and more flexible.

The Naïve Approach

For this first attempt at text movement, I want to try something simple. I'm just going to move the text up and down vertically so the movement is entirely in one dimension. All we have to worry about is increasing and decreasing the *Y* coordinate of *textPosition*.

If you want to play along, you can create a Visual Studio project named NaiveTextMovement and add the 14-point Kootenay font to the Content directory. The fields in the *Game1* class are defined like so:

XNA Project: NaiveTextMovement File: Game1.cs (excerpt showing fields)

```
namespace NaiveTextMovement
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 240f / 1000;      // pixels per millisecond
        const string TEXT = "Hello, Windows Phone!";

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Rectangle clientBounds;
        Vector2 textSize;
        Vector2 textPosition;
        bool isGoingUp = false;

        …
    }
}
```

Nothing should be too startling here. I've defined both the SPEED and TEXT as constants. I like to write the speed as a ratio with 1000 in the denominator so I can easily convert it in my head to 240 pixels per second. The Boolean *isGoingUp* indicates whether the text is currently moving down the screen or up the screen.
The *LoadContent* method is very familiar from the program in Chapter 2 except that the viewport is saved as a field:

XNA Project: NaiveTextMovement   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    clientBounds = this.Window.ClientBounds;
    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    textSize = kootenay14.MeasureString(TEXT);



textPosition =
    new Vector2((int)(clientBounds.X + (clientBounds.Width - textSize.X) / 2), 0);
}
```

Notice that this *textPosition* centers the text horizontally but positions it on the top of the screen. As is usual with most XNA programs, all the real calculational work occurs during the *Update* method:

XNA Project: NaiveTextMovement   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
    if (!isGoingUp)
    {
        textPosition.Y += SPEED * (float)gameTime.ElapsedGameTime.TotalMilliseconds;

        if (textPosition.Y + textSize.Y > clientBounds.Bottom)
        {
            float excess = textPosition.Y + textSize.Y - clientBounds.Bottom;
            textPosition.Y -= 2 * excess;
            isGoingUp = true;
        }
    }
    else
    {
        textPosition.Y -= SPEED * (float)gameTime.ElapsedGameTime.TotalMilliseconds;

        if (textPosition.Y < clientBounds.Top)
        {
            float excess = clientBounds.Top - textPosition.Y;
            textPosition.Y += 2 * excess;
            isGoingUp = false;
        }
    }
    base.Update(gameTime);
}
```

The *GameTime* argument to *Update* has two crucial properties of type *TimeSpan*: *TotalGameTime* and *ElapsedGameTime*. This "game time" might not exactly keep pace with real time. There are some approximations involved so that animations are smoothly paced. But it's close. *TotalGameTime* reflects the length of time since the game was started; *ElapsedGameTime* is the time since the previous *Update* call. In general, *ElapsedGameTime* will always equal the same value—33-1/3 milliseconds reflecting the 30 Hz refresh rate of the phone's video display. I'll discuss exceptions to this rule in a later chapter.

You can use either *TotalGameTime* or *ElapsedGameTime* to pace movement. In this example, on the first call to *Update*, the *textPosition* has been calculated so the text is positioned on the upper edge of the screen and *isGoingUp* is false. The code increments *textPosition.Y* based on the product of SPEED (which is in units of pixels per millisecond) and the total milliseconds that have elapsed since the last *Update* call.

It could be that performing this calculation moves the text too far—for example, partially off the bottom of the screen. This can be detected if the vertical text position plus the height of the text is greater than the *Bottom* property of the client rectangle. In that case I calculate something I call *excess*. This is the distance that the vertical text position has exceeded the boundary of the display. I compensate with two times that—as if the text has bounced off the bottom and is now *excess* pixels above the bottom of the screen. At that point, *isGoingUp* is set to true.

The logic for moving up is (as I like to say) the same but completely opposite. The actual *Draw* override is simple:

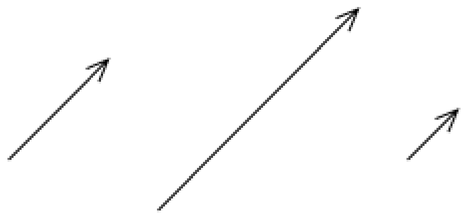
XNA Project: NaiveTextMovement   File: Game1.cs (excerpt)

```
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.Navy);

            spriteBatch.Begin();
            spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White);
            spriteBatch.End();

            base.Draw(gameTime);
        }
```

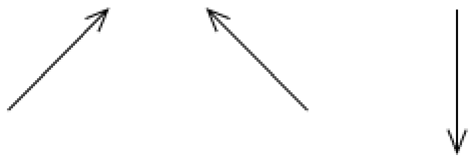
The big problem with this naïve approach is that it doesn't incorporate any mathematical tools that would allow us to do something a little more complex—for example, move the text diagonally rather than just in one dimension.
What's missing from the NaiveTextMovement program is any concept of direction that would allow escaping from horizontal or vertical movement. What we need are vectors.

A Brief Review of Vectors

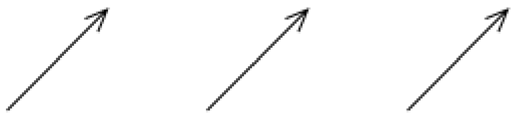
A vector is a mathematical entity that encapsulates both a direction and a magnitude. Very often a vector is symbolized by a line with an arrow. These three vectors have the same direction but different magnitudes:



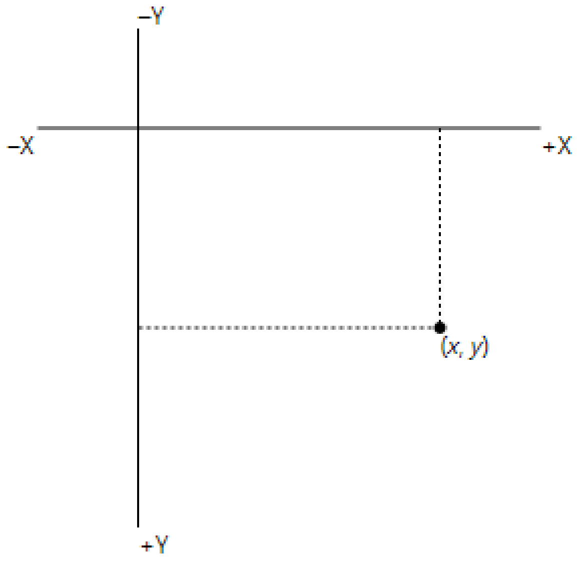
These three vectors have the same magnitude but different directions:



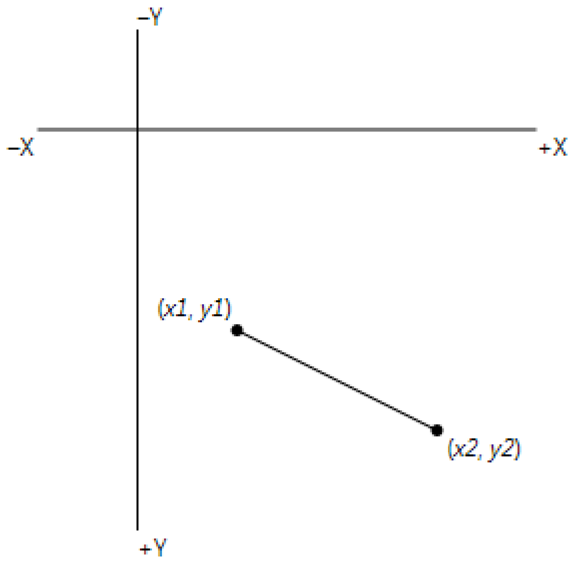
These three vectors have the same magnitude and the same direction, and hence are considered to be identical:



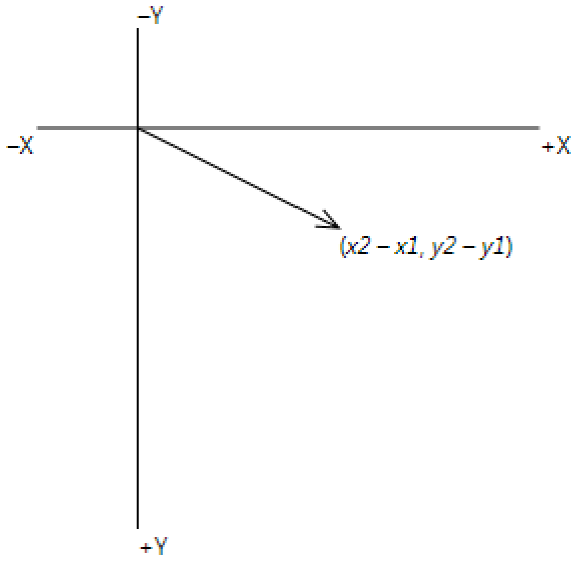
A vector has no location, so even if these three vectors seem to be in different locations and, perhaps for that reason, somewhat distinct, they really aren't in any location at all.
A point has no magnitude and no dimension. A point is *just* location. In two-dimensional space, a point is represented by a number pair (*x*, *y*) to represent a horizontal distance and a vertical distance from an origin (0, 0):



The figure shows increasing values of Y going down for consistency with the coordinate system in XNA.
A vector has magnitude and dimension but no location., but like the point a vector is represented by the number pair (*x*, *y*) except that it's usually written in boldface like **(*x*, *y*)** to indicate a vector rather than a point.
How can it be that two-dimensional points and two-dimensional vectors are both represented in the same way? Consider two points (*x1*, *y1*) and (*x2*, *y2*), and a line from the first point to the second:



That line has the same length and is in the same direction as a line from the origin to (*x2 – x1*, *y2 – y1*):



That magnitude and direction define the vector ***(x2 – x1, y2 – y1)***.
For that reason, XNA uses the same *Vector2* structure to store two-dimensional coordinate points and two-dimensional vectors.

For the vector ***(x, y)***, the magnitude is the length of the line from the point (0, 0) to the point (*x*, *y*). You can determine the length of the line and the vector using the Pythagorean Theorem, which has the honor of being the most useful tool in computer graphics programming:

$$length = \sqrt{x^2 + y^2}$$

The *Vector2* structure defines a *Distance* method that will perform this calculation for you. *Vector2* also includes a *DistanceSquared* method, which despite the longer name, is actually a simpler calculation. It is very likely that the *Vector2* structure implements *DistanceSquared* like this:

```
public float DistanceSquare()
{
    return x * x + y * y;
}
```

The *Distance* method is then based on *DistanceSquared*:

```
public float Distance()
{
    return (float)Math.Sqrt(DistanceSquare());
}
```

If you only need to compare magnitudes between two vectors, use *DistanceSquared* because it's faster. In the context of working with *Vector2* objects, the terms "length" and "distance" and "magnitude" can be used interchangeably.
Because you can represent points, vectors, and sizes with the same *Vector2* structure, the structure provides plenty of flexibility for performing arithmetic calculations. It's up to you to perform these calculations with some degree of intelligence. For example, suppose *point1* and *point2* are both objects of type *Vector2* but you're using them to represent points. It makes no sense to add those two points together, although *Vector2* will allow you to do so. But it makes lot of sense to *subtract* one point from another to obtain a vector:

```
Vector2 vector = point2 - point1;
```

The operation just subtracts the *X* values and the *Y* values; the vector is in the direction from *point1* to *point2* and its magnitude is the distance between those points. It is also common to add a vector to a point:

```
Vector2 point = point1 + vector;
```

This operation obtains a point that is a certain distance and in a certain direction from another point. You can multiply a vector by a single number. If *vector* is an object of type *Vector2*, then

```
vector *= 5;
```

is equivalent to:

```
vector.X *= 5;
vector.Y *= 5;
```

The operation effectively increases the magnitude of the vector by a factor of 5. Similarly you can divide a vector by a number. If you divide a vector by its length, then the resultant length becomes 1. This is known as a *normalized* vector, and *Vector2* has a *Normalize* method specifically for that purpose. The statement:

```
vector.Normalize();
```

is equivalent to

```
vector /= vector.Distance();
```

A normalized vector represents just a direction without magnitude, but it can be multiplied by a number to give it that length.
If *vector* has a certain length and direction, then –*vector* has the same length but the opposite direction. I'll make use of this in the next program coming up.
The direction of a vector ***(x, y)*** is the direction from the point (0, 0) to the point (*x*, *y*). You can convert that direction to an angle with the second most useful tool in computer graphics programming, the *Math.Atan2* method:

```
double angle = Math.Atan2(vector.Y, vector.X);
```

Notice that the *Y* component is specified first. The angle is in radians—remember that there are 2π radians to 360 degrees—measured clockwise from the positive X axis.
If you have an angle in radians, you can obtain a normalized vector from it like so:

```
Vector2 vector = new Vector2((float)Math.Cos(angle), (float)Math.Sin(angle));
```

The *Vector2* structure has four static properties. *Vector2.Zero* returns a *Vector2* object with both *X* and *Y* set to zero. That's actually an invalid vector because it has no direction, but it's useful for representing a point at the origin. *Vector2.UnitX* is the vector **(1, 0)** and *Vector2.UnitY* is the vector **(0, 1)**. *Vector2.One* is the point (1, 1) or the vector **(1, 1)**, which is useful if you're using the *Vector2* for horizontal and vertical scaling factors (as I do later in this chapter.)

Moving Sprites with Vectors

That little refresher course should provide enough knowledge to revamp the text-moving program to use vectors. The Visual Studio project is called VectorTextMovement. Here are the new fields:

XNA Project: VectorTextMovement   File: Game1.cs (excerpt showing fields)

```
namespace VectorTextMovement
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 240f / 1000;      // pixels per millisecond
        const string TEXT = "Hello, Windows Phone!";
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Vector2 midPoint;
        Vector2 pathVector;
        Vector2 pathDirection;
        Vector2 textPosition;
        …
    }
}
```

The text will be moved between two points (called *position1* and *position2* in the *LoadContent* method), and the *midPoint* field will store the point midway between those two points. The *pathVector* field is the vector from *position1* to *position2*, and *pathDirection* is *pathVector* normalized.
The *LoadContent* method calculates and initializes all these fields:

XNA Project: VectorTextMovement   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Rectangle clientBounds = this.Window.ClientBounds;

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);

    Vector2 position1 = new Vector2(clientBounds.Right - textSize.X, clientBounds.Top);
    Vector2 position2 = new Vector2(clientBounds.Left, clientBounds.Bottom - textSize.Y);
    midPoint = Vector2.Lerp(position1, position2, 0.5f);

    pathVector = position2 - position1;
    pathDirection = pathVector;
    pathDirection.Normalize();
    textPosition = position1;
}
```

The starting point is *position1*, which puts the text in the upper-right corner. The *position2* point is the lower-left corner. The calculation of *midPoint* makes use of the static *Vector2.Lerp* method, which stands for Linear intERPolation. If the third argument is 0, *Vector2.Lerp* returns its first argument; if the third argument is 1, *Vector2.Lerp* returns its second argument, and for values in between, the method performs a linear interpolation. *Lerp* is probably overkill for calculating a midpoint: All that's really necessary is to average the two *X* values and the two *Y* values.
Note that *pathVector* is the entire vector from *position1* to *position2* while *pathDirection* is the same vector normalized. The method concludes by initializing *textPosition* to *position1*. The use of these fields should become apparent in the *Update* method:

XNA Project: VectorTextMovement   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float pixelChange = SPEED * (float)gameTime.ElapsedGameTime.TotalMilliseconds;
    textPosition += pixelChange * pathDirection;

    if ((textPosition - midPoint).LengthSquared() >
                                    (0.5f * pathVector).LengthSquared())
    {
        float excess = (textPosition - midPoint).Length() -
                                    (0.5f * pathVector).Length();
        pathDirection = -pathDirection;
        textPosition += 2 * excess * pathDirection;
    }

    base.Update(gameTime);
}
```

The first time *Update* is called, *textPosition* equals *position1* and *pathDirection* is a normalized vector from *position1* to *position2*. This is the crucial calculation:

```
textPosition += pixelChange * pathDirection;
```

Multiplying the normalized *pathDirection* by *pixelChange* results in a vector that is in the same direction as *pathDirection* but with a length of *pixelChange*. The *textPosition* point is increased by this amount.

After a few seconds of *textPosition* increases, *textPosition* will go beyond *position2*. That can be detected when the length of the vector from *midPoint* to *textPosition* is greater than the length of half the *pathVector*. The direction must be reversed: *pathDirection* is set to the negative of itself, and *textPosition* is adjusted for the bounce.

Notice there's no longer a need to determine if the text is moving up or down. The calculation involving *textPosition* and *midPoint* works for both cases. Also notice that the *if* statement performs a comparison based on *LengthSquared* but the calculation of *excess* requires the actual *Length* method. Because the *if* clause is calculated for every *Update* call, it's good to try to keep the code efficient. The length of half the *pathVector* never changes, so I could have been even more efficient by storing *Length* or *LengthSquared* (or both) as fields.

The *Draw* method is the same as before:

XNA Project: VectorTextMovement   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);
    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Working with Parametric Equations

It is well known that when the math or physics professor says "Now let's introduce a new variable to simplify this mess," no one really believes that the discussion is heading towards a simpler place. But it's very often true, and it's the whole rationale behind parametric equations. Into a seemingly difficult system of formulas a new variable is introduced that is often simply called *t*, as if to suggest *time*. The value of *t* usually ranges from 0 to 1 (although that's just a convention) and other variables are calculated based on *t*. Amazingly enough, simplicity often results.

Let's think about the problem of moving text around the screen in terms of a "lap." One lap consists of the text moving from the upper-right corner (*position1*) to the lower-left corner (*position2*) and back up to *position1*.

How long does that lap take? We can easily calculate the lap time based on the regular speed in pixels-per-second and the length of the lap, which is twice the magnitude of the vector called *pathVector* in the previous program, and which was calculated as *position2* – *position1*. Once we know the speed in laps per millisecond, it should be easy to calculate a *tLap* variable ranging from 0 to 1, where 0 is the beginning of the lap and 1 is the end, at which point *tLap* starts over again at 0. From *tLap* we can get *pLap*, which is a relative position on the lap ranging from 0 (the top or *position1*) to 1 (the bottom or *position2*). From *pLap*, calculating *textPosition* should also be easy. The following table shows the relationship between these three variables:

tLap:	0	0.5	1
pLap:	0	1	0
textPosition:	position1	position2	position1

Probably right away we can see that

```
textPosition = position1 + pLap * pathVector;
```

The only really tricky part is the calculation of *pLap* based on *tLap*.

The ParametricTextMovement project contains the following fields:

XNA Project: ParametricTextMovement   File: Game1.cs (excerpt showing fields)

```
namespace ParametricTextMovement
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 240f / 1000;      // pixels per millisecond
        const string TEXT = "Hello, Windows Phone!";

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Vector2 position1;
        Vector2 pathVector;
        Vector2 textPosition;
        float lapSpeed;                      // laps per millisecond
        float tLap;
        …
    }
}
```

The only new variables here are *lapSpeed* and *tLap*. As is now customary, most of the variables are set during the *LoadContent* method:

XNA Project: ParametricTextMovement File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Rectangle clientBounds = this.Window.ClientBounds;

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);
    position1 = new Vector2(clientBounds.Right - textSize.X, clientBounds.Top);
    Vector2 position2 = new Vector2(clientBounds.Left, clientBounds.Bottom - textSize.Y);
    pathVector = position2 - position1;

    lapSpeed = SPEED / (2 * pathVector.Length());
}
```

In the calculation of *lapSpeed*, the numerator is in units of pixels-per-millisecond. The denominator is the length of the entire lap, which is two times the length of *pathVector*; therefore the denominator is in units of pixels per lap. Dividing pixels-per-millisecond by pixels-per-lap give you a speed in units of laps-per-millisecond.
One of the big advantages of this parametric technique is the sheer elegance of the *Update* method:

XNA Project: ParametricTextMovement File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
    tLap += lapSpeed * (float)gameTime.ElapsedGameTime.TotalMilliseconds;
    tLap %= 1;
    float pLap = tLap < 0.5f ? 2 * tLap : 2 - 2 * tLap;
    textPosition = position1 + pLap * pathVector;

    base.Update(gameTime);
}
```

The *tLap* field is incremented by the *lapSpeed* times the elapsed time in milliseconds. The second calculation removes any integer part, so if *tLap* is incremented to 1.1 (for example), it gets bumped back down to 0.1.

I will agree the calculation of *pLap* from *tLap*—which is a transfer function of sorts—looks like an indecipherable mess at first. But if you break it down, it's not too bad: If *tLap* is less than 0.5, then *pLap* is twice *tLap*, so for *tLap* from 0 to 0.5, *pLap* goes from 0 to 1. If *tLap* is greater than or equal to 0.5, *tLap* is doubled and subtracted from 2, so for *tLap* from 0.5 to 1, *pLap* goes from 1 back down to 0.

The *Draw* method remains the same:

XNA Project: ParametricTextMovement File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

There are some equivalent ways of performing these calculations. Instead of saving *pathVector* as a field you could save *position2*. Then during the *Update* method you would calculate *textPosition* using the *Vector2.Lerp* method:

```
textPosition = Vector2.Lerp(position1, position2, pLap);
```

In *Update*, instead of calculating an increment to *tLap*, you can calculate *tLap* directly from the *TotalGameState* of the *GameTime* argument and keep the variable local:

```
Float tLap = (lapSpeed * (float)gameTime.TotalGameTime.TotalMilliseconds) % 1;
```

Fiddling with the Transfer Function

I want to change one statement in the ParametricTextMovement program and improve the program enormously by making the movement of the text more natural and fluid. Can it be done? Of course!

Earlier I showed you the following table:

tLap:	0	0.5	1
pLap:	0	1	0
textPosition:	position1	position2	position1

In the ParametricTextMovement project I assumed that the transfer function from *tLap* to *pLap* would be linear, like so:

```
float pLap = tLap < 0.5f ? 2 * tLap : 2 - 2 * tLap;
```

But it doesn't have to be linear. The VariableTextMovement project is the same as ParametricTextMovent except for the calculation of *pLap*, which is now:

```
float pLap = (1 - (float)Math.Cos(tLap * MathHelper.TwoPi)) / 2;
```

When *tLap* is 0, the cosine is 1 and *pLap* is 0. When *tLap* is 0.5, the argument to the cosine function is π radians (180 degrees). The cosine is -1, it's subtracted from 1 and the result is divided by 2, so the result is 1. And so forth. But the difference is dramatic: The text now slows down as it approaches the corners and then speeds up as it moves away.
You can also try a couple others. This one slows down only when it reaches the bottom:

```
float pLap = (float)Math.Sin(tLap * Math.PI);
```

At the top of the screen it's at full velocity and seems to ricochet off the edge of the screen. This one's just the opposite and seems more like a bouncing ball slowed down by gravity at the top:

```
float pLap = 1 - Math.Abs((float)Math.Cos(tLap * Math.PI));
```

So you see that it's true: Using parametric equations not only simplified the code but made it much more amenable to enhancements.

Scaling the Text

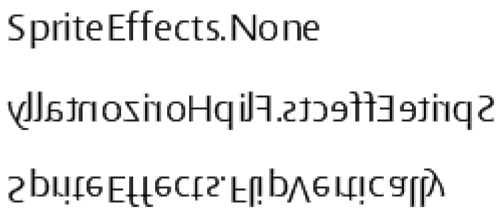
If you've glanced at the documentation of the *SpriteBatch* class, you've seen five other versions of the *DrawString* method. Until now I've been using this one:

```
DrawString(spriteFont, text, position, color);
```

There are also these two:

```
DrawString(spriteFont, text, position, color,
           rotation, origin, uniformScale, effects, layerDepth);

DrawString(spriteFont, text, position, color,
           rotation, origin, vectorScale, effects, layerDepth);
```

The other three versions of *DrawString* are the same except the second argument is a *StringBuilder* rather than a *string*. If you're displaying text that frequently changes, you might want to switch to *StringBuilder* to avoid lots of memory allocations from the local heap.
The additional arguments to these longer versions of *DrawString* are primarily for rotating, scaling, and flipping the text. The exception is the last argument, which is a *float* value that indicates how multiple sprites should be arranged from front (0) to back (1). I won't be using that argument in connection with *DrawString*.
The penultimate argument is a member of the *SpriteEffects* enumeration: The default is *None*. The *FlipHorizontally* and *FlipVertically* members both create mirror images but don't change the location of the text:



The argument labeled *origin* is a point with a default value of (0, 0). This argument is used for three related purposes:

- It is the point relative to the text string that is aligned with the *position* argument relative to the screen.
- It is the center of rotation. The *rotation* argument is a clockwise angle in radians.
- It is the center of scaling. Scaling can be specified with either a single number, which scales equally in the horizontal and vertical directions to maintain the correct aspect ratio, or a *Vector2*, which allows unequal horizontal and vertical scaling. (Sometimes these two modes of scaling are called isotropic—equal in all directions—and anisotropic.)

If you use one of the longer versions of *DrawString* and aren't interested in scaling, do not set that argument to zero! A sprite scaled to a zero dimension will not show up on the screen and you'll spend many hours trying to figure out what went wrong. (I speak from experience.) If you don't want any scaling, set the argument to 1 or the static property *Vector2.One*.
The very first XNA program in this book calculated *textPosition* based on the dimensions of the screen and the dimensions of the text:

```
textPosition = new Vector2((int)(clientBounds.X + (clientBounds.Width - textSize.X) / 2),
                           (int)(clientBounds.Y + (clientBounds.Height - textSize.Y) / 2));
```

The *textPosition* is the point on the screen where the upper-left corner of the text is to be aligned. With the longer versions of *DrawString*, some alternatives become possible. For example:

```
textPosition = new Vector2(clientBounds.Left + clientBounds.Width / 2,
                           clientBounds.Top + clientBounds.Height / 2);
origin = new Vector2(textSize.X / 2, textSize.Y / 2);
```

(I've eliminated the casting for purposes of clarity.) Now the *textPosition* is set to the center of the screen and the origin is set to the center of the text. This *DrawString* call uses those two variables to put the text in the center of the screen:

```
spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White,
                       0, origin, 1, SpriteEffects.None, 0);
```

The *textPosition* could be set to the lower-right corner of the screen, and *origin* could be set to the lower-right corner of the text:

```
textPosition = new Vector2(clientBounds.Right, clientBounds.Bottom);
origin = new Vector2(textSize.X, textSize.Y);
```

Now the text will be positioned in the lower-right corner of the screen.

Rotation and scaling are always relative to a point. This is most obvious with rotation, as anyone who's ever explored the technology of propeller beanies will attest. But scaling is also relative to a point. As an object grows or shrinks in size, one point remains anchored; that's the point indicated by the *origin* argument to *DrawString*. (The point could actually be outside the area of the text string.)

The ScaleTextToViewport project displays a text string in its center and expands it out to fill the viewport. To keep the scaling at least close to isotropic, the program aligns the text with the longest dimension of the screen. As with the other programs, it includes a font. Here are the fields:

XNA Project: ScaleTextToViewport   File: Game1.cs (excerpt showing fields)

```
namespace ScaleTextToViewport
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 0.5f / 1000;      // laps per millisecond
        const string TEXT = "Hello, Windows Phone!";

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Vector2 textPosition;
        Vector2 origin;
        Vector2 maxScale;
        Vector2 scale;
        float angle;
        float tLap;
        …
    }
}
```

The "lap" in this program is a complete cycle of scaling the text up and then back down to normal. During this lap, the *scale* field will vary between *Vector2.One* and *maxScale*.

The *LoadContent* method sets the *textPosition* field to the center of the screen, and the *origin* field to the center of the text. If the screen has a portrait orientation, the text is rotated using the *angle* field, and *maxScale* is calculated based on the swapped dimensions. All alignment, rotation, and scaling are based on both the center of the text and the center of the screen.

XNA Project: ScaleTextToViewport   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Rectangle clientBounds = this.Window.ClientBounds;

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);
    textPosition = new Vector2(clientBounds.Left + clientBounds.Width / 2,
                               clientBounds.Top + clientBounds.Height / 2);
    origin = new Vector2(textSize.X / 2, textSize.Y / 2);

    // flip 90 degrees if portrait mode
    if (clientBounds.Width < clientBounds.Height)
    {
        angle = MathHelper.PiOver2;
        maxScale = new Vector2(clientBounds.Height / textSize.X,
                               clientBounds.Width / textSize.Y);
    }
    else
    {
        maxScale = new Vector2(clientBounds.Width / textSize.X,
                               clientBounds.Height / textSize.Y);
    }
}
```

As in the previous couple programs, *tLap* repetitively cycles from 0 through 1. During this single lap, the *pLap* variable goes from 0 to 1 and back to 0, where 0 means unscaled and 1 means maximally scaled. The *Vector2.Lerp* method calculates *scale* based on *pLap*.

XNA Project: ScaleTextToViewport   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    tLap = (SPEED * (float)gameTime.TotalGameTime.TotalMilliseconds) % 1;
    float pLap = (1 - (float)Math.Cos(tLap * MathHelper.TwoPi)) / 2;
    scale = Vector2.Lerp(Vector2.One, maxScale, pLap);

    base.Update(gameTime);
}
```

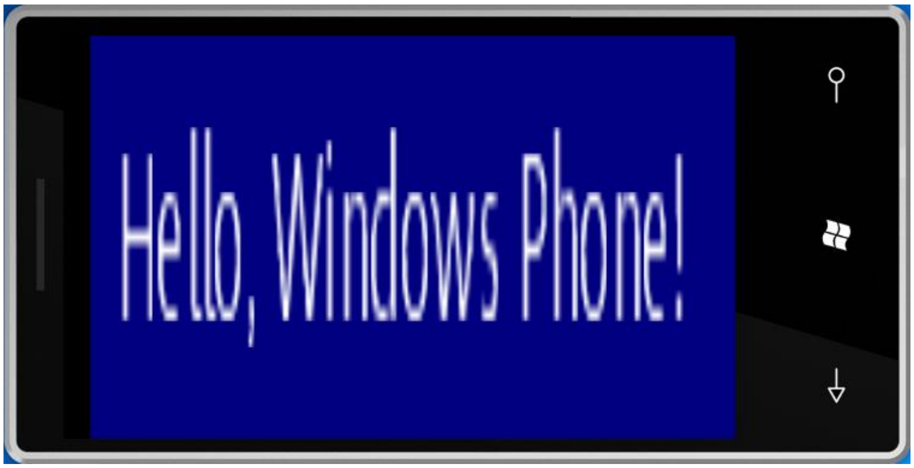
The *Draw* method uses one of the long versions of *DrawString* with the *textPosition*, *angle*, and *origin* calculated during *LoadContent*, and the *scale* calculated during *Update*:

XNA Project: ScaleTextToViewport   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White,
                           angle, origin, scale, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

As you run this program, you'll notice that the vertical scaling doesn't make the top and bottom of the text come anywhere close to the edges of the screen. The reason is that *MeasureString* returns a vertical dimension based on the maximum text height for the font, which includes space for descenders, possible diacritical marks, and a little breathing room as well.

It should also be obvious that you're dealing with a bitmap font here:



The display engine tries to smooth out the jaggies but it's debatable whether the fuzziness is an improvement. If you need to scale text and maintain smooth vector outlines, that's a job for Silverlight.

Two Text Rotation Programs

The ScaleTextToViewport project rotates the text 90 degrees for portrait displays, but it doesn't change the rotation during the *Update* method. Let's conclude this chapter with two programs that do.

It would be fairly simple to write a program that just rotates text around its center, but let's try something just a little more challenging. Let's gradually speed up the rotation and then stop it when a finger touches the screen. After the finger is released, the rotation should start up slowly again and then get faster. As the speed in revolutions-per-millisecond approaches the refresh rate of the video display (or some integral fraction thereof), the rotating text should seem to slow down, stop, and reverse. That will be fun to see as well.

A little background about working with acceleration: One of the most common forms of acceleration we experience in day-to-day life involves objects in free-fall. In a vacuum on the surface of the Earth, the effect of gravity produces an acceleration of a constant 32 feet per second per second, or, as it's often called, 32 feet per second squared:

$$a = 32\ ft/sec^2$$

The seemingly odd units of "feet per second per second" really means that every second, the velocity increases by 32 feet per second. At any time *t* in seconds, the velocity is given by the simple formula:

$$v(t) = at$$

where *a* is 32 feet per second squared. When the acceleration units of feet per second squared is multiplied by a time, the result has units of feet per second, which is a velocity. At 0 seconds, the velocity is 0. At 1 second the velocity is 32 feet per second. At 2 seconds the velocity is 64 feet per second, and so forth.

The distance an object in free fall travels is given by the formula:

$$x(t) = ½at^2$$

Rudimentary calculus makes this family of formulas comprehensible: The velocity is the derivative of the distance, and the acceleration is the derivative of the velocity. In this formula, the acceleration is multiplied by a time squared, so the units reduce to feet. At the end of one second the velocity of an object in free fall is up to 32 feet per second but because the free-fall started at a zero velocity, the object has only traveled a distance of 16 feet. By the end of two seconds, it's gone 64 feet.

In the TouchToStopRotation project, velocity is in units of revolutions per millisecond, so you shouldn't be surprised to see acceleration in units of revolutions per millisecond squared, a seemingly very tiny number:

XNA Project: TouchToStopRevolution   File: Game1.cs (excerpt showing fields)

```
namespace TouchToStopRevolution
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float ACCELERATION = 1f / 1000000;    // revs per msec squared
        const float MAXSPEED = 30f / 1000;          // revs per millisecond
        const string TEXT = "Hello, Windows Phone!";

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Vector2 textPosition;
        Vector2 origin;
        Vector2 statusPosition;
        float speed;
        float angle;
        StringBuilder strBuilder = new StringBuilder();
        …
    }
}
```

The MAXSPEED constant is set at 30 revolutions per second, which is the same as the frame rate. Theoretically, as the spinning text reaches that speed, it should appear to stop. In reality, it doesn't quite stop but it gets very slow. The ACCELERATION is 1 revolution per second squared, which means that the every second, the velocity increases by 1 revolution per second. At the end of the first second, the speed is 1 revolution per second. At the end of the second second, the speed is 2 revolutions per second. Velocity gets to MAXSPEED at the end of 30 seconds.

The fields include a *speed* variable and a *StringBuilder*, which I'll use for displaying the current velocity on the screen at *statusPosition*. The *LoadContent* method prepares most of these fields:

XNA Project: TouchToStopRevolution   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Rectangle clientBounds = this.Window.ClientBounds;
    textPosition = new Vector2(clientBounds.Left + clientBounds.Width / 2,
                               clientBounds.Top + clientBounds.Height / 2);

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);
    origin = new Vector2(textSize.X / 2, textSize.Y / 2);
    statusPosition = new Vector2((int)(clientBounds.Right - textSize.X),
                                 (int)(clientBounds.Bottom - textSize.Y));
}
```

The *Update* method increases *speed* based on the acceleration, and then increases *angle* based on *speed*.

XNA Project: TouchToStopRevolution   File: Game1.cs (excerpt)

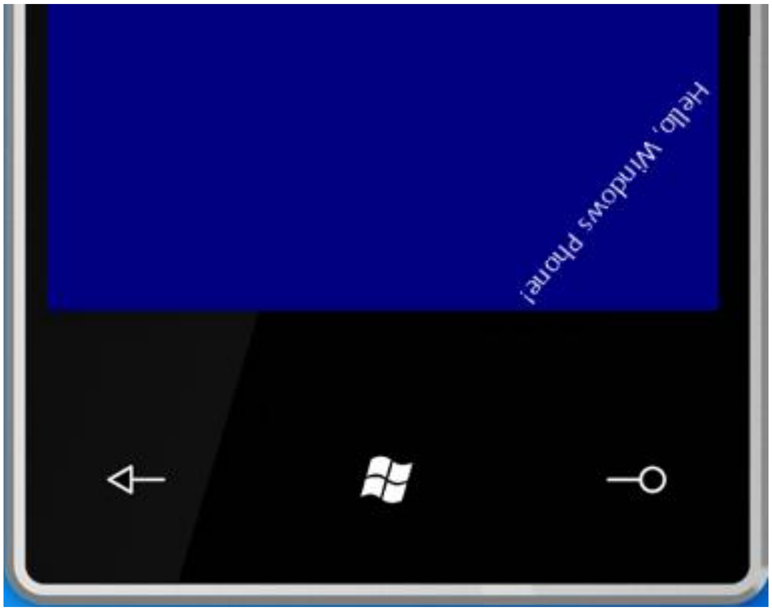
```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (TouchPanel.GetState().Count == 0)
    {
        speed += ACCELERATION * (float)gameTime.ElapsedGameTime.TotalMilliseconds;
        speed = Math.Min(MAXSPEED, speed);
        angle += MathHelper.TwoPi * speed * gameTime.ElapsedGameTime.Milliseconds;
        angle %= MathHelper.TwoPi;
    }
    else
    {
        if (speed == 0)
            SuppressDraw();

        speed = 0;
    }
    strBuilder.Remove(0, strBuilder.Length);
    strBuilder.AppendFormat(" {0:F1} revolutions/second", 1000 * speed);

    base.Update(gameTime);
}
```

If *TouchPanel.GetState()* returns a collection containing anything—that is, if anything is touching the screen—then *speed* is set back to zero. Moreover, the next time *Update* is called and something is still touching the screen, then *SuppressDraw* is called. So by touching the screen you're not only inhibiting the rotation of the text, but you're saving power as well. Also notice the use of *StringBuilder* to update the status field. The *Draw* method is similar to those in previous programs but with two calls to *DrawString*:

XNA Project: TouchToStopRevolution   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, strBuilder, statusPosition, Color.White);
    spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White,
                           angle, origin, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

For the final program in this chapter, I went back to a default origin of the upper-left corner of the text. But I wanted that upper-left corner of the text string to crawl around the inside perimeter of the display, and I also wanted the text to be fully visible at all times. The result is that the text rotates 90 degrees as it makes it way past each corner. Here's the text maneuvering around the lower-right corner of the display:



The program is called TextCrawl, and I'm afraid I found it necessary to resort to code that handles each of the four sides separately. I also decided to go full screen and use the *Viewport* rather than the *ClientBounds* just to simplify the code a bit. The fields should look mostly familiar at this point:

XNA Project: TextCrawl   File: Game1.cs (excerpt showing fields)

```
namespace TextCrawl
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 0.1f / 1000;    // laps per millisecond
        const string TEXT = "Hello, Windows Phone!";

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont kootenay14;
        Viewport viewport;
        Vector2 textSize;
        Vector2 textPosition;
        float tCorner;          // height / perimeter
        float tLap;
        float angle;
        …
    }
}
```

The *tLap* variable goes from 0 to 1 as the text makes its way counter-clockwise around the perimeter. To help figure out what side it's currently on, I also define *tCorner*. If *tLap* is less than *tCorner*, the text is on the left edge of the display; if *tLap* is greater than *tCorner* but less than 0.5, it's on the bottom of the display, and so forth. The *LoadContent* method is nothing special:

XNA Project: TextCrawl   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;
    tCorner = 0.5f * viewport.Height / (viewport.Width + viewport.Height);
    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    textSize = kootenay14.MeasureString(TEXT);
}
```

The *Update* method is the real monster, I'm afraid. The objective here is to calculate a *textPosition* and *angle* for the eventual call to *DrawString*.

XNA Project: TextCrawl   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    tLap = (tLap + SPEED * (float)gameTime.ElapsedGameTime.TotalMilliseconds) % 1;

    if (tLap < tCorner)             // down left side of screen
    {
        textPosition.X = 0;
        textPosition.Y = (tLap / tCorner) * viewport.Height;
        angle = -MathHelper.PiOver2;

        if (textPosition.Y < textSize.X)
            angle += (float)Math.Acos(textPosition.Y / textSize.X);
    }
    else if (tLap < 0.5f)           // across bottom of screen
    {
        textPosition.X = ((tLap - tCorner) / (0.5f - tCorner)) * viewport.Width;
        textPosition.Y = viewport.Height;
        angle = MathHelper.Pi;

        if (textPosition.X < textSize.X)
            angle += (float)Math.Acos(textPosition.X / textSize.X);
    }
    else if (tLap < 0.5f + tCorner) // up right side of screen
    {
        textPosition.X = viewport.Width;
        textPosition.Y = (1 - (tLap - 0.5f) / tCorner) * viewport.Height;
        angle = MathHelper.PiOver2;

        if (textPosition.Y + textSize.X > viewport.Height)
            angle += (float)Math.Acos((viewport.Height - textPosition.Y) /
                                                                textSize.X);
    }
    else                            // across top of screen
    {
        textPosition.X = (1 - (tLap - 0.5f - tCorner) /
                                            (0.5f - tCorner)) * viewport.Width;
        textPosition.Y = 0;
        angle = 0;

        if (textPosition.X + textSize.X > viewport.Width)
            angle += (float)Math.Acos((viewport.Width - textPosition.X) /
                                                                textSize.X);
    }

    base.Update(gameTime);
}
```

As I was developing this code, I found it convenient to concentrate on getting the first three statements in each *if* and *else* block working correctly. These statements simply move the upper-left corner of the text string counter-clockwise around the inside perimeter of the display. The initial calculation of *angle* ensures that the top of the text is flush against the edge. Only when I got all that working was I ready to attack the code that alters *angle* for the movement around the corners. A couple simple drawings convinced me that the inverse cosine was the right tool for the job. After all that work in *Update*, the *Draw* method is trivial:

XNA Project: TextCrawl   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, TEXT, textPosition, Color.White,
                           angle, Vector2.Zero, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

In the next chapter you'll see how to make sprites travel along curves.

Chapter 6
Textures and Sprites

I promised that learning how to use XNA to move text around the screen would provide a leg up in the art of moving regular bitmap sprites. This relationship becomes very obvious when you begin examining the *Draw* methods supported by the *SpriteBatch*. The *Draw* methods have almost the same arguments as *DrawString* but work with bitmaps rather than text. In this chapter I'll examine techniques in moving sprites, particularly along curves.

The Draw Variants

Both the *Game* class and the *SpriteBatch* class have methods named *Draw*. Despite the identical names, the two methods are not genealogically related through a class hierarchy. In your class derived from *Game* you override the *Draw* method so that you can call the *Draw* method of *SpriteBatch*. This latter *Draw* method comes in seven different versions. The simplest one is:

```
Draw(Texture2D texture, Vector2 position, Color color)
```

The first argument is a *Texture2D*, which is basically a bitmap. A *Texture2D* is potentially a little more complex than an ordinary bitmap because it could have multiple "mipmap" levels that allow the image to be displayed at a variety of sizes, but for the most part, the *Texture2D* objects that I'll be discussing there are plain old bitmaps. Professional game developers often use specialized tools to create these bitmaps, but I'm going to use Paint because it's readily available. After you create these bitmaps, you add them to the content of the XNA project, and then load them into your program the same way you load a font.

The second argument to *Draw* indicates where the bitmap is to appear on the display. By default, the *position* argument indicates the point on the display where the upper-left corner of the texture is to appear.

The *Color* argument is used a little differently than with *DrawString* because the texture itself can contain color information. The argument is referred to in the documentation as a "color channel modulation," and it serves as a filter through which to view the bitmap.

Conceptually, every pixel in the bitmap has a one-byte red value, a one-byte green value, and a one-byte blue value. When the bitmap is displayed by *Draw*, these red, green, and blue colors values are effectively multiplied by the one-byte red, green, and blue values of the *Color* argument to *Draw*, and the results are divided by 255 to bring them back in the range of 0 to 255. That's what's used to color that pixel.

For example, suppose your texture has lots of color information and you wish all those colors to be preserved on the display. Use a value of *Color.White* in the *Draw* method.

Now suppose you want to draw that same texture but darker. Perhaps the sun is setting in your game world. Use some gray color value in the *Draw* method. The darker the gray, the darker the texture will appear. If you use *Color.Black*, the texture will appear as a silhouette with no color.

Suppose your texture is all white and you wish to display it as blue. Use *Color.Blue* in the *Draw* method. You can display the same all-white texture in a variety of colors. (I'll do precisely that in the first sample program in this chapter.)

If your texture is yellow (a combination of red and green) and you use *Color.Green* in the *Draw* method, it will be displayed as green. If you use *Color.Red* in the *Draw* method it will be displayed as red. If you use *Color.Blue* in the *Draw* method, it will turn black. The argument to *Draw* you can only attenuate or suppress color. You cannot get colors that aren't in the texture to begin with.

The second version of the Draw method is:

```
Draw(Texture2D texture, Rectangle destination, Color color)
```

Instead of a *Vector2* to indicate the position of the texture, you use a *Rectangle*, which is the combination of a point (the upper-left corner), a width, and a height. If the width and height of the *Rectangle* don't match the width and height of the texture, the texture will be scaled to the size of the *Rectangle*.

If you only want to display a rectangular subset of the texture, you can use one of the two slightly expanded versions of the *Draw* method:

```
Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color)
Draw(Texture2D texture, Rectangle destination, Rectangle? source, Color color)
```

The third arguments are nullable *Rectangle* objects. If you set this argument to *null*, the result is the same as using one of the first two versions of *Draw*.

The next two versions of *Draw* have five additional arguments that you'll recognize from the *DrawString* methods:

```
Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color,
     float rotation, Vector2 origin, float scale, SpriteEffects effects, float depth)

Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color,
     float rotation, Vector2 origin, Vector2 scale, SpriteEffects effects, float depth)
```

As with *DrawString*, the *rotation* angle is in radians, measured clockwise. The *origin* is a point in the texture that is to be aligned with the *position* argument. You can scale uniformly with a single *float* or differently in the horizontal and vertical directions with a *Vector2*. The *SpriteEffects* enumeration lets you flip an image horizontally or vertically to get its mirror image. The last argument allows overriding the defaults for layering multiple textures on the screen.

Finally, there's also a slightly shorter longer version where the second argument is a destination rectangle:

```
spriteBatch.Draw (Texture2D texture, Rectangle destination, Rectangle? source, Color color,
                  float rotation, Vector2 origin, SpriteEffects effects, float depth)
```

Notice there's no separate scaling argument because scaling in this one is handled through the *destination* argument.

Within the *Draw* method of your *Game* class, you use the *SpriteBatch* object like so:

```
spriteBatch.Begin();
spriteBatch.Draw …
spriteBatch.End();
```

Within the *Begin* and *End* calls, you can have any number of calls to *Draw* and *DrawString*. The *Draw* calls can reference the same texture. You can also have multiple calls to *Begin* followed by *End* with *Draw* and *DrawString* in between.

Another Hello Program?

If you're tired of "hello, world" programs by now, I've got some bad news. But this time I'll compose a very blocky rendition of the word "HELLO" using two different bitmaps—a vertical bar and a horizontal bar. The letter "H" will be two vertical bars and one horizontal bar. The "O" at the end will look like a rectangle.

And then, when you tap the screen, all 15 bars will fly apart in random directions and then come back together. Sound like fun?

The first step in the FlyAwayHello project is to add content to the Content directory—not a font this time but two bitmaps called HorzBar.png and VertBar.png. You can create these in Paint. By default, Paint creates an all-white bitmap for you. That's ideal! All I want you to do is change the size. Click the Paint Button menu (upper-left below the title bar) and select Properties. Change the size to 45 pixels wide and 5 pixels high. (The exact dimensions really don't matter; the program is coded to be a little flexible.) It's most convenient to save the file right in the Content directory of the project under the name HorzBar.png. Now change the size to 5 pixels wide and 75 pixels high. Save under the name VertBar.png.

Although the bitmaps are now in the proper directory, the XNA project doesn't know of their existence. In Visual Studio, right click the Content directory and choose Add Existing Item. You can select both PNG files and add them to the project.

I'm going to use a little class called *SpriteInfo* to keep track of the 15 textures required for forming the text. If you're creating the project from scratch, right-click the project name, and select Add and then New Item (or select Add New Item from the main Project menu). From the dialog box select Class and give it the name SpriteInfo.cs.

XNA Project: FlyAwayHello   File: SpriteInfo.cs (complete)

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace FlyAwayHello
{
    public class SpriteInfo
    {
        public static float InterpolationFactor { set; get; }

        public Texture2D Texture2D { protected set; get; }
        public Vector2 BasePosition { protected set; get; }
        public Vector2 PositionOffset { set; get; }
        public float MaximumRotation { set; get; }

        public SpriteInfo(Texture2D texture2D, int x, int y)
        {
            Texture2D = texture2D;
            BasePosition = new Vector2(x, y);
        }

        public Vector2 Position
        {
            get
            {
                return BasePosition + InterpolationFactor * PositionOffset;
            }
        }

        public float Rotation
        {
            get
            {
                return InterpolationFactor * MaximumRotation;
            }
        }
    }
}
```

The required constructor stores a *Texture2D* along with positioning information. This is how each sprite is initially positioned to spell out the word "HELLO." Later in the "fly away" animation, the program sets the *PositionOffset* and *MaximumRotation* properties. The *Position* and *Rotation* properties perform calculations based on the static *InterpolationFactor*, which can range from 0 to 1.

Here are the fields of the *Game1* class:

XNA Project: FlyAwayHello   File: Game1.cs (excerpt showing fields)

```
namespace FlyAwayHello
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        static readonly TimeSpan ANIMATION_DURATION = TimeSpan.FromSeconds(5);
        const int CHAR_SPACING = 5;

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Rectangle clientBounds;
        List<SpriteInfo> spriteInfos = new List<SpriteInfo>();
        Random rand = new Random();
        bool isAnimationGoing;
        TimeSpan animationStartTime;
        …
    }
}
```

This program initiates an animation only when the user taps the screen, so I'm handling the timing just a little differently than in earlier programs, as I'll demonstrate in the *Update* method.

The *LoadComponent* method loads the two *Texture2D* objects using the same generic *Load* method that previous programs used to load a *SpriteFont*. Enough information is now available to create and initialize all *SpriteInfo* objects:

XNA Project: FlyAwayHello   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    clientBounds = this.Window.ClientBounds;

    Texture2D horzBar = Content.Load<Texture2D>("HorzBar");
    Texture2D vertBar = Content.Load<Texture2D>("VertBar");
    int x = (viewport.Width - 5 * horzBar.Width - 4 * CHAR_SPACING) / 2;
    int y = (viewport.Height - vertBar.Height) / 2;
    int xRight = horzBar.Width - vertBar.Width;
    int yMiddle = (vertBar.Height - horzBar.Height) / 2;
    int yBottom = vertBar.Height - horzBar.Height;

    // H
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(vertBar, x + xRight, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yMiddle));

    // E
    x += horzBar.Width + CHAR_SPACING;
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yMiddle));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));

    // LL
    for (int i = 0; i < 2; i++)
    {
        x += horzBar.Width + CHAR_SPACING;
        spriteInfos.Add(new SpriteInfo(vertBar, x, y));
        spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));
    }

    // O
    x += horzBar.Width + CHAR_SPACING;
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));
    spriteInfos.Add(new SpriteInfo(vertBar, x + xRight, y));
}
```

The *Update* method is responsible for keeping the animation going. If the *isAnimationGoing* field is *false*, it checks for a new finger pressed on the screen.

XNA Project: FlyAwayHello   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (isAnimationGoing)
    {
        TimeSpan animationTime = gameTime.TotalGameTime - animationStartTime;
        double fractionTime = (double)animationTime.Ticks /
ANIMATION_DURATION.Ticks;
```

```
        if (fractionTime >= 1)
        {
            isAnimationGoing = false;
            fractionTime = 1;
        }

        SpriteInfo.InterpolationFactor =
                          (float)Math.Sin(Math.PI * fractionTime);
    }
    else
    {
        TouchCollection touchCollection = TouchPanel.GetState();
        bool atLeastOneTouchPointPressed = false;

        foreach (TouchLocation touchLocation in touchCollection)
            atLeastOneTouchPointPressed |=
                touchLocation.State == TouchLocationState.Pressed;

        if (atLeastOneTouchPointPressed)
        {
            foreach (SpriteInfo spriteInfo in spriteInfos)
            {
                float r1 = (float)rand.NextDouble() - 0.5f;
                float r2 = (float)rand.NextDouble() - 0.5f;
                float r3 = (float)rand.NextDouble();

                spriteInfo.PositionOffset = new Vector2(r1 * clientBounds.Width,
                                                        r2 * clientBounds.Height);

                spriteInfo.MaximumRotation = 2 * (float)Math.PI * r3;
            }
            animationStartTime = gameTime.TotalGameTime;
            isAnimationGoing = true;
        }
        else if (gameTime.TotalGameTime != TimeSpan.Zero)
        {
            SuppressDraw();
        }
    }
    base.Update(gameTime);
}
```

When the animation begins, the *animationStartTime* is set from the *TotalGameTime* property of *GameTime*. During subsequent calls, *Update* compares that value with the new *TotalGameTime* and calculates an interpolation factor. The *InterpolationFactor* property of *SpriteInfo* is static so it need be set only once to affect all the *SpriteInfo* instances. The *Draw* method loops through the *SpriteInfo* objects to access the *Position* and *Rotation* properties:

XNA Project: FlyAwayHello   File: Game1.cs (excerpt)

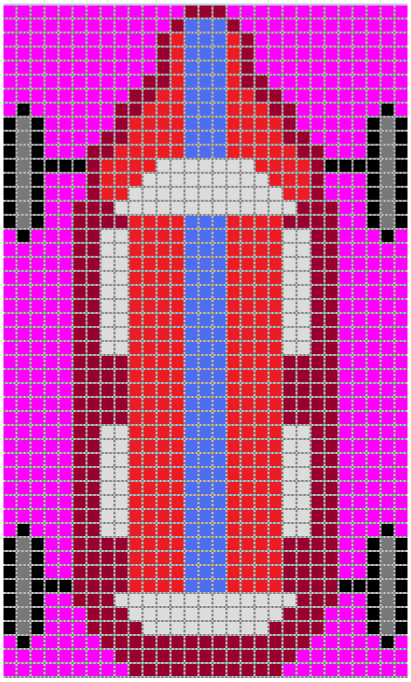
```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);
    spriteBatch.Begin();

    foreach (SpriteInfo spriteInfo in spriteInfos)
    {
        spriteBatch.Draw(spriteInfo.Texture2D, spriteInfo.Position, null,
            Color.Lerp(Color.Blue, Color.Red, SpriteInfo.InterpolationFactor),
            spriteInfo.Rotation, Vector2.Zero, 1, SpriteEffects.None, 0);
    }

    spriteBatch.End();
    base.Draw(gameTime);
}
```

The *Draw* call also uses *SpriteInfo.InterpolationFactor* to interpolate between blue and red for coloring the bars. Notice that the *Color* structure also has a *Lerp* method. The text is normally blue but changes to red as the pieces fly apart.

That call to *Draw* could actually be part of *SpriteInfo*. *SpriteInfo* could define its own *Draw* method with an argument of type *SpriteBatch*, and then pass its own *Texture2D*, *Position*, and *Rotation* properties to the *Draw* method of the *SpriteBatch*.

Driving Around the Block

For the remainder of this chapter I want to focus on techniques to maneuver a sprite around some kind of path. To make it more “realistic,” I commissioned my wife Deirdre to make a little racecar in Paint:



The car is 48 pixels tall and 29 pixels in width. Notice the magenta background: If you want part of an image to be transparent in an XNA scene, you can use a bitmap format that supports transparency, such as the 32-bit Windows BMP format. Each pixel in this format has 8-bit red, green, and blue components but also an 8-bit alpha channel for transparency. (I'll use this format in the next chapter.) The Paint program in Windows, does not support bitmap transparency, alas, so you can use magenta instead. In Paint, create magenta by setting the red and blue values to 255 and green to 0.

In each of the projects in this chapter, this image is stored as the file car.png as part of the project's content. The first project is called CarOnRectangularCourse and demonstrates a rather clunky approach to driving a car around the perimeter of the screen. Here are the fields:

XNA Project: CarOnRectangularCourse File: Game1.cs (excerpt showing fields)

```
namespace CarOnRectangularCourse
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 100f / 1000; // pixels per millisecond
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D car;
        Vector2 carCenter;
        Vector2[] turnPoints = new Vector2[4];
        int sideIndex = 0;
        Vector2 position;
        float rotation;
        …
    }
}
```

The *turnPoints* array stores the four points near the corners of the display where the car makes a sharp turn. Calculating these points is one of the primary activities of the *LoadContent* method, which also loads the *Texture2D* and initializes other fields:

XNA Project: CarOnRectangularCourse   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    float margin = car.Width;
    Rectangle clientBounds = this.Window.ClientBounds;
    turnPoints[0] = new Vector2(clientBounds.Left + margin, clientBounds.Top +
margin);
    turnPoints[1] = new Vector2(clientBounds.Right - margin, clientBounds.Top +
margin);
    turnPoints[2] = new Vector2(clientBounds.Right - margin, clientBounds.Bottom -
margin);
    turnPoints[3] = new Vector2(clientBounds.Left + margin, clientBounds.Bottom -
margin);
    position = turnPoints[0];
    rotation = MathHelper.PiOver2;
}
```

I use the *carCenter* field as the *origin* argument to the *Draw* method, so that it is the point on the car that aligns with a point on the course defined by the four members of the *turnPoints* array. The *margin* value makes this course one car width from the edge of the display; hence the car is really separated from the edge of the display by half its width.

I described this program as "clunky" and the *Update* method proves it:

XNA Project: CarOnRectangularCourse   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
```

```
        this.Exit();

    float pixels = SPEED * (float)gameTime.ElapsedGameTime.TotalMilliseconds;

    switch (sideIndex)
    {
         case 0:         // top
            position.X += pixels;

            if (position.X > turnPoints[1].X)
            {
                position.X = turnPoints[1].X;
                position.Y = turnPoints[1].Y + (position.X - turnPoints[1].X);
                rotation = MathHelper.Pi;
                sideIndex = 1;
            }
            break;

        case 1:         // right
            position.Y += pixels;

            if (position.Y > turnPoints[2].Y)
            {
                position.Y = turnPoints[2].Y;
                position.X = turnPoints[2].X - (position.Y - turnPoints[2].Y);
                rotation = -MathHelper.PiOver2;
                sideIndex = 2;
            }
            break;

        case 2:         // bottom
            position.X -= pixels;

            if (position.X < turnPoints[3].X)
            {
                position.X = turnPoints[3].X;
                position.Y = turnPoints[3].Y + (position.X - turnPoints[3].X);
                rotation = 0;
                sideIndex = 3;
            }
            break;

        case 3:         // left
            position.Y -= pixels;

            if (position.Y < turnPoints[0].Y)
            {
                position.Y = turnPoints[0].Y;
                position.X = turnPoints[0].X - (position.Y - turnPoints[0].Y);
                rotation = MathHelper.PiOver2;
                sideIndex = 0;
            }
            break;
    }
    base.Update(gameTime);
}
```

This is the type of code that screams out “There’s got to be a better way!” Elegant it is not, and not very versatile either. But before I take a stab at a more flexible approach, here’s the entirely predictable *Draw* method that incorporates the updated *position* and *rotation* values calculated during *Update*:

XNA Project: CarOnRectangularCourse   File: Game1.cs (excerpt)

```
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.Blue);

            spriteBatch.Begin();
            spriteBatch.Draw(car, position, null, Color.White, rotation,
                             carCenter, 1, SpriteEffects.None, 0);
            spriteBatch.End();

            base.Draw(gameTime);
        }
```

Movement Along a Polyline

The code in the previous program will work for any rectangle whose corners are stored in the *turnPoints* array, but it won’t work for any arbitrary collection of four points, or more than four points. In computer graphics, a collection of points that describe a series of straight lines is often called a *polyline*, and it would be nice to write some code that makes the car travel around any arbitrary polyline.

The next project, called CarOnPolylineCourse, includes a class named *PolylineInterpolator* that does precisely that. Let me show you the *Game1* class first, and then I’ll describe the *PolylineInterpolator* class that makes this possible. Here are the fields:

XNA Project: CarOnPolylineCourse   File: Game1.cs (excerpt showing fields)

```
namespace CarOnPolylineCourse
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 0.25f / 1000; // laps per millisecond
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D car;
        Vector2 carCenter;
        PolylineInterpolator polylineInterpolator = new PolylineInterpolator();
        Vector2 position;
        float rotation;
        …
    }
}
```

You'll notice a speed in terms of laps, and the instantiation of the mysterious *PolylineInterpolator* class. The *LoadContent* method is very much like that in the previous project except instead of adding points to an array called *turnPoints*, it adds them to a *Vertices* property of the *PolylineInterpolator* class:

XNA Project: CarOnPolylineCourse   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("Car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    float margin = car.Width;
    Rectangle clientBounds = this.Window.ClientBounds;

    polylineInterpolator.Vertices.Add(
        new Vector2(clientBounds.Left + car.Width, clientBounds.Top + car.Width));
    polylineInterpolator.Vertices.Add(
        new Vector2(clientBounds.Right - car.Width, clientBounds.Top + car.Width));
    polylineInterpolator.Vertices.Add(
        new Vector2(clientBounds.Left + car.Width, clientBounds.Bottom - 
car.Width));
    polylineInterpolator.Vertices.Add(
        new Vector2(clientBounds.Right - car.Width, clientBounds.Bottom - 
car.Width));
    polylineInterpolator.Vertices.Add(
        new Vector2(clientBounds.Left + car.Width, clientBounds.Top + car.Width));
}
```

Also notice that the method adds the beginning point in again at the end, and that these points don't exactly describe the same course as the previous project. The previous project caused the car to travel from the upper-left to the upper-right down to lower-right and across to the lower-left and back up to upper-left. The order here goes from upper-left to upper-right but then diagonally down to lower-left and across to lower-right before another diagonal trip up to the beginning. This is precisely the kind of versatility the previous program lacked.

As with the programs in the last chapter that used a parametric-equation approach, the *Update* method is now so simple it makes you want to weep:

XNA Project: CarOnPolylineCourse   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float t = (SPEED * (float)gameTime.TotalGameTime.TotalMilliseconds) % 1;
    float angle;
    position = polylineInterpolator.GetValue(t, false, out angle);
    rotation = angle + MathHelper.PiOver2;

    base.Update(gameTime);
}
```

As usual, *t* is calculated to range from 0 to 1, where 0 indicates the beginning of the course in the upper-left corner of the screen, and *t* approaches 1 as it's heading towards that initial position again. This *t* is passed directly to *GetValue* method of *PolylineInterpolator*, which returns a *Vector2* value somewhere along the polyline.

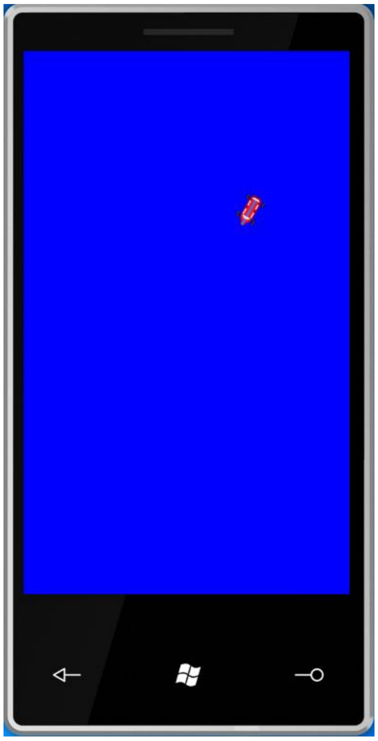
As an extra bonus, the last argument of *GetValue* allows obtaining an *angle* value that is the tangent of the polyline at that point. This angle is measured clockwise relative to the positive X axis. For example, when the car is travelling from the upper-left corner to the upper-right, *angle* is 0. When the car is travelling from the upper-right corner to the lower-left, the angle is somewhere between π/2 and π, depending on the aspect ratio of the screen. The car in the bitmap is facing up so it needs to be rotated an additional π/2 radians.

The *Draw* method is the same as before:

XNA Project: CarOnPolylineCourse   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Here's the car heading towards the lower-left corner:



For demonstration purposes, the *PolylineInterpolator* class sacrifices efficiency for simplicity. Here's the entire class:

XNA Project: **CarOnPolylineCourse**   File: **PolylineInterpolator.cs** (complete)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace CarOnPolylineCourse
{
    public class PolylineInterpolator
    {
        public PolylineInterpolator()
        {
            Vertices = new List<Vector2>();
        }

        public List<Vector2> Vertices { protected set; get; }
        public float TotalLength()
        {
            float totalLength = 0;

            // Notice looping begins at index 1
            for (int i = 1; i < Vertices.Count; i++)
            {
                totalLength += (Vertices[i] - Vertices[i - 1]).Length();
            }
            return totalLength;
        }

        public Vector2 GetValue(float t, bool smooth, out float angle)
        {
            if (Vertices.Count == 0)
            {
                return GetValue(Vector2.Zero, Vector2.Zero, t, smooth, out angle);
            }

            else if (Vertices.Count == 1)
            {
                return GetValue(Vertices[0], Vertices[0], t, smooth, out angle);
            }

            if (Vertices.Count == 2)
            {
                return GetValue(Vertices[0], Vertices[1], t, smooth, out angle);
            }

            // Calculate total length
            float totalLength = TotalLength();
            float accumLength = 0;

            // Notice looping begins at index 1
            for (int i = 1; i < Vertices.Count; i++)
            {
                float prevLength = accumLength;
                accumLength += (Vertices[i] - Vertices[i - 1]).Length();

                if (t >= prevLength / totalLength && t <= accumLength / totalLength)
                {
                    float tPrev = prevLength / totalLength;
                    float tThis = accumLength / totalLength;
                    float tNew = (t - tPrev) / (tThis - tPrev);

                    return GetValue(Vertices[i - 1], Vertices[i], tNew, smooth, out
angle);
                }
            }

            return GetValue(Vector2.Zero, Vector2.Zero, t, smooth, out angle);
        }
        Vector2 GetValue(Vector2 vertex1, Vector2 vertex2, float t,
                         bool smooth, out float angle)
        {

            angle = (float)Math.Atan2(vertex2.Y - vertex1.Y, vertex2.X - vertex1.X);

            return smooth ? Vector2.SmoothStep(vertex1, vertex2, t) :
                            Vector2.Lerp(vertex1, vertex2, t);
        }
    }
}
```

The single *Vertices* property allows you to define a collection of *Vector2* objects that define the polyline. If you want the polyline to end up where it started, you need to explicitly duplicate that point. All the work occurs during the *GetValue* method. At that time, the method determines the total length of the polyline. It then loops through the vertices and accumulates their lengths, finding the pair of vertices whose accumulated length straddles the *t* value. These are passed to the private *GetValue* method to perform the linear interpolation using *Vector2.Lerp*, and to calculate the tangent angle with the graphics programmer's second BFF, *Math.Atan2*.

But wait: There's also a Boolean argument to *GetValue* that causes the method to use *Vector2.SmoothStep* rather than *Vector2.Lerp*. You can try out this alternative by replacing this call in the *Update* method of *Game1*:

```
position = polylineInterpolator.GetValue(t, false, out angle);
```

with this one:

```
position = polylineInterpolator.GetValue(t, true, out angle);
```

The "smooth step" interpolation is based on a cubic, and causes the car to slow down as it approaches one of the vertices, and speed up afterwards. It still makes an abrupt and unrealistic turn but the speed change is quite nice.

What I don't like about the *PolylineInterpolator* class is its inefficiency. *GetValue* needs to make several calls to the *Length* method of *Vector2*, which of course involves a square-root calculation. It would be nice for the class to retain the total length and the accumulated length at each vertex so it could simply re-use that information on successive *GetValue* calls. As written, the class can't do that because it has no knowledge when *Vector2* values are added to or removed from the *Vertices* collection. One possibility is to make that collection private, and to only allow a collection of points to be submitted in the class's constructor. Another approach is to replace the *List* with an *ObservableCollection*, which provides an event notification when objects are added and removed.

The Elliptical Course

The most unrealistic behavior of the previous program involves the turns. Cars slow down to turn around corners, but they actually travel along a curved path to change direction. To really make the previous program realistic, the corners would have to be replaced by curves. These curves could be approximated with polylines, but the increasing number of polylines would then require *PolylineInterpolator* to be restructured for better performance.

Instead, I'm going to go off on a somewhat different tangent and drive the car around a traditional *oval* course, or to express it more mathematically, an *elliptical* course.

Let's look at some math. A circle centered on the point (0, 0) with a radius of *R* consists of all points (*x*, *y*) where

$$x^2 + y^2 = R^2$$

An ellipse has two radii. If these are parallel to the horizontal and vertical axes, they are sometimes called R_x and R_y, and the ellipse formula is:

$$\left(\frac{x}{R_x}\right)^2 + \left(\frac{y}{R_y}\right)^2 = 1$$

For our purposes, it is more convenient to represent the ellipse in the parametric form. In these two equations, *x* and *y* are functions of the angle *α*, which ranges from 0 to 2π:

$$x = R_x \cos \propto$$
$$y = R_y \sin \propto$$

When the ellipse is centered around the point (*Cx*, *Cy*), the formulas become:

$$x = C_x + R_x \cos \propto$$
$$y = C_y + R_y \sin \propto$$

If we also want to introduce a variable *t*, where *t* goes from 0 to 1, the formulas are:

$$x(t) = C_x + R_x \cos(2\pi t)$$
$$y(t) = C_y + R_y \sin(2\pi t)$$

And these will be ideal for our purpose. As *t* goes from 0 to 1, the car goes around the lap once. But how do we rotate the car so it appears to be travelling in a tangent to this ellipse? For that job, the differential calculus comes to the rescue. First, take the derivatives of the parametric equations:

$$x'(t) = -R_x \sin(2\pi t)$$
$$y'(t) = R_y \cos(2\pi t)$$

In physical terms, these equations represent the instantaneous change in direction in the X direction and Y direction, respectively. To turn that into a tangent angle, simply apply *Math.Atan2*.

And now we're ready to code. Here are the fields:

XNA Project: CarOnOvalCourse   File: Game1.cs (excerpt showing fields)

```
namespace CarOnOvalCourse
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 0.25f / 1000;   // laps per millisecond
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D car;
        Vector2 carCenter;
        Point ellipseCenter;
        float ellipseRadiusX, ellipseRadiusY;
        Vector2 position;
        float rotation;
        …
    }
}
```

The fields include the three items required for the parametric equations for the ellipse: the center and the two radii. These are determined during the *LoadContent* method based on the dimensions of the available area of the screen:

XNA Project: CarOnOvalCourse   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    Rectangle clientBounds = this.Window.ClientBounds;
    ellipseCenter = clientBounds.Center;
    ellipseRadiusX = clientBounds.Width / 2 - car.Width;
    ellipseRadiusY = clientBounds.Height / 2 - car.Width;
}
```

Notice that the *Update* method below calculates two angles. The first, called *ellipseAngle*, is based on *t* and determines where on the ellipse the car is located. This is the angle passed to the parametric equations for the ellipse, to obtain the position as a combination of *x* and *y*:

XNA Project: CarOnOvalCourse File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float t = (SPEED * (float)gameTime.TotalGameTime.TotalMilliseconds) % 1;
    float ellipseAngle = MathHelper.TwoPi * t;
    float x = ellipseCenter.X + ellipseRadiusX * (float)Math.Cos(ellipseAngle);
    float y = ellipseCenter.Y + ellipseRadiusY * (float)Math.Sin(ellipseAngle);
    position = new Vector2(x, y);

    float dxdt = -ellipseRadiusX * (float)Math.Sin(ellipseAngle);
    float dydt = ellipseRadiusY * (float)Math.Cos(ellipseAngle);
    rotation = MathHelper.PiOver2 + (float)Math.Atan2(dydt, dxdt);

    base.Update(gameTime);
}
```

The second angle that *Update* calculates is called *rotation*. This is the angle that determines the orientation of the car. The *dxdt* and *dydt* variables are the derivatives of the parametric equations that I showed earlier. The *Math.Atan2* method provides the rotation angle relative to the positive X axis, and this must be rotated another 90 degrees for the original orientation of the bitmap.

By this time, you can probably recite *Draw* by heart:

XNA Project: CarOnOvalCourse File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

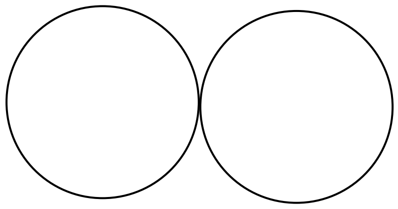
A Generalized Curve Solution

For movement along curves that are not quite convenient to express in parametric equations, XNA itself provides a generalized solution that involves the *Curve* and *CurveKey* classes defined in the *Microsoft.Xna.Framework* namespace.

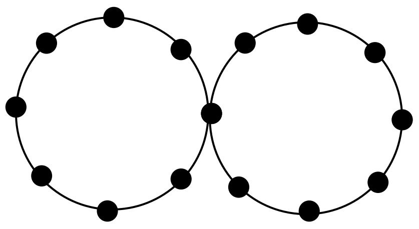
The *Curve* class contains a property named *Keys* of type *CurveKeyCollection*, a collection of *CurveKey* objects. Each *CurveKey* object allows you to specify a number pair of the form (*Position*, *Value*). Both the *Position* and *Value* properties are of type *float*. Then you pass a position to the *Curve* method *Evaluate*, and it returns an interpolated value.

But it's all rather confusing because—as the documentation indicates—the *Position* property of *CurveKey* is almost always a *time*, and the *Value* property is very often a *position*, or more accurately, one *coordinate* of a position. If you want to use *Curve* to interpolate between points in two-dimensional space, you need two instances of *Curve*—one for the X coordinate and the other for Y. These *Curve* instances are treated very much like parametric equations.

Suppose you want the car to go around a path that looks like an infinity sign, and let's assume that we're going to approximate the infinity sign with two adjacent circles. (The technique I'm going to show you will allow you to move those two circles apart at a later time if you'd like.)



Draw dots every 45 degrees on these two circles:



If the radius of each circle is 1 unit, the entire figure is 4 units wide and 2 units tall. The X coordinates of these dots (going from left to right) are the values 0, 0..293, 1, 0.707, 2, 2.293, 3, 3.707, and 4, and the Y coordinates (going from top to bottom) are the values 0, 0.293, 1, 1.707, and 2. The value 0.707 is simply the sine and cosine of 45 degrees, and 0.293 is one minus that value.

Let's begin at the point on the far left, and let's travel clockwise around the first circle. At the center of the figure, let's switch to going counter-clockwise around the second circle (because we really want an infinity sign) and finish with the same dot we started with. The X values are:

0, 0.293, 1, 1.707, 2, 2.293, 3, 3.707, 4, 3.707, 3, 2.293, 2, 1.707, 1, 0.293, 0

If we're using values of *t* ranging from 0 to 1 to drive around the infinity sign, then the first value corresponds to a *t* of 0, and the last (which is the same) to a *t* of 1. For each value, *t* is incremented by 1/16 or 0.0625. The Y values are:

1, 0.293, 0, 0.293, 1, 1.707, 2, 1.707, 1, 0.293, 0, 0.293, 1, 1.707, 2, 1.707, 1

We are now ready for some coding. Here are the fields for the CarOnInfinityCourse project:

XNA Project: CarOnInfinityCourse File: Game1.cs (excerpt showing fields)

```
namespace CarOnInfinityCourse
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        const float SPEED = 0.1f / 1000; // laps per millisecond
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Rectangle clientBounds;
        Texture2D car;
        Vector2 carCenter;
        Curve xCurve = new Curve();
        Curve yCurve = new Curve();
        Vector2 position;
        float rotation;
        …
    }
}
```

Notice the two *Curve* objects, one for X coordinates and the other for Y. Because the initialization of these objects use precisely the coordinates I described above and don't require accessing any resources or program content, I decided to use the *Initialize* override for this work.

XNA Project: CarOnInfinityCourse   File: Game1.cs (excerpt)

```
protected override void Initialize()
{
    float[] xValues = { 0, 0.293f, 1, 1.707f, 2, 2.293f, 3, 3.707f,
                        4, 3.707f, 3, 2.293f, 2, 1.707f, 1, 0.293f };
    float[] yValues = { 1, 0.293f, 0, 0.293f, 1, 1.707f, 2, 1.707f,
                        1, 0.293f, 0, 0.293f, 1, 1.707f, 2, 1.707f };

    for (int i = -1; i < 18; i++)
    {
        int index = (i + 16) % 16;
        float t = 0.0625f * i;
        xCurve.Keys.Add(new CurveKey(t, xValues[index]));
        yCurve.Keys.Add(new CurveKey(t, yValues[index]));
    }
    xCurve.ComputeTangents(CurveTangent.Smooth);
    yCurve.ComputeTangents(CurveTangent.Smooth);

    clientBounds = this.Window.ClientBounds;
    base.Initialize();
}
```

The *xValues* and *yValues* arrays only have 16 values; they don't include the last point that duplicates the first. Very oddly, the *for* loop goes from –1 through 17 but the modulo 16 operation ensures that the arrays are indexed from 0 through 15. The end result is that the *Keys* collections of *xCurve* and *yCurve* get coordinates associated with *t* values of –0.0625, 0, 0.0625, 0.0125, …, 0.875, 0.9375, 1, and 1.0625, which are apparently two more points than is necessary to make this thing work right.

These extra points are necessary for the *ComputeTangents* calls following the *for* loop. The *Curve* class performs a type of interpolation called a cubic Hermite spline, also called a *cspline*. Consider two points *pt1* and *pt2*. The cspline interpolates between these two points based not only on *pt1* and *pt2* but also on assumed tangents of the curve at *pt1* and *pt2*. You can specify these tangents to the *Curve* object yourself as part of the *CurveKeys* objects, or you can have the *Curve* object calculate tangents for you based on adjoining points. That is the approach I've taken by the two calls to *ComputeTangents*. With an argument of *CurveTangent.Smooth*, the *ComputeTangents* method uses not only the two adjacent points, but the points on either side. It's really just a simple weighted average but it's better than the alternatives.

The *Curve* and *CurveKey* classes have several other options, but the approach I've taken seemed to offer the best results with the least amount of work.

The *Initialize* method ends by obtaining the *ClientBounds* property, leaving the *LoadContent* method with very little to do:

XNA Project: CarOnInfinityCourse   File: Game1.cs (excerpt)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("Car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
}
```

Now it's time for *Update*. The method calculates *t* based on *TotalGameTime*. The *Curve* class defines a method named *Evaluate* that can accept this *t* value directly; this is how the program obtains interpolated X and Y coordinates. However, all the data in the two *Curve* objects are based on a maximum X coordinate of 4 and a Y coordinate of 2. For this reason, *Update* calls a little method I've supplied named *GetValue* that scales the values based on the size of the display and whether the display is in portrait or landscape mode.

XNA Project: CarOnInfinityCourse   File: Game1.cs (excerpt)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float t = (SPEED * (float)gameTime.TotalGameTime.TotalMilliseconds) % 1;
    float x = GetValue(t, true);
    float y = GetValue(t, false);
    position = new Vector2(x, y);

    rotation = MathHelper.PiOver2 + (float)
        Math.Atan2(GetValue(t + 0.001f, false) - GetValue(t - 0.001f, false),
                   GetValue(t + 0.001f, true) - GetValue(t - 0.001f, true));

    base.Update(gameTime);
}

float GetValue(float t, bool isX)
{
    bool isLandscape = clientBounds.Width > clientBounds.Height;

    if (isX == isLandscape)
        return xCurve.Evaluate(t) * (clientBounds.Height - 2 * car.Width) / 4 +
                                    clientBounds.X + car.Width;

    return yCurve.Evaluate(t) * (clientBounds.Width - 2 * car.Width) / 2 +
                                    clientBounds.Y + car.Width;
}
```

After calculating the *position* field, we have a little bit of a problem because the *Curve* class is missing an essential method: the method that provides the tangent of the spline. Tangents are required by the *Curve* class to *calculate* the spline, but after the spline is calculated, the class doesn't provide access to the tangents of the spline itself!

That's the purpose of the other four calls to *GetValue*. Small values are added to and subtracted from *t* to approximate the derivative and allow *Math.Atan2* to calculate the *rotation* angle.

Once again, *Draw* is trivial:

XNA Project: CarOnInfinityCourse   File: Game1.cs (excerpt)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

If you want the *Curve* class to calculate the tangents used for calculating the spline (as I did in this program) it is essential to give the class sufficient points, not only beyond the range of points you wish to interpolate between, but enough so that these calculated tangents are more or less accurate. I originally tried defining the infinity course with points on the two circles every 90 degrees, and it didn't well work at all.