SPECIAL EXCERPT 2

Microsoft

PREVIEW CONTENT

Complete book available Fall 2010

Programming Windows® Phone 7

Charles Petzold

PREVIEW CONTENT

This excerpt provides early content from a book currently in development, and is still in draft, unedited format. See additional notice below.

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, Microsoft Press, Azure, Expression, Expression Blend, Internet Explorer, MS, Silverlight, Visual C#, Visual Studio, Webdings, Windows, Windows Azure, Windows Live, Windows Mobile, Xbox, Xbox 360, XNA, and Zune are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Introduction

This is the second "draft preview" of a longer ebook that will be completed and published later this year. That final 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.

The first "draft preview" of this book was created in time for the Microsoft MIX conference in March 2010. This one is issued in conjunction with VSLIve in Redmond on August 2–6.

Even with this book's defects and limited scope, I hope it helps get you started in writing great programs for Windows Phone 7. 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. Get the latest information at http://developer.windowsphone.com.

You can install Visual Studio 2010 Express for Windows Phone on top of Visual Studio 2010, in effect enhancing Visual Studio 2010 for phone development. That's the configuration I used.

When finalizing these eleven chapters, I've been working with the beta release of the Windows Phone Developer Tools dated July 26. I have not yet been able to test my programs on an actual Windows Phone 7 device.

Windows Phone 7 supports multi-touch, and working with multi-touch is an important part of developing programs for the phone. When using the Windows Phone Emulator, mouse clicks and mouse movement on the PC can mimic touch on the emulator. You can test out multi-touch for real on the phone emulator if you have a multi-touch display running under Windows 7.

The Essential People

This book owes its existence to Dave Edson — an old friend from the early 90's era of *Microsoft Systems Journal*— who had the brilliant idea that I would be the perfect person to write a tutorial on Windows Phone 7. Dave arranged for me to attend a technical deep dive on the phone at Microsoft 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 project and Devon Musgrave had the unenviable job of trying to make my code and prose resemble an actual book. (We all go way back: You'll see Ben and Devon's names on the bottom of the copyright page of *Programming Windows*, fifth edition, published in 1998.)

Dave Edson also reviewed chapters and served as conduit to the Windows Phone team to deal with my technical problems and questions. For the first draft preview, 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 to Bonnie Lehenbauer for reviewing one of the chapters at the last minute.

For this second draft preview, I am indebted to Shawn Hargreaves for his XNA expertise, and Yochay Kiriaty and Richard Bailey for the lowdown on tombstoning.

My wife Deirdre Sinnott has been a marvel of patience and tolerance over the past 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
Roscoe, NY
July 27, 2010

Part I

The Basics

Chapter 1

Hello, Windows Phone 7

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. With its clean look, striking fonts, and new organizational paradigms, Microsoft Windows Phone 7 not only represents a break with the Windows Mobile past but also differentiates itself from other smartphones currently in the market.

For programmers, Windows Phone 7 is also exciting, for it supports two popular and modern programming platforms: Silverlight and XNA.

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. Windows Phone 7 extends Silverlight to mobile devices.

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 use XNA to target the PC itself, as well as Microsoft's classy audio player, the Zune HD.

Either Silverlight or XNA would make good sense as the sole application platform for the Windows Phone 7, but programmers have a choice. And this we call "an embarrassment of riches."

Targeting Windows Phone 7

All programs for Windows Phone 7 are written in .NET managed code. At the present time, C# is the only supported programming language. The free downloadable Microsoft Visual Studio 2010 Express for Windows Phone includes XNA Game Studio 4.0 and an on-screen phone emulator, and also integrates with Visual Studio 2010. You can develop visuals and animations for Silverlight applications using Microsoft Expression Blend.

The Silverlight and XNA platforms for Windows Phone 7 share some libraries, and you can use some XNA libraries in a Silverlight program and vice versa. But you can't create a program that mixes visuals from both platforms. Maybe that will be possible in the future, but not now. 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 and logic, 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. Silverlight for Windows Phone is a version of Silverlight 3 excluding some features not appropriate for the phone, but compensating with some enhancements.

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-in 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 use vector graphics rather than bitmap graphics, or 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.

The first several chapters in this book describe Silverlight and XNA together, and then the book splits into different parts for the two platforms. 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!

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 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.

One of 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 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.

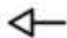
I've characterized Windows Phone 7 as representing a severe break with the past. If you compare it with past versions of Windows Mobile, that is certainly true. But the support of Silverlight, XNA, and C# are not breaks with the past, but a balance of continuity and innovation. As young as they are, 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 programs for other .NET environments.

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. Microsoft has been extremely proactive in defining the hardware specification, often referred to as a "chassis."

Initial releases of Windows Phone 7 devices will have one consistent screen size. (A second screen size is expected in the future.) Many other hardware features are guaranteed to exist on each device.

The front of the phone consists of a multi-touch 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:



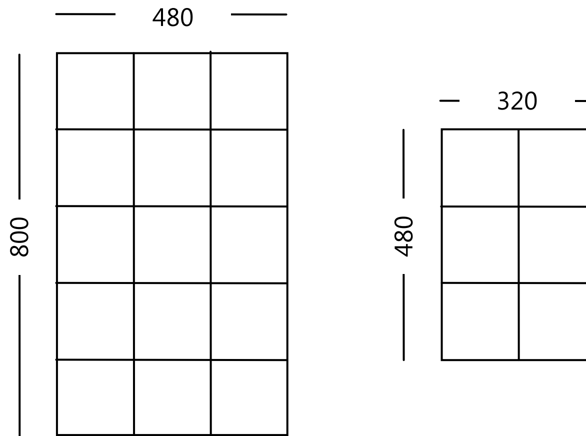
- **Back** Programs can use this button for their own navigation needs, much like the Back button on a Web browser. From the home page of a program, the button causes the program to terminate.
- **Start** This button takes the user to the start screen of the phone; it is otherwise inaccessible to programs running on the phone.

- **Search** The operating system uses this button to initiate a search feature.

The initial releases of Windows Phone 7 devices have a display size of 480 × 800 pixels. In the future, screens of 320 × 480 pixels are also expected. There are no other screen options for Windows Phone 7, 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 conditional code paths 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. Some programs might require the phone to be held in a certain orientation; others might be more adaptable.

You have complete control over the extent to which you support orientation. By default, Silverlight applications appear in portrait mode, but you'll probably want to write your Silverlight applications so they adjust themselves to orientation changes. 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. XNA programs use landscape mode by default, but it's easy to override that.

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 displays will likely use OLED ("organic light emitting diode") technology, although this isn't a hardware requirement. OLEDs are different from flat displays of the past in that power consumption is 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.

Because battery life is extremely important on mobile devices, this characteristic of OLED displays implies an aesthetic of mostly black backgrounds with sparse graphics and light-stroked fonts. Regardless, Windows Phone 7 users can choose between two major color themes: light text on a dark background, or dark text on a light background.

Most user input to a Windows Phone 7 program will come through multi-touch. The screens incorporate capacitance-touch technology, which means that they respond to a human fingertip but not to a stylus or other forms of pressure. Windows Phone 7 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 changes 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

A Windows Phone 7 device is required to contain several other hardware features—sometimes called sensors—and provide some software services, perhaps through the assistance of hardware. These are the ones that affect developers the most:

- **Wi-Fi** The phone has Wi-Fi for Internet access. Software on the phone includes a version of Internet Explorer.
- **Camera** The phone has at least a 5-megapixel camera with flash. Programs can invoke the camera program for their own input, or 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 gravity. Programs can obtain a three-dimensional vector that indicates how the camera is oriented with respect to the earth. The accelerometer can also detect sharp movements of the phone.
- **Location** If the user so desires, the phone can use multiple strategies for determining where it is geographically located. The phone supplements a hardware GPS device with information from the Web or cell phone towers. If the phone is moving, course and speed might also be available. An internal compass helps the device determine this information, although the compass is not inaccessible to application programs.
- **Vibration** The phone can be vibrated through program control.
- **FM Radio** An FM Radio is available accessible 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 there's more: Although I haven't been able to confirm this, a persistent rumor indicates that a Windows Phone 7 device can also be used to make and receive telephone calls.

File | New | Project

I'll assume that you have Visual Studio 2010 Express for Windows Phone installed, either by itself or supplementing a regular version of Visual Studio 2010. For convenience, I'm going to refer to this development environment simply as "Visual Studio."

The traditional "hello, world" program that displays just 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 a 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 7!"

Just to make these programs a little more interesting, I want to display the text in the center of the display. The Silverlight program will use the background and foreground colors selected by the user in the Themes section of the phone's Settings screen. In the XNA program, the text will be white on a dark background to use less power on OLED.

If you're playing along, it's time to bring up Visual Studio and from the File menu select New and then Project.

A First Silverlight Phone Program

In the New Project 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.

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 code and markup.

Several files have been created for this SilverlightHelloPhone project and 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 you selected. Insert some spaces to make it a little friendlier:

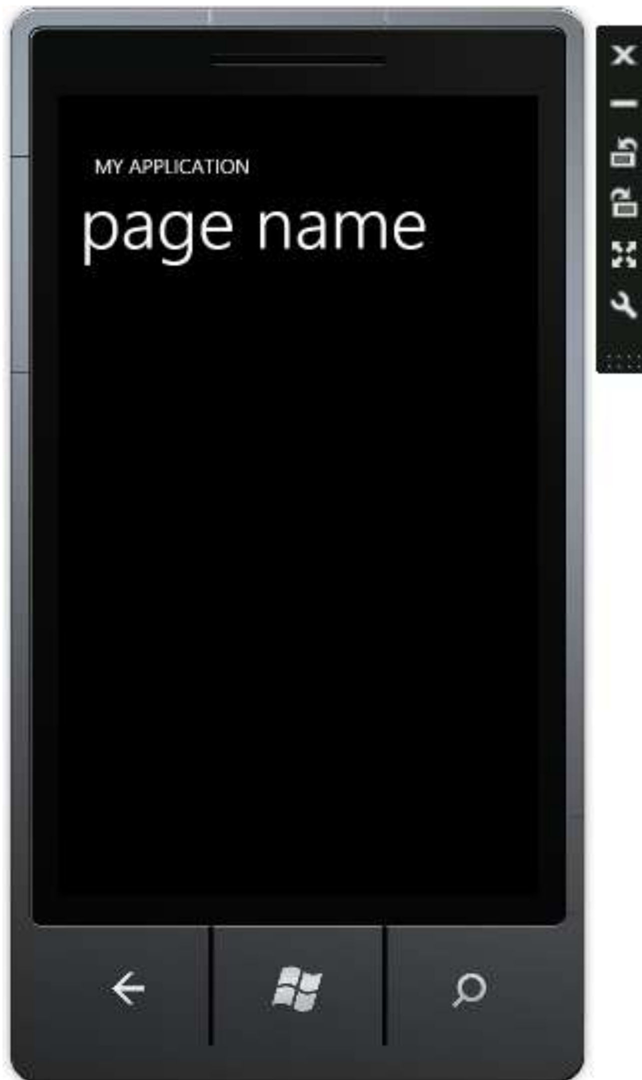
```
Title="Silverlight Hello Phone"
```

This is the name used by the phone and the phone emulator to display the program in the list of installed applications presented to the user. If you're really ambitious, you can also edit the ApplicationIcon.png and Background.png files that the phone uses to visually symbolize the program. The SplashScreenImage.jpg file is what the program displays as it's initializing.

In the standard Visual Studio 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, it might take a little time to start up. If you leave the emulator running between edit/build/run cycles, Visual Studio 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.



The phone emulator has a little floating menu at the upper right that comes into view when you move the mouse to that location. You can change orientation through this menu, or change the emulator size. By default, the emulator is displayed at 50% actual size, about the same size as the image on this page. When you display the emulator at 100%, it becomes enormous, and you might wonder "How will I ever fit a phone this big into my pocket?"

The difference involves pixel density. Your computer screen probably has about 100 pixels per inch. (By default, Windows assumes that screens are 96 DPI.) The screen on an actual Windows Phone 7 device is about double that. When you display the emulator at 100%, you're seeing all the pixels of the phone's screen, but at about twice their actual size.

You can terminate execution of this program and return to editing the program either though Visual Studio (using Shift-F5 or by selecting Stop Debugging from the Debug menu) or by by clicking the Back button on the emulator.

Don't exit the emulator itself by clicking the X at the top of the floating menu! Keeping the emulator running will make subsequent deployments go much faster.

While the emulator is still running, it retains all Silverlight programs deployed to it, but not XNA programs. (Actually, the difference involves the genre of the program as indicated by the Genre attribute in the App tag of the WMAppManifest.xml file.) 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" and you can run the program again. The program will disappear from this list when you exit the emulator.

Back in Visual Studio, take a look at the Solution Explorer for the project. You'll see 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. This is a basic Silverlight concept.

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 the project should probably include another C# file that contains additional members of the *App* class. But where is it?

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"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone">
    …
</Application>
```

You'll recognize this file as XML, but more precisely it is a XAML file, which is an important part of Silverlight programming. In particular, developers often use the App.xaml file for storing *resources* that are used throughout the application. These resources might include color schemes, gradient brushes, styles, and so forth.

The root element is *Application*, which is the Silverlight class that the *App* class derives from. The root element contains four XML namespace declarations. Two are common in all Silverlight applications; two are unique to the phone.

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* itself. 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 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 particular XAML 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 you saw in the App.xaml.cs file but with very 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.

You're free to edit the App.xaml and App.xaml.cs files, but don't mess around with App.g.cs. That file is recreated when you build the project.

When a program is run, the *App* class creates an object of type *PhoneApplicationFrame* and sets that object to its own *RootVisual* property. This frame is 480 pixels wide and 800 pixels tall and occupies the entire display surface of the phone. The *PhoneApplicationFrame* object then behaves somewhat like a web browser by navigating to an object called *MainPage*.

MainPage is the second major class in every Silverlight 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)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;

namespace SilverlightHelloPhone
{
    public partial class MainPage : PhoneApplicationPage
    {
        // Constructor
        public MainPage()
        {
            InitializeComponent();
        }
    }
```

```
}
```

The *using* directives for namespaces that begin with the words *System.Windows* are for the Silverlight classes; sometimes you'll need to supplement these with some other *using* directives as well. The *Microsoft.Windows.Controls* namespace contains extensions to Silverlight for the phone, including the *PhoneApplicationPage* class.

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. Here's the nearly complete file, reformatted a bit to fit the printed page, and excluding a section that's commented out at the end, but still a rather frightening chunk of markup:

Silverlight Project: SilverlightHelloPhone File: MainPage.xaml (almost complete)

```
<phone:PhoneApplicationPage
    x:Class="SilverlightHelloPhone.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    shell:SystemTray.IsVisible="True">

    <!--LayoutRoot contains the root grid where all other page content is placed-->
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!--TitlePanel contains the name of the application and page title-->
        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="24,24,0,12">
            <TextBlock x:Name="ApplicationTitle" Text="SILVERLIGHT HELLO PHONE"
                       Style="{StaticResource PhoneTextNormalStyle}"/>
            <TextBlock x:Name="PageTitle" Text="main page" Margin="-3,-8,0,0"
                       Style="{StaticResource PhoneTextTitle1Style}"/>
        </StackPanel>

        <!--ContentPanel - place additional content here-->
        <Grid x:Name="ContentGrid" Grid.Row="1">
```

```
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

These first four 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.

The "d" (for "designer") and "mc" (for "markup compatibility") namespace declarations are for the benefit of XAML design programs, such as Expression Blend and the designer in Visual Studio itself. The *DesignerWidth* and *DesignerHeight* attributes are ignored during compilation.

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) with 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 into 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.

In the root element of MainPage.xaml you'll see settings for *FontFamily*, *FontSize*, and *Foreground* that apply to the whole page. I'll describe the *StaticResource* and this syntax in Chapter 7.

The body of the MainPage.xaml file contains several nested elements named *Grid*, *StackPanel*, and *TextBlock* in a parent-child hierarchy.

Notice the word I used: *element*. In Silverlight programming, this word has two related meanings. It's an XML term used to indicate items delimited by start tags and end tags. But it's also a word used in Silverlight to refer to visual objects, and in fact, the word *element* shows up in the names of two actual Silverlight classes.

Many of the classes you use in Silverlight are part of this important class hierarchy:

Object
        DependencyObject (abstract)
                UIElement (abstract)
                        FrameworkElement (abstract)

Besides *UIElement*, many other Silverlight classes derive from *DependencyObject*. But *UIElement* has the distinction of being the class that has the power to appear as a visual object on the screen and to receive user input. (In Silverlight, all visual objects can receive user input.) Traditionally, this user input comes from the keyboard and mouse; on the phone, most user input comes from touch.

The only class that derives from *UIElement* is *FrameworkElement*. The distinction between these two classes is a historical artifact of the Windows Presentation Foundation. In WPF, it is possible for developers to create their own unique frameworks by deriving from *UIElement*. In Silverlight this is not possible, so the distinction is fairly meaningless.

One of the classes that derives from *FrameworkElement* is *Control*, a word more common in graphical user-interface programming. Some objects commonly referred to as *controls* in other programming environments are more correctly referred to as *elements* in Silverlight. Control derivatives include buttons and sliders that I'll discuss in Chapter 10.

Another class that derives from *FrameworkElement* is *Panel*, which is the parent class to the *Grid* and *StackPanel* elements you see in MainPage.xaml. Panels are elements that can host multiple children and arrange them in particular ways on the screen. I'll discuss panels in more depth in Chapter 9, and I'll cover the phone-specific *PanoramaPanel* in connection with navigation.

Another class that derives from *FrameworkElement* is *TextBlock*, the element you'll use most often in displaying blocks of text up to about a paragraph in length. The two *TextBlock* elements in MainPage.xaml display the two chunks of title text in a new Silverlight program.

PhoneApplicationPage, *Grid*, *StackPanel*, and *TextBlock* are all Silverlight classes. In Markup these become XML elements. Properties of these classes become XML attributes.

The nesting of elements in MainPage.xaml is said to define a *visual tree*. In a Silverlight program for Windows Phone 7, the visual tree always begins with an object of type *PhoneApplicationFrame*, which occupies the entire visual surface of the phone. A Silverlight program for Windows Phone 7 always has one and only one instance of *PhoneApplicationFrame*, referred to informally as the *frame*.

In contrast, a program can have multiple instances of *PhoneApplicationPage*, referred to informally as a *page*. At any one time, the frame hosts one page, and lets you navigate to the other pages. By default, the page does not occupy the full display surface of the frame because it makes room for the system tray (also known as the status bar) at the top of the phone.

Our simple application has only one page, appropriately called *MainPage*. This *MainPage* contains a *Grid*, which contains a *StackPanel* with a couple *TextBlock* elements, and another *Grid*, all in a hierarchical tree.

Our original goal was to create a Silverlight program that displays some text in the center of the display, but given the presence of a couple titles, let's amend that goal to displaying the text in the center of the page apart from the titles. The area of the page for program content is the *Grid* towards the bottom of the file preceded by the comment "ContentPanel - place additional content here." This *Grid* has a name of "ContentGrid" and I'm going to refer to it informally as the "content grid."

In the content grid, you can insert a new *TextBlock*:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Text, *HorizontalAlignment*, and *VerticalAlignment* are all properties of the *TextBlock* class. The *Text* property is of type *string*. The *HorizontalAlignment* and *VerticalAlignment* properties are of numeration types HorizontalAliignment and *VerticalAlignment*, respectively. When you reference an enumeration type in XAML, you only need the member name.

While you're editing MainPage.xaml you might also want to fix the other *TextBlock* elements so that they aren't so generic. 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. You can now compile and run this program:

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 a visual tree. These elements are capable of arranging themselves dynamically. The *HorizontalAlignment* and *VerticalAlignment* properties can put an element in the center of another element, or (as you might suppose) 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.

Color Themes

From the Start screen of the phone emulator, click or touch the right arrow at the upper right and navigate to the Settings page. You'll see that you can select a visual theme: Either Dark (light text on a dark background, which you've been seeing) or Light (the opposite). Select the Light theme, run SilverlightHelloPhone again, and express some satisfaction that the theme colors are automatically applied:

It is possible to override these theme colors. 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*.

In a real-world program, you'll want to test out any custom colors with the available themes so text doesn't mysteriously disappear or becomes hard to read.

Points and Pixels

Another property of the *TextBlock* that you can easily change is *FontSize*:

```
FontSize="36"
```

But what exactly does this mean?

All dimensions in Silverlight are in units of pixels, and the *FontSize* is no exception. When you specify 36, you get a font that from the top of its ascenders to the bottom of its descenders measures approximately 36 pixels.

Traditionally, font sizes are expressed in units of *points*. In classical typography, a point is very close to 1/72nd inch but in digital typography the point 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 point size indicates 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.)

How do you convert between pixels and points? Obviously you can't except for a particular output device. On a 600 dots-per-inch (DPI) printer, for example, the 72-point font will be 600 pixels tall.

Desktop video displays in common use today usually have a resolution somewhere in the region of 100 DPI. For example, consider a 21" monitor that displays 1600 pixels horizontally and 1200 pixels vertically. That's 2000 pixels diagonally, which divided by 21" is about 95 DPI.

By default, Microsoft Windows assumes that video displays have a resolution of 96 DPI. Under that assumption, font sizes and pixels are related by the following formulas:

points = ¾ × pixels

pixels = 4/3 × points

This relationship applies only to common video displays, but people so much enjoy having these conversion formulas, they show up in Windows Phone 7 programming as well.

So, when you set a *FontSize* property such as

```
FontSize="36"
```

you can also claim to be setting a 27-point font. But the resultant *TextBlock* will actually have a height more like 48 pixels — about 33% higher than the *FontSize* would imply (which is 33% more than the point size). This additional space (called *leading*) prevents successive lines of text from jamming against each other.

The issue of font size becomes more complex when dealing with high-resolution screens found on devices such as Windows Phone 7. The 480 × 800 pixel display has a diagonal of 933 pixels in about 4½" for a pixel density closer to 200 DPI — about double the resolution of conventional video displays.

This doesn't necessarily mean that all the font sizes used on a conventional screen need to be doubled on the phone. The higher resolution of the phone — and the closer viewing distance — allows smaller font sizes to be more readable.

When running in a Web browser, the default Silverlight *FontSize* is 11 pixels, corresponding to a font size of 8.25 points, with is fine for a desktop video display but a little too small for the phone. For that reason, Silverlight for Windows Phone defines a collection of common font sizes that you can use. (I'll describe how these work in Chapter 7.) The standard MainPage.xaml file includes the following attribute in the root element:

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

This *FontSize* is inherited through the visual tree and applies to all *TextBlock* elements that don't set their own *FontSize* properties. It has a value of 20 pixels — almost double the default Silverlight *FontSize* on the desktop. Using the standard formulas, this 20-pixel *FontSize* corresponds to 15 points, but as actually displayed on the phone, it's about half the size that a 15-point font would appear in printed text.

The actual height of the *TextBlock* displaying text with this font is about 33% more than the *FontSize*, in this case about 27 pixels.

An XNA Program for the Phone

Next up on the agenda is an XNA program that displays a little greeting in the center of the screen. 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 (which is integrated into Visual 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.

From the File menu of Visual Studio select New and 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.

Visual Studio creates two projects, one for the program and the other for the program's content. XNA programs usually contain lots of content, mostly bitmaps and 3D models, but fonts as well. To add a font to this program, right-click the Content project (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 can edit an extensively 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 Miramonte or Pescadero, you'll get the font actually designed for bold.

The Size tags indicate the point size of the font. In XNA as in Silverlight, you deal almost exclusively with pixel coordinates and dimensions, but the conversion between points and pixels used within XNA is based on 96 DPI displays. The point size of 14 becomes a pixel size of 18-2/3 within your XNA program. This is very close to the 15-point and 20-pixel "normal" *FontSize* in Silverlight for Windows Phone.

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 non-control 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 to load the font. 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 turns out to be irrelevant for Windows Phone 7 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 7!";
        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 on the screen. That position is specified in 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";

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

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. The Windows Phone 7 screens are refreshed at 30 frames per second.

After *Game1* is instantiated, a *Run* method is called 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");
    Vector2 textSize = kootenay14.MeasureString(text);
    Viewport viewport = this.GraphicsDevice.Viewport;

    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.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 *GraphicsDevice* 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 content 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*.

In XNA, sprites (including text strings) are usually displayed by specifying the pixel coordinates relative to the upper-left corner or the sprite relative to the upper-left corner of the display. To calculate these coordinates, it's helpful to know both the screen size and the size of the text when displayed with a particular font.

The *SpriteFont* class has a very handy method named *MeasureString* that returns a *Vector2* object with the size of a particular text string in pixels. (For the 14-point Kootenay font, which has an equivalent height of 18-2/3 pixels, the *MeasureString* call returns a height of 25 pixels.)

An XNA program generally uses the *Viewport* property of the *GraphicsDevice* class to obtain the size of the screen. This is accessible through the *GraphicsDevice* property of *Game* and provides *Width* and *Height* properties.

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.

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 frame-per-second 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 mostly involves fingers touching the screen. All handling of user input should also occur during the *Update* method. You'll see an example in Chapter 3.

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*. Conveniently, XNA provides methods to 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 XNA Game Studio originally created it:

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);
}
```

The color known as cornflower blue has achieved iconic status in the XNA programming community. When you're developing an XNA program, the appearance of the cornflower blue screen is very comforting because it means the program has at least gotten as far as *Draw*. But if you want to conserve power on OLED displays, you want to go with darker backgrounds. In my revised version, I've compromised by setting the background to a darker blue. As in Silverlight, XNA supports the 140 colors that have come to be regarded as standard. The text is colored white:

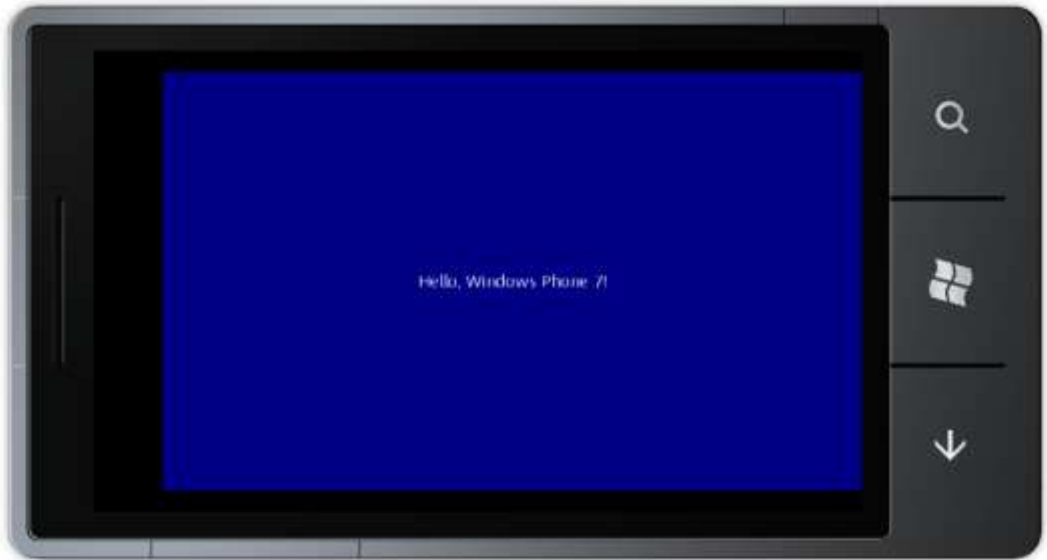
XNA Project: XnaHelloPhone File: Game1.cs (excerpt)

```
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 the color. And here it is:

Oh, that's interesting! By default, Silverlight programs come up in portrait mode, but XNA programs come up in landscape mode. Let's turn the emulator sideways:



Much better!

But this raises a question: Do Silverlight programs always run in portrait mode and XNA programs always run in landscape mode?

Is program biology destiny?

Chapter 2
Getting Oriented

By default, Silverlight programs for Windows Phone 7 run in portrait mode, and XNA programs run in landscape mode. This chapter discusses how to transcend those defaults and explores other issues involving screen sizes, element sizes, and events.

Silverlight and Dynamic Layout

If you run the SilverlightHelloPhone program from the last chapter, and you turn the emulator sideways, you'll discover that the display doesn't change to accommodate the new orientation. That's easy to fix. In the root *PhoneApplicationPage* tag, change the attribute

```
SupportedOrientations="Portrait"
```

to:

```
SupportedOrientations="PortraitOrLandscape"
```

SupportedOrientations is a property of *PhoneApplicationPage*. It's set to a member of the *SupportedPageOrientation* enumeration, either *Portrait*, *Landscape*, or *PortraitOrLandscape*.

Recompile. Now when you turn the emulator sideways, the contents of the page shift around accordingly:

The *SupportedOrientations* property also allows you to restrict your program to Landscape if you need to.

This response to orientation really shows off dynamic layout in Silverlight. Everything has shifted position and some elements have changed size. Silverlight originated in WPF and the desktop, so historically it was designed to react to changes in window sizes and aspect ratios. This facility carries well into the phone.

Two of the most important properties in working with dynamic layout are *HorizontalAlignment* and *VerticalAlignment*. In the last chapter, using these properties to center text in a Silverlight program was certainly easier than performing calculations based on screen size and text size that XNA required.

On the other hand, at this point you would probably find it straightforward to stack a bunch of text strings in XNA, but it's not so obvious how to do the same job in Silverlight.

Rest assured that there are ways to organize elements in Silverlight. A whole category of elements called *panels* exist solely for that purpose. You can even position elements based on pixel coordinates, if that's your preference. But a full coverage of panels won't come until Chapter 10.

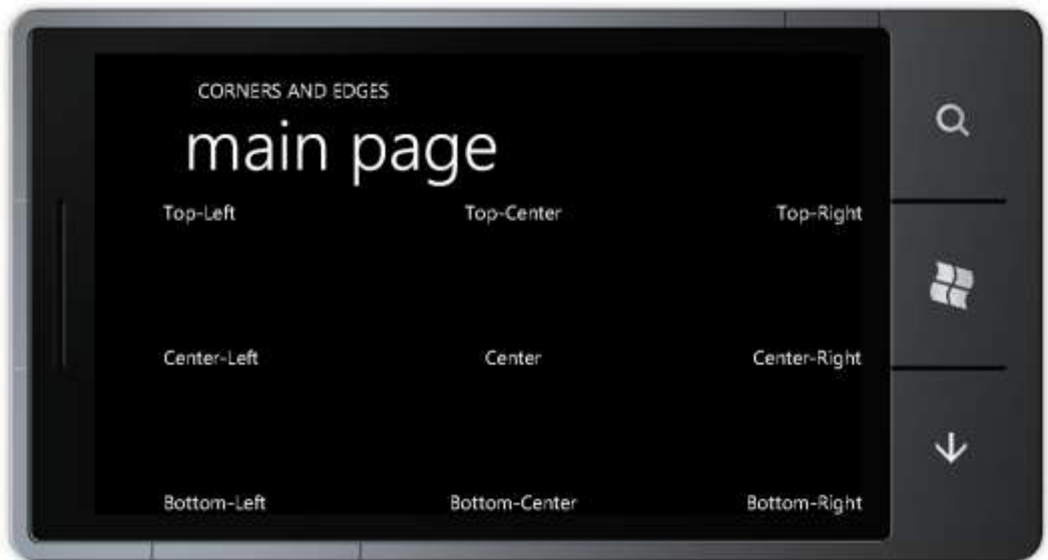
In the meantime, you can try putting multiple elements into the content grid. Normally a *Grid* organizes its content into rows and columns, but this program puts nine *TextBlock* elements in a single-cell *Grid* to demonstrate the use of *HorizontalAlignment* and *VerticalAlignment* in nine different combinations:

Silverlight Project: SilverlightCornersAndEdges   File: MainPage.xaml

```
<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>
```

As with most of the Silverlight programs in the remainder of this book, I've set the *SupportedOrientations* property of *MainPage* to *PortraitOrLandscape*. And here it is turned sideways:



Although this screen appears to show all the combinations, the program does *not* actually show the *default* settings of the *HorizontalAlignment* and *VerticalAlignment* properties. The default settings are enumeration members named *Stretch*. If you try them out, you'll see that the *TextBlock* sits in the upper-left corner, just as with values of *Top* and *Left*. But what won't be so obvious is that the *TextBlock* occupies the entire interior of the *Grid*. The *TextBlock* has a transparent background (and you can't set an alternative) so it's a little difficult to tell the difference. But I'll demonstrate the effect in the next chapter.

Obviously the *HorizontalAlignment* and *VerticalAlignment* properties are very important in the layout system in Silverlight. So is *Margin*. Try adding a *Margin* setting to the first *TextBlock* in this program:

```
<TextBlock Text="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           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 specify only 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 the 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="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           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 (as I'll demonstrate in the next chapter), 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 gets a non-zero *Margin* property to prevent the elements from being jammed up against each other. The *Padding* property is rarer; it's defined only by *TextBlock*, *Border*, and *Control*.

It's possible to use *Margin* to position multiple elements within a single-cell *Grid*. It's not common — and there are better ways to do the job — but it is possible. I'll have an example in Chapter 5.

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="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           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, access the properties named *ActualWidth* and *ActualHeight* instead. (But watch out: These values will have non-zero values only when the element has been rendered on the screen.)

Some useful events are also available for obtaining information involving element sizes. The *Loaded* event is fired when visuals are first arranged 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, such as occurs when orientation changes.

The SilverlightWhatSize project demonstrates the use of the *SizeChanged* method by displaying the sizes of several elements in the standard page. It's not often that you need these precise sizes, but they might be of interest occasionally.

Although you can associate a particular event with an event handler right in XAML, of course the actual event handler must be implemented in code. When you type an event name in XAML (such as *SizeChanged*) Visual Studio will offer to create an event handler for you. That's what I did with the *SizeChanged* event for the content grid:

SilverlightProject: SilverlightWhatSize File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1" SizeChanged="ContentGrid_SizeChanged">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

I also assigned the *TextBlock* property *Name* to "txtblk." The *Name* property plays a very special role in Silverlight. If you compile the program at this point and look inside 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, among them being field named *txtblk* of type *TextBlock*:

```
internal System.Windows.Controls.TextBlock txtblk;
```

You'll also notice that this field is assigned from code in the *InitializeComponent* method:

```
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 *MainPage* class can reference that *TextBlock* element in the XAML file using the *txtblk* variable stored as a field in the class.

You'll notice that several of the elements in the MainPage.xaml file are assigned names with *x:Name* rather than *Name*. As used in XAML, these two attributes are basically equivalent. *Name* only works with elements (that is, instances of classes that derive from *FrameworkElement* because that's where the *Name* property is defined) but *x:Name* works with everything.

Assigning names to elements is one of two primary ways in which code and XAML interact. The second way is for the element defined in XAML to fire an event that is handled in code. Here's the handler for the *SizeChanged* event of the content grid as Visual Studio creates it:

SilverlightProject: SilverlightWhatSize File: MainPage.xaml (excerpt)

```
private void ContentGrid_SizeChanged(object sender, SizeChangedEventArgs e)
{

}
```

I usually don't like the way Visual Studio creates these handlers. Normally I remove the keyword *private*, I rename the event handlers to start them with the word *On,* and I eliminate underscores. This one I'd call *OnContentGridSizeChanged*. I also tend to change the event arguments from *e* to *args*.

But for this program I'll leave it as is. On entry to the method, the *sender* argument is the element that fired the event, in this case the *Grid* named *ContentGrid*. The second argument contains information specific to the event.

I added a body to this method that just sets the *Text* property of *txtblk* to a longish multi-line string:

SilverlightProject: SilverlightWhatSize File: MainPage.xaml (excerpt)

```
private void ContentGrid_SizeChanged(object sender, SizeChangedEventArgs e)
{
    txtblk.Text = String.Format("ContentGrid size: {0}\n" +
                                "TitlePanel size: {1}\n" +
                                "LayoutRoot size: {2}\n" +
```

```
                                "MainPage size: {3}\n" +
                                "Frame size: {4}",
                                e.NewSize,
                                new Size(TitlePanel.ActualWidth,
TitlePanel.ActualHeight),
                                new Size(LayoutRoot.ActualWidth,
LayoutRoot.ActualHeight),
                                new Size(this.ActualWidth, this.ActualHeight),
                                Application.Current.RootVisual.RenderSize);
}
```

The five items are of type *Size*, a structure with *Width* and *Height* properties. The size of the *ContentGrid* itself is available from the *NewSize* property of the event arguments. For the next three, I used the *ActualWidth* and *ActualHeight* properties.

Notice the last item. The static property *Application.Current* returns the *Application* object associated with the current process. This is the *App* object created by the program. It has a property named *RootVisual* that references the frame, but the property is defined to be of type *UIElement*. The *ActualWidth* and *ActualHeight* properties are defined by *FrameworkElement*, the class that derives from *UIElement*. Rather than casting, I chose to use a property of type *Size* that *UIElement* defines.

The first *SizeChanged* event occurs when the page is created and laid out, that is, when the content grid changes size from 0 to a finite value:

The 32-pixel difference between the *MainPage* size and the frame size accommodates the system tray at the top. The topmost *Grid* named *LayoutRoot* is the same size as *MainPage*. The vertical size of the *TitlePanel* (containing the two titles) and the vertical size of *ContentGrid* don't add up to the vertical size of *LayoutRoot* because of the 36-pixel vertical margin (24 pixels on the top and 12 pixels on the bottom) of the *TitlePanel*.

Subsequent *SizeChanged* events occur when something in the visual tree causes a size change, or when the phone changes orientation:

Notice that the frame doesn't change orientation. In the landscape view, the system tray takes away 72 pixels of width from *MainPage*.

Orientation Events

In most of the Silverlight programs in this book, I'll set *SupportedOrientations* to *PortraitOrLandscape*, and try to write orientation-independent applications. For Silverlight programs that get text input, it's crucial for the program to be aligned with the hardware keyboard (if one exists) and the location of that keyboard can't be anticipated.

Obviously there is more to handling orientation changes than just setting the *SupportedOrientations* property! In some cases, you might want to manipulate your layout from code in the page class. If you need to perform any special handling, both *PhoneApplicationFrame* and *PhoneApplicationPage* include *OrientationChanged* events. *PhoneApplicationPage* supplements that event with a convenient and equivalent protected overridable method called *OnOrientationChanged*.

The *MainPage* class in the SilverlightOrientationDisplay project shows how to override *OnOrientationChanged*, but what it does with this information is merely to display the current orientation. The content grid in this project contains a simple *TextBlock*:

SilverlightProject: SilverlightOrientationDisplay   File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Here's the complete code-behind file. The constructor initializes the *TextBlock* text with the current value of the *Orientation* property, which is a member of the *PageOrientation* enumeration:

SilverlightProject: SilverlightOrientationDisplay File: MainPage.xaml.cs

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

namespace SilverlightOrientationDisplay
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
            txtblk.Text = Orientation.ToString();
        }

        protected override void OnOrientationChanged(OrientationChangedEventArgs
args)
        {
            txtblk.Text = args.Orientation.ToString();
            base.OnOrientationChanged(args);
        }
    }
}
```

The *OnOrientationChanged* method obtains the new value from the event arguments.

XNA Orientation

By default, XNA for Windows Phone is set up for a landscape orientation, perhaps to be compatible with other screens on which games are played. If you prefer designing your game for a portrait display, it's easy to do that. In the constructor of the *Game1* class of XnaHelloPhone, try inserting the following statements:

```
graphics.PreferredBackBufferWidth = 240;
graphics.PreferredBackBufferHeight = 320;
```

The *back buffer* is the surface area on which XNA constructs the graphics you display in the *Draw* method. You can control both the size and the aspect ratio of this buffer. Because the buffer width I've specified here is smaller than the buffer height, XNA assumes that I want a portrait display:

Look at that! The back buffer I specified is not the same aspect ratio as the Windows Phone 7 display, so the drawing surface is letter-boxed! The text is larger because it's the same pixel size but now the display resolution has been reduced.

Although you may not be a big fan of the retro graininess of this particular display, you should seriously consider specifying a smaller back buffer if your game doesn't need the high resolution provided by the phone. Performance will improve and battery consumption will decrease. You can set the back buffer to anything from 240 × 240 up to 480 × 800 (for portrait mode) or 800 × 480 (for landscape). XNA uses the aspect ratio to determine whether you want portrait or landscape.

Setting a desired back buffer is also an excellent way to target a specific display dimension in code but allow for devices of other sizes that may come in the future.

By default the back buffer is 800 × 480, but it's actually not displayed at that size. It's scaled down a bit to accommodate the system tray. To get rid of the system tray (and possibly annoy your users who like to always know what time it is) you can set

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

in the *Game1* constructor.

It's also possible to have your XNA games respond to orientation changes, but they'll definitely have to be restructured a bit. The simplest type of restructuring to accommodate orientation changes is demonstrated in the XnaOrientableHelloPhone project. The fields now include a *textSize* variable:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    string text = "Hello, Windows Phone 7!";
    SpriteFont kootenay14;
    Vector2 textSize;
    Vector2 textPosition;
    …
}
```

The *Game1* constructor includes a statement that sets the *SupportedOrientations* property of the *graphics* field:

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

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Allow portrait mode as well
    graphics.SupportedOrientations = DisplayOrientation.Portrait |
                                     DisplayOrientation.LandscapeLeft |
                                     DisplayOrientation.LandscapeRight;

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

The statement looks simple, but there are repercussions. When the orientation changes, the graphics device is effectively reset (which generates some events) and the back buffer dimensions are swapped. You can subscribe to the *OrientationChanged* event of the *GameWindow* class (accessible through the *Window* property) or you can check the *CurrentOrientation* property of the *GameWindow* object.

I chose a little different approach. Here's the new *LoadContent* method, which you'll notice obtains the text size and stores it as a field, but does not get the viewport.

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

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

Instead, the viewport is obtained during the *Update* method because the dimensions of the viewport reflect the orientation of the display.

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

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

    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);
    base.Update(gameTime);
}
```

Whatever the orientation currently is, the Update method calculates a location for the text. The Draw method is the same as several you've seen before.

XNA Project: XnaOrientableHelloPhone   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);
}
```

Now the phone emulator can be turned between portrait and landscape, and the display will switch as well.

If you need to obtain the size of the phone's display independent of any back buffers or orientation (but taking account of the system tray), that's available from the *ClientBounds* property of the *GameWindow* class, which you can access from the *Window* property of the *Game* class:

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

Simple Clocks (*Very* Simple Clocks)

So far in this chapter I've described two Silverlight events — *SizeChanged* and *OrientationChanged* — but used them in different ways. For *SizeChanged*, I associated the event with the event handler in XAML, but for *OrientationChanged*, I overrode the equivalent *OnOrientationChanged* method.

Of course, you can define events entirely in code as well. One handy event for Silverlight programs is the timer, which periodically nudges the program and lets it do some work. A timer is essential for a clock program, for example.

The content grid of the SilverlightSimpleClock project contains just a centered *TextBlock*:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Here's the entire code-behind file. Notice the *using* directive for the *System.Windows.Threading* namespace, which isn't included by default. That's the namespace where *DispatcherTimer* resides:

Silverlight Project: SilverlightSimpleClock   File: MainPage.xaml.cs

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

```
namespace SilverlightSimpleClock
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();

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

        void OnTimerTick(object sender, EventArgs args)
        {
            txtblk.Text = DateTime.Now.ToString();
        }
    }
}
```

The constructor initializes the *DispatcherTimer*, instructing it to call *OnTimerTick* once every second. The event handler simply converts the current time to a string to set it to the *TextBlock*.

Although *DispatcherTimer* is defined in the *System.Windows.Threading* namespace, the *OnTimerTick* method is called in the same thread as the rest of the program. If that was not the case, the program wouldn't be able to access the *TextBlock* directly. (Silverlight elements and related objects are not thread safe.) I'll discuss the procedure for accessing Silverlight elements from secondary threads in Chapter 5.

The clock is yet another Silverlight program in this chapter that changes the *Text* property of a *TextBlock* dynamically during runtime. The new value shows up rather magically without any additional work. This is a very different from older graphical environments like Windows API programming or MFC programming, where a program draws "on demand," that is, when an area of a window becomes invalid and needs to be repainted, or when a program deliberately invalidates an area to force painting.

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 — and re-rendering itself when one of its properties such as *Text* changes — 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 quite well what must also come with great power.

Sometimes an XNA program's display is static; the program might not need to update the display every frame. 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.

An XNA clock program doesn't need a timer because a timer is effectively built into the normal game loop. However, the clock doesn't display milliseconds so the display only needs to be updated every second. For that reason it uses the *SuppressDraw* method to inhibit superfluous *Draw* calls.

Here are the XnaSimpleClock fields:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont kootenay14;
    Viewport viewport;
    Vector2 textPosition;
    StringBuilder text = new StringBuilder();
    DateTime lastDateTime;
    …
}
```

Notice that instead of defining a field of type *string* named *text*, I've defined a *StringBuilder* instead. If you're creating new strings in your *Update* method for display during *Draw* (as this program will do), you should use *StringBuilder* to avoid the heap allocations associated with the normal *string* type. This program will only be creating a new string every second, so I

really didn't need to use *StringBuilder* here, but it doesn't hurt to get accustomed to it. *StringBuilder* requires a *using* directive for the *System.Text* namespace.

Notice also the *lastDateTime* field. This will be used in the *Update* method to determine if the displayed time needs to be updated.

The *LoadContent* method gets the font and the viewport of the display:

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

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    viewport = this.GraphicsDevice.Viewport;
}
```

The logic to compare two *DateTime* values to see if the time has changed is just a little tricky because *DateTime* objects obtained during two consecutive *Update* calls will *always* be different because they have will have different *Millisecond* fields. For this reason, a new *DateTime* is calculated based on the current time obtained from *DateTime.Now*, but subtracting the milliseconds:

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

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

    // Get DateTime with no milliseconds
    DateTime dateTime = DateTime.Now;
    dateTime = dateTime - new TimeSpan(0, 0, 0, 0, dateTime.Millisecond);

    if (dateTime != lastDateTime)
    {
       text.Remove(0, text.Length);
       text.Append(dateTime);
        Vector2 textSize = kootenay14.MeasureString(text);
        textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                                   (viewport.Height - textSize.Y) / 2);
        lastDateTime = dateTime;
    }
    else
    {
        SuppressDraw();
    }
```

```
    base.Update(gameTime);
}
```

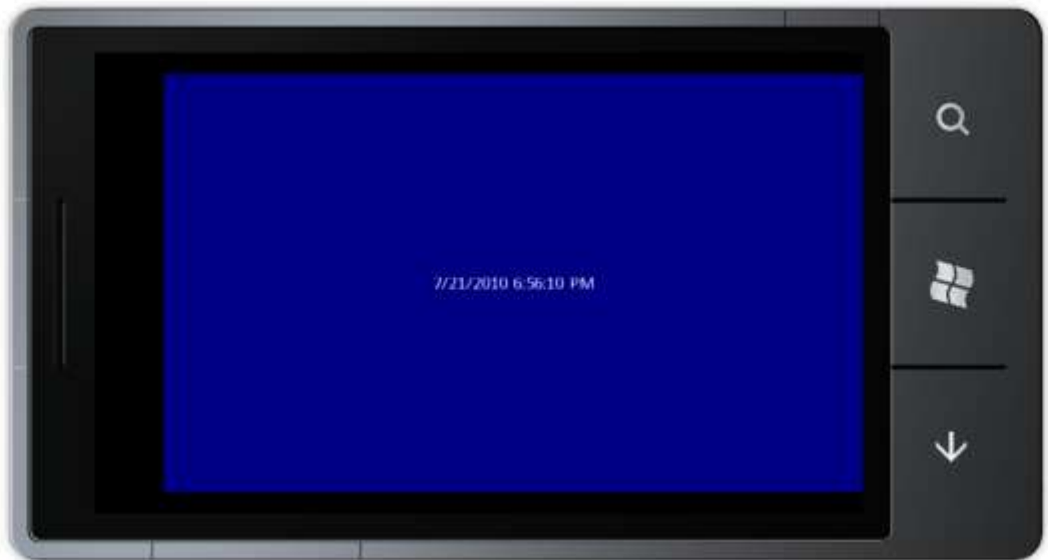
At that point it's easy. If the time has changed, new values of *text*, *textSize*, and *textPosition* are calculated. Because *text* is a *StringBuilder* rather than a *string*, the old contents are removed and the new contents are appended. The *MeasureString* method of *SpriteFont* has an overload for *StringBuilder*, so that call looks exactly the same.

If the time has not changed, *SuppressDraw* is called. The result: *Draw* is called only once per second.

XNA Project: XnaSimpleClock   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);
}
```

DrawString also has an overload for *StringBuilder*. And here's the result:



SuppressDraw can be a little difficult to use — I've found it particularly tricky during the time that the program is first starting up — but it's one of the primary techniques used in XNA to reduce the power requirements of the program.

Chapter 3

An Introduction to Touch

Even for experienced Silverlight and XNA programmers, Windows Phone 7 comes with a feature that is likely to be new and unusual. The screen on the phone is sensitive to touch. And not like old touch screens that basically mimic a mouse, or the tablet screens that recognize handwriting.

The multi-touch screen on a Windows Phone 7 device can detect at least four simultaneous fingers. It is the interaction of these fingers that makes multi-touch so challenging. For this chapter, however, I have much a less ambitious goal. I want only to introduce the touch interfaces in the context of sample programs that respond to simple taps.

The programs in this chapter look much like the "Hello, Windows Phone 7!" programs in the first chapter, except that when you tap the text with your finger, it changes to a random color, and when you tap outside the area of the text, it goes back to white (or whatever color the text was when the program started up).

In a Silverlight program, touch input is obtained through events. In an XNA program, touch input comes through a static class polled during the *Update* method. One of the primary purposes of the XNA *Update* method is to check the state of touch input and make changes that affect what goes out to the screen during the *Draw* method.

Low-Level Touch Handling in XNA

The multi-touch input device is referred to in XNA as a *touch panel*. You use methods in the static *TouchPanel* class to obtain this input. Although you can obtain gestures, let's begin with the lower-level touch information.

It is possible (although not necessary) to obtain information about the multi-touch device itself by calling the static *TouchPanel.GetCapabilities* method. The *TouchPanelCapabilities* object returned from this method has two 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.

For most purposes, you just need to use one of the other two static methods in *TouchPanel*. To obtain low-level touch input, you'll probably 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 three 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.
- *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*.

Very often when dealing with low-level touch input, you'll use a *Dictionary* object with keys based on the *Id* property to retain information for a particular finger.

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. 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* that 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 examine only *TouchLocation* objects with *State* values of *Pressed*.

This project is called XnaTouchHello. Like the other XNA projects you've seen so far, it needs a font, which I've made a little larger so it provides a more substantial touch target. A few more fields are required:

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

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

    Random rand = new Random();
    string text = "Hello, Windows Phone 7!";
    SpriteFont kootenay36;
    Vector2 textSize;
    Vector2 textPosition;
    Color textColor = Color.White;
    …

}
```

The *LoadContent* method is similar to earlier versions except that *textSize* is saved as a field because it needs to be accessed in later calculations:

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

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

    kootenay36 = this.Content.Load<SpriteFont>("Kootenay36");
    textSize = kootenay36.MeasureString(text);
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.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*.

XNA Project: XnaTouchHello   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;
            }
        }
    }

    base.Update(gameTime);
}
```

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*.

The *Draw* method looks very similar to the versions you've seen before, except that the text color is a variable:

XNA Project: XnaTouchHello   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();

    base.Draw(gameTime);
}
```

One problem you might notice is that touch is not quite as deterministic as you might like. Even when you touch the screen with a single finger, the finger might make contact with the

screen in more than one place. In some cases, the same *foreach* loop in *Update* might set *textColor* more than once!

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

The XNA Gesture Interface

The *TouchPanel* class also includes gesture recognition, which is demonstrated by the XnaTapHello project. The fields of this project are the same as those in XnaTouchHello, but the *LoadContent* method is a little different:

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

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

    kootenay36 = this.Content.Load<SpriteFont>("Kootenay36");
    textSize = kootenay36.MeasureString(text);
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);

    TouchPanel.EnabledGestures = GestureType.Tap;
}
```

Notice the final statement. *GestureType* is an enumeration with members *Tap*, *DoubleTap*, *Flick*, *Hold*, *Pinch*, *PinchComplete*, *FreeDrag*, *HorizontalDrag*, *VerticalDrag*, and *DragComplete*, defined as bit flags so you can combine the ones you want with the C# bitwise OR operator.

The *Update* method is very different.

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

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

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gestureSample = TouchPanel.ReadGesture();

        if (gestureSample.GestureType == GestureType.Tap)
        {
```

```
            Vector2 touchPosition = gestureSample.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;
            }
        }
    }

    base.Update(gameTime);
}
```

Although this program is interested in only one type of gesture, the code is rather generalized. If a gesture is available, it is returned from the *TouchPanel.ReadGesture* method as an object of type *GestureSample*. Besides the *GestureType* and *Position* used here, a *Delta* property provides movement information in the form of a *Vector2* object. For some gestures (such as *Pinch*), the *GestureSample* also reports the status of a second touch point with *Position2* and *Delta2* properties.

The *Draw* method is the same as the previous program, but you'll find that the program behaves a little differently from the first one: In the first program, the text changes color when the finger touches the screen; in the second, the color change occurs when the finger lifts from the screen. The gesture recognizer needs to wait until that time to determine what type of gesture it is.

Low-Level Touch Events in Silverlight

Like XNA, Silverlight also supports two different programming interfaces for working with multi-touch, which can be most easily categorized as low-level and high-level. The low-level interface is based around the static *Touch.FrameReported* event, which is very similar to the XNA *TouchPanel* except that it's an event and it doesn't include gestures.

The high-level interface consists of three events defined by the *UIElement* class: *ManipulationStarted*, *ManipulationDelta*, and *ManipulationCompleted*. The *Manipulation* events, as they're collectively called, consolidate the interaction of multiple fingers into movement and scaling factors.

Let me begin with the low-level touch interface in Silverlight by dissecting a class called *TouchPoint*, an instance of which represents a particular finger touching the screen. *TouchPoint* has four get-only properties:

- *Action* of type *TouchAction*, an enumeration with members *Down*, *Move*, and *Up*.
- *Position* of type *Point*, relative to the upper-left corner of a particular element. Let's call this element the *reference* element.
- *Size* of type *Size*. This is supposed to represent the touch area (and, hence, finger pressure, more or less) but the Windows Phone 7 emulator doesn't return useful values.
- *TouchDevice* of type *TouchDevice*.

The *TouchDevice* object has two get-only properties:

- *Id* of type *int*, used to distinguish between fingers. A particular finger is associated with a unique *Id* for all events from *Down* through *Up*.
- *DirectlyOver* of type *UIElement*, the topmost element underneath the finger.

As you can see, the Silverlight *TouchPoint* and *TouchDevice* objects give you mostly the same information as the XNA *TouchLocation* object, but the *DirectlyOver* property of *TouchDevice* is often very useful for determining what element the user is touching.

To use the low-level touch interface, you install a handler for the static *Touch.FrameReported* event:

```
Touch.FrameReported += OnTouchFrameReported;
```

The *OnTouchFrameReported* method looks like this:

```
void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
{
    …
}
```

The event handler gets all touch events throughout your application. The *TouchFrameEventArgs* object has a *TimeStamp* property of type *int*, plus three methods:

- *GetTouchPoints*(*refElement*) returns a *TouchPointCollection*
- *GetPrimaryTouchPoint*(*refElement*) returns one *TouchPoint*
- *SuspendMousePromotionUntilTouchUp*()

In the general case, you call *GetTouchPoints*, passing to it a reference element. The *TouchPoint* objects in the returned collection have *Position* properties relative to that element. You can

pass *null* to *GetTouchPoints* to get *Position* properties relative to the upper-left corner of the application.

The reference element and the *DirectlyOver* element have no relationship to each other. The event always gets all touch activity for the entire program. Calling *GetTouchPoints* or *GetPrimaryTouchPoints* with a particular element does *not* limit the events to only those involving that element. All that it does is cause the *Position* property to be calculated relative to that element. (For that reason, *Position* coordinates can easily be negative if the finger is to the left of or above the reference element.) The *DirectlyOver* element indicates the element under the finger.

A discussion of the second and third methods requires some background: The *Touch.FrameReported* event originated on Silverlight for the desktop, where it is convenient for the mouse logic of existing controls to automatically use touch. For this reason, touch events are "promoted" to mouse events.

But this promotion only involves the "primary" touch point, which is the activity of the first finger that touches the screen when no other fingers are touching the screen. If you don't want the activity of this finger to be promoted to mouse events, the event handler usually begins like this:

```
void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
{
    TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);

    if (primaryTouchPoint != null && primaryTouchPoint.Action == TouchAction.Down)
    {
        args.SuspendMousePromotionUntilTouchUp();
    }
    …
}
```

The *SuspendMousePromotionUntilTouchUp* method can only be called when a finger first touches the screen when no other fingers are touching the screen.

On Windows Phone 7, such logic presents something of a quandary. As written, it basically wipes out all mouse promotion throughout the application. If your phone application incorporates Silverlight controls that were originally written for mouse input but haven't been upgraded to touch, you're basically disabling those controls.

Of course, you can also check the *DirectlyOver* property to suspend mouse promotion selectively. But on the phone, no elements should be processing mouse input except for those controls that don't process touch input! So it might make more sense to *never* suspend mouse promotion.

I'll leave that matter for your consideration and your older mouse-handling controls. Meanwhile, the program I want to write is only interested in the primary touch point when it has a *TouchAction* of *Down*, so I can use that same logic.

The SilverlightTouchHello project has a *TextBlock* in the XAML file:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               Padding="0 22"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Notice the *Padding* value. I know that the font displayed here has a *FontSize* property of 20 pixels, which actually translates into a *TextBlock* that is about 27 pixels tall. I also know that it's recommended that touch targets not be smaller than 9 millimeters. If the resolution of the phone display is 200 DPI, then 9 millimeters is 71 pixels. (The calculation is 9 millimeters divided by 25.4 millimeters to the inch, times 200 pixels per inch.) The *TextBlock* is short by 44 pixels. So I set a *Padding* value that puts 22 more pixels on both the top and bottom (but not the sides).

I used *Padding* rather than *Margin* because *Padding* is space *inside* the *TextBlock*. The *TextBlock* actually becomes larger than the text size would imply. *Margin* is space *outside* the *TextBlock*. It's not part of the *TextBlock* itself and is excluded for purposes of hit-testing.

Here's the complete code-behind file. The constructor of *MainPage* installs the *Touch.FrameReported* event handler.

Silverlight Project: SilverlightTouchHello File: MainPage.xaml.cs

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

namespace SilverlightTouchHello
{
    public partial class MainPage : PhoneApplicationPage
    {
        Random rand = new Random();
        Brush originalBrush;

        public MainPage()
        {
```

```
            InitializeComponent();
            originalBrush = txtblk.Foreground;
            Touch.FrameReported += OnTouchFrameReported;
        }

        void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
        {
            TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);

            if (primaryTouchPoint != null && primaryTouchPoint.Action ==
TouchAction.Down)
            {
                if (primaryTouchPoint.TouchDevice.DirectlyOver == txtblk)
                {
                    txtblk.Foreground = new SolidColorBrush(
                                Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
                }
                else
                {
                    txtblk.Foreground = originalBrush;
                }
            }
        }
    }
}
```

The event handler is only interested in primary touch points with an *Action* of *Down*. If the *DirectlyOver* property is the element named *txtblk*, a random color is created. 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 *DirectlyOver* is not *txtblk*, then the program doesn't change the color to white, because that wouldn't work if the user chose a color theme of black text on a white background. Instead, it sets the *Foreground* property to the brush originally set on the *TextBlock*. This is obtained in the constructor.

The Manipulation Events

The high-level touch interface involves three events: *ManipulationStarted*, , *ManipulationDelta*, and *ManipulationCompleted*. These events don't bother with reporting the activity of individual fingers. Instead, they consolidate the activity of multiple fingers into translation and

scaling operations. The events also accumulate velocity information, so while they don't support inertia directly, they can be used to implement inertia.

The *Manipulation* events will receive more coverage in the chapters ahead. 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.

While *Touch.FrameReported* delivered touch information for the entire application, the *Manipulation* events are based on individual elements, so in SilverlightTapHello1, a *ManipulationStarted* event handler is set on the *TextBlock*:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone 7!"
               Padding="0 22"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

The MainPage.xaml.cs contains this event handler:

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

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

    public MainPage()
    {
        InitializeComponent();
    }
    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);

       args.Complete();
    }
}
```

The event handler is able to get the element generating the message from the *sender* argument. The TextBlock is also available from the *args.OriginalSource* property.

Notice the call to the *Complete* method of the event arguments at the end. This is not required by effectively tells the system that further *Manipulation* events involving this finger won't be necessary.

This program is flawed: If you try it out, you'll see that it works only partially. Touching the *TextBlock* changes the text to a random color. But if you touch outside the *TextBlock*, the text does *not* go back to white. Because this event was set on the *TextBlock*, the event handler is called only when the user touches the *TextBlock*. No other *Manipulation* events are processed by the program.

A program that functions correctly according to my original specification needs to get touch events 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 defines all the *Manipulation* events. But the *Control* class (from which MainPage derives) supplements those events with protected virtual methods. 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 7!"
              Padding="0 22"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

The *MainPage* class overrides the *OnManipulationStarted* method:

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

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    Brush originalBrush;

    public MainPage()
```

```
    {
        InitializeComponent();
        originalBrush = txtblk.Foreground;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        if (args.OriginalSource == txtblk)
        {
            txtblk.Foreground = new SolidColorBrush(
                        Color.FromArgb(255, (byte)rand.Next(256),
                                            (byte)rand.Next(256),
                                            (byte)rand.Next(256)));
        }
        else
        {
            txtblk.Foreground = originalBrush;
        }

       args.Complete();
        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 for the *Foreground* property. If not, then the *Foreground* property is set to the original brush.

In this *OnManiulationStarted* method we're handling events for *MainPage*, but that *OriginalSource* property tells us the event actually originated lower in the visual tree. This is part of the benefit of routed event handling.

Routed Events

In Microsoft Windows programming, keyboard and mouse input always go to particular controls. Keyboard input always goes to the control with the input focus. Mouse input always goes to the topmost enabled control under the mouse pointer, and stylus and touch input is similar. But sometimes this is inconvenient. Sometimes the control underneath needs the user-input more than the control on top.

To be a bit more flexible, Silverlight implements a system called *routed event handling*. Most user input events—including the three *Manipulation* events—do indeed originate using the same paradigm as Windows. The Manipulation events originate at the topmost enabled element touched by the user. However, if that element is not interested in the event, the event then goes to that element's parent, and so forth up the visual tree ending at the *PhoneApplicationFrame* element. Any element along the way can grab the input and do something with it, and also inhibit further progress of the event up the tree.

This is why you can override the *OnManipulationStarted* method in *MainPage* and also get manipulation events for the *TextBlock*. By default the *TextBlock* isn't interested in those events.

The event argument for the *ManipulationStarted* event is *ManipulationStartedEventArgs*, which derives from *RoutedEventArgs*. It is *RoutedEventArgs* that defines the *OriginalSource* property that indicates the element on which the event began.

But this suggests another approach that combines the two techniques shown in SilverlightTapHello1 and SilverlightTapHello2. Here's the XAML file of SilverlightTapHello3:

Silverlight Project: SilverlightTapHello3 File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               Padding="0 22"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

The *TextBlock* has a *Name* as in the first program. A handler for the *ManipulationStarted* event is set on the *TextBlock* as in the first program. Both the event handler and an override of *OnManipulationStarted* appear in the code-behind file:

Silverlight Project: SilverlightTapHello3 File: MainPage.xaml.cs (excerpt)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    Brush originalBrush;

    public MainPage()
    {
        InitializeComponent();
        originalBrush = txtblk.Foreground;
    }

    void OnTextBlockManipulationStarted(object sender,
                                        ManipulationStartedEventArgs args)
    {
        txtblk.Foreground = new SolidColorBrush(
                    Color.FromArgb(255, (byte)rand.Next(256),
                                        (byte)rand.Next(256),
                                        (byte)rand.Next(256)));
       args.Complete();
        args.Handled = true;
    }
```

```
    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        txtblk.Foreground = originalBrush;

       args.Complete();
        base.OnManipulationStarted(args);
    }
}
```

The logic has been split between the two methods, making the whole thing rather more elegant, I think. The *OnTextBlockManipulationStarted* method only gets events when the *TextBlock* is touched. The *OnManipulationStarted* event gets all events for *MainPage*.

At first there might seem to be a bug here. After *OnTextBlockManipulationStarted* is called, the event continues to travel up the visual tree and *OnManipulationStarted* sets the color back to white. But that's not what happens: The crucial statement that makes this work right is this one at the end of the *OnTextBlockManipulationStarted* handler for the TextBlock:

```
args.Handled = true;
```

That statement says that the event has now been handled and it should *not* travel further up the visual tree. Remove that statement and the *TextBlock* never changes from its initial color — at least not long enough to see.

Some Odd Behavior?

Now try this. In many of the Silverlight programs I've shown so far, I've centered the *TextBlock* within the content grid by setting the following two attributes:

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

Delete them from SilverlightTapHello3, and recompile and run the program. The text appears at the upper-left corner of the *Grid*. But now 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.

By default the *HorizontalAlignment* and *VerticalAlignment* properties are set to enumeration values called *Stretch*. The *TextBlock* is actually filling the *Grid*. You can't see it, of course, but the fingers don't lie.

With other elements, as you'll see, this stretching effect is much less subtle.

Chapter 4

Bitmaps, Also Known as Textures

Aside from text, one of the most common objects to appear in both Silverlight and XNA applications is the *bitmap*, formally defined as a two-dimensional array of bits corresponding to the pixels of a graphics display device.

In Silverlight, a bitmap is sometimes referred to as an *image*, but that's mostly a remnant of the Windows Presentation Foundation, where the word *image* refers to both bitmaps and vector-based drawings. In WPF and Silverlight, the *Image* element displays bitmaps but is not the bitmap itself.

In XNA, a bitmap has a data type of *Texture2D* and hence is often referred to as a *texture*, but that term is mostly related to 3D programming where bitmaps are used to cover surfaces of 3D solids. In XNA 2D programming, bitmaps are often used as sprites.

Bitmaps are also used to symbolize the program on the phone. A new XNA project in Visual Studio results in the creation of two bitmaps, and Silverlight adds a third.

The native Windows bitmap format has an extension of BMP but it's become less popular in recent years as compressed formats have become widespread. At this time, the three most popular bitmap formats are probably:

- JPEG (Joint Photography Experts Group)
- PNG (Portable Network Graphics)
- GIF (Graphics Interchange File)

XNA supports all three (and more). Silverlight supports only JPEG and PNG. (And if you're like most Silverlight programmers, you'll not always remember this simple fact and someday wonder why your Silverlight program simply refuses to display a GIF or a BMP.)

The compression algorithms implemented by PNG and GIF do not result in the loss of any data. The original bitmap can be reconstituted exactly. For that reason, these are often referred to as "lossless" compression algorithms.

JPEG implements a "lossy" algorithm by discarding visual information that is less perceptible by human observers. This type of compression works well for real-world images such as photographs, but less suitable for bitmaps that derive from text or vector-based images, such as architectural drawings or cartoons.

Both Silverlight and XNA allow manipulating bitmaps at the pixel level for generating bitmaps — or altering existing bitmaps — interactively or algorithmically. That topic is relegated to

future chapters. This chapter will focus more on the techniques of obtaining bitmaps from various sources, including the program itself, the Web, and the phone's built-in camera.

XNA Texture Drawing

Because XNA 2D programming is almost entirely a process of moving sprites around the screen, you might expect that loading and drawing bitmaps in an XNA program is fairly easy, and you would be correct.

The first project is called XnaLocalBitmap, so named because this bitmap will be stored as part of the program's content. To add a new bitmap to the program's content project, right-click the XnaLocalBitmapContent project name, select Add and then New Item, and then Bitmap File. You can create the bitmap right in Visual Studio.

Or, you can create the bitmap in an external program, as I did. Windows Paint is often convenient, so for this exercise I created the following bitmap with a dimension of 320 pixels wide and 160 pixels high:

This was saved under the name Hello.png.

To add this file as part of the program's content, right-click the XnaLocalBitmapContent project in Visual Studio, select Add and Existing Item, and then navigate to the file. Once the file shows up, you can right-click it to display Properties, and you'll see that it has an Asset Name of "Hello."

The goal is to display this bitmap centered on the screen. Define a field in the Game1.cs file to store the *Texture2D* and another field for the position:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
```

```
    SpriteBatch spriteBatch;
    Texture2D helloTexture;
    Vector2 position;
    …
}
```

Both fields are set during the *LoadContent* method. Use the same generic method to load the *Texture2D* as you use to load a *SpriteFont*. The *Texture2D* class has properties named *Width* and *Height* that provide the dimensions of the bitmap in pixels. As with the programs that centered text in the Chapter 1, the *position* field indicates the pixel location on the display that corresponds to the upper-left corner of the bitmap:

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

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    helloTexture = this.Content.Load<Texture2D>("Hello");
    Viewport viewport = this.GraphicsDevice.Viewport;
    position = new Vector2((viewport.Width - helloTexture.Width) / 2,
                           (viewport.Height - helloTexture.Height) / 2);
}
```

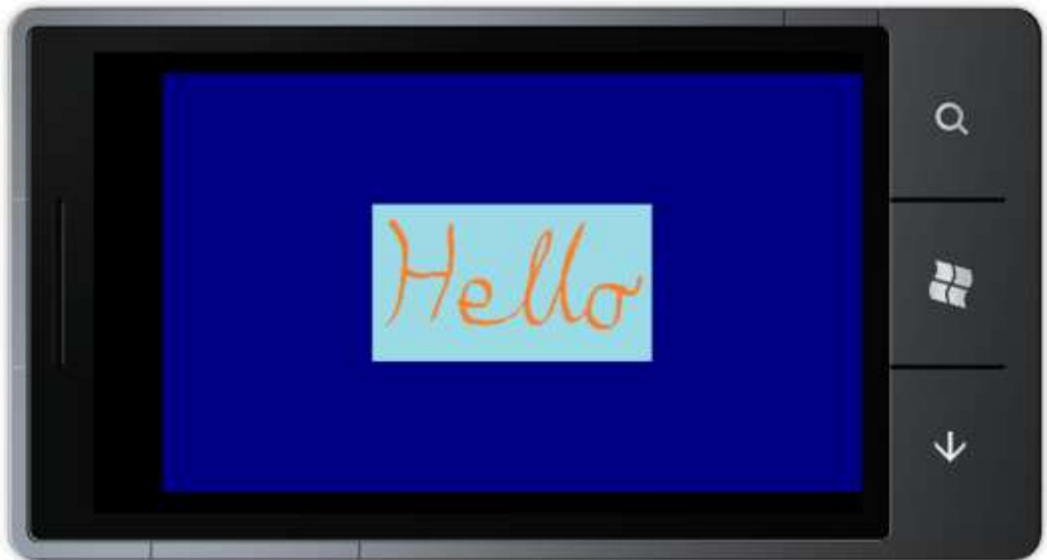
The *SpriteBatch* class has seven *Draw* methods to render bitmaps. This one is certainly the simplest:

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

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

    spriteBatch.Begin();
    spriteBatch.Draw(helloTexture, position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

The final argument to Draw is a color that can be used to attenuate the existing colors in the bitmap. Use *Color.White* if you want the bitmap's colors to display without any alteration.

And here it is:



The Silverlight *Image* Element

The equivalent program in Silverlight is even simpler. Let's create a project named SilverlightLocalBitmap. First create a directory in the project to store the bitmap. This isn't strictly required but it makes for a tidier project. Programmers usually name this directory Images or Media or Assets depending on the types of files that might be stored there. Right-click the project name and choose Add and then New Folder. Let's name it Images. Then right-click the folder name and choose Add and Existing Item. Navigate to the Hello.png file. (If you've created a different bitmap on your own, keep in mind that Silverlight supports only JPEG and PNG files.)

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 bitmap filename and display Properties. Make sure the Build Action is Resource.

In Silverlight, you use the *Image* element to display bitmaps just as you use the *TextBlock* element to display text. Set the *Source* property to the folder and filename of the bitmap within the project:

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

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

The display looks a little different than the XNA program, and it's not just the titles. By default, the *Image* element expands or contracts the bitmap as much as possible to fill its container (the content grid) while retaining the correct aspect ratio. This is most noticeable if you set the *SupportedOrientations* attribute of the *PhoneApplicationPage* start tag to *PortraitOrLandscape* and turn the phone sideways:

If you want to display the bitmap in its native pixel size, you can set the *Stretch* property of Image to *None*:

```
<Image Source="Images/Hello.png"
       Stretch="None" />
```

I'll discuss more options in Chapter 7.

Images Via the Web

One feature that's really nice about the *Image* element is that you can set the *Source* property to a URL, such as in this Silverlight project:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Image Source="http://www.charlespetzold.com/Media/HelloWP7.jpg" />
</Grid>
```

Here it is:

This is certainly easy enough, and pulling images off the Web rather than binding them into the application certainly keeps the size of the executable down. But an application running on Windows Phone 7 is not guaranteed to have an Internet connection, aside from the other problems associated with downloading. The *Image* element has two events named

ImageOpened and *ImageFailed* that you can use to determine if the download was successful or not.

For Windows Phone 7 programs that display a lot of bitmaps, you need to do some hard thinking. You can embed the bitmaps into the executable and have their access guaranteed, or you can save space and download them when necessary.

In XNA, downloading a bitmap from the Web is not quite as easy, but a .NET class named *WebClient* makes the job relatively painless. It's somewhat easier to use than the common alternative (*HttpWebRequest* and *HttpWebResponse*) and is often the preferred choice for downloading individual items.

You can use *WebClient* to download either strings (commonly XML files) or binary objects. The actual transfer occurs asynchronously and then *WebClient* calls a method in your program to indicate completion or failure. This method call is in your program's thread, so you get the benefit of an asynchronous data transfer without explicitly dealing with secondary threads.

To use *WebClient* in an XNA program, you'll need to add a reference to the System.Net library: In the Solution Explorer, under the project name, right click References and select Add Reference. In the .NET table, select System.Net. (Silverlight programs get a reference to System.Net automatically.)

The Game1.cs file of the XnaWebBitmap project also requires a *using* directive for the *System.Net* namespace. The program defines the same fields as the earlier program:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D helloTexture;
    Vector2 position;
    …
}
```

The *LoadContent* method creates an instance of *WebClient*, sets the callback method, and then initiates the transfer:

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

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

    WebClient webClient = new WebClient();
```

```
    webClient.OpenReadCompleted += OnWebClientOpenReadCompleted;
    webClient.OpenReadAsync(new
Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg"));
}
```

The *OnWebClientOpenReadCompleted* method is called when the entire file has been downloaded. You'll want to check if the download hasn't been cancelled and that no error has been reported. If everything is OK, the *Result* property of the event arguments is of type *Stream*. You can use that *Stream* with the static *Texture2D.FromStream* method to create a *Texture2D* object:

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

```
void OnWebClientOpenReadCompleted(object sender, OpenReadCompletedEventArgs args)
{
    if (!args.Cancelled && args.Error == null)
    {
        helloTexture = Texture2D.FromStream(this.GraphicsDevice, args.Result);
        Viewport viewport = this.GraphicsDevice.Viewport;
        position = new Vector2((viewport.Width - helloTexture.Width) / 2,
                               (viewport.Height - helloTexture.Height) / 2);
    }
}
```

The *Texture2D.FromStream* method supports JPEG, PNG, and GIF.

By default, the *AllowReadStreamBuffering* property of *WebClient* is *true*, which means that the entire file will have been downloaded when the *OpenReadCompleted* event is raised. The *Stream* object available in the *Result* property is actually a memory stream, except that it's an instance of a class internal to the .NET libraries rather than *MemoryStream* itself.

If you set *AllowReadStreamBuffering* to *false*, then the *Result* property will be a network stream. The *Texture2D* class will not allow you to read from that stream on the main program thread.

Normally the *LoadContent* method is called before the first call to the *Update* or *Draw* method, but it is essential to remember that a gap of time will separate *LoadContent* from the *OnWebClientOpenReadCompleted* method. During that time an asynchronous read is occurring, but the *Game1* class is proceeding as normal with calls to *Update* and *Draw*. For that reason, you should only attempt to access the *Texture2D* object when you know that it's valid:

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

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

    if (helloTexture != null)
    {
        spriteBatch.Begin();
        spriteBatch.Draw(helloTexture, position, Color.White);
        spriteBatch.End();
    }

    base.Draw(gameTime);
}
```

In a real program, you'd also want to provide some kind of notification to the user if the bitmap could not be downloaded.

Image and *ImageSource*

Although you can certainly use *WebClient* in a *Silverlight* application, it's not generally necessary with bitmaps because the bitmap-related classes already implement asynchronous downloading.

However, once you begin investigating the *Image* element, it may seem a little confusing. The *Image* element is not the bitmap; the *Image* element merely displays the bitmap. In the uses you've seen so far, the *Source* property of *Image* has been set to a file path or a URL:

```
<Image Source="Images/Hello.png" />
<Image Source="http://www.charlespetzold.com/Media/HelloWP7.jpg" />
```

You might have assumed that this *Source* property was of type *string*. Sorry, not even close! You're actually seeing XAML syntax that hides some extensive activity behind the scenes. The *Source* property is really of type *ImageSource*, an abstract class from which derives *BitmapSource*, another abstract class but one that defines a method named *SetSource* that allows loading the bitmap from a *Stream* object.

From *BitmapSource* derives *BitmapImage*, which supports a constructor that accepts a *Uri* object and also includes a *UriSource* property of type *Uri*. The SilverlightTapToDownload1 project mimics a program that needs to download a bitmap whose URL is known only at runtime. The XAML contains an *Image* element with no bitmap to display:

Silverlight Project: SilverlightTapToDownload1 File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Image Name="img" />
</Grid>
```

BitmapImage requires a *using* directive for the *System.Windows.Media.Imaging* namespace. When *MainPage* gets a tap, it creates a *BitmapImage* from the *Uri* object and sets that to the *Source* property of the *Image*:

Silverlight Project: SilverlightTapToDownload1   File: MainPage.xaml.cs (excerpt)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    Uri uri = new Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg");
    BitmapImage bmp = new BitmapImage(uri);
    img.Source = bmp;

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Remember to tap the screen to initiate the download!

The *BitmapImage* class defines *ImageOpened* and *ImageFailed* events (which the *Image* element also duplicates) and also includes a *DownloadProgess* event.

If you want to explicitly use *WebClient* in a Silverlight program, you can do that as well, as the next project demonstrates. The SilverlightTapToDownload2.xaml file is the same as SilverlightTapToDownload1.xaml. The code-behind file uses *WebClient* much like the earlier XNA program:

Silverlight Project: SilverlightTapToDownload2   File: MainPage.xaml.cs (excerpt)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    WebClient webClient = new WebClient();
    webClient.OpenReadCompleted += OnWebClientOpenReadCompleted;
    webClient.OpenReadAsync(new
Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg"));

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}

void OnWebClientOpenReadCompleted(object sender, OpenReadCompletedEventArgs args)
{
    if (!args.Cancelled && args.Error == null)
    {
        BitmapImage bmp = new BitmapImage();
        bmp.SetSource(args.Result);
        img.Source = bmp;
```

```
    }
}
```

Notice the use of *SetSource* to create the bitmap from the *Stream* object.

Loading Local Bitmaps from Code

In a Silverlight program, you've seen that a bitmap added to the project as a resource is bound into the executable. It's so customary to reference that local bitmap directly from XAML that very few experienced Silverlight programmers could tell you offhand how to do it in code. The SilverlightTapToLoad project shows you how.

Like the other Silverlight programs in this chapter, the SilverlightTapToLoad project contains an *Image* element in its content grid. The Hello.png bitmap is stored in the Images directory and has a Build Action of Resource.

The MainPage.xaml.cs file requires a *using* directive for the *System.Windows.Media.Imaging* namespace for the *BitmapImage* class. Another *using* directive for *System.Windows.Resources* is required for the *StreamResourceInfo* class.

When the screen is tapped, the event handler accesses the resource using a static method defined by the *Application* class:

Silverlight Project: SilverlightTapToLoad   File: MainPage.xaml.cs

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    Uri uri = new Uri("/SilverlightTapToLoad;component/Images/Hello.png",
UriKind.Relative);
    StreamResourceInfo resourceInfo = Application.GetResourceStream(uri);
    BitmapImage bmp = new BitmapImage();
    bmp.SetSource(resourceInfo.Stream);
    img.Source = bmp;

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Notice how complicated that URL is! It begins with the name of the program followed by a semicolon, followed by the word "component" and then the folder and filename of the file. If you change the Build Action of the Hello.png file to Content rather than Resource, you can simplify the syntax considerably:

```
Uri uri = new Uri("Images/Hello.png", UriKind.Relative);
```

What’s the difference?

If you navigate to the *Bin/Debug* subdirectory of the Visual Studio project, you’ll see a SilverlightTapToLoad.xap. That’s the executable. The filename extension XAP is pronounced “zap” and the file is actually in ZIP format. (Yes, the XAP is a ZIP.) If you rename it the file to a ZIP extension you can look inside. The bulk of the file will be SilverlightTapToLoad.dll, the compiled program.

In both cases, the bitmap is obviously stored somewhere within the XAP file. The difference is this:

- With a Build Action of Resource for the bitmap, it is stored inside the SilverlightTapToLoad.dll file with the compiled program
- With a Build Action of Content, the bitmap is stored external to the SilverlightTapToLoad.dll file but within the XAP file, and you when you rename the XAP file to a ZIP file, you can see the *Images* directory and the file.

If you have a number of images in your program, and you don’t want to include them all in the XAP file, but you’re nervous about downloading the images, why not do a little of both? Include low resolution (or highly compressed) images in the XAP file, but download better versions asynchronously while the application is running.

Capturing from the Camera

Besides embedding bitmaps in your application or accessing them from the web, Windows Phone 7 also allows you to acquire images from the built-in camera.

Your application has no control over the camera itself. For reasons of security, you cannot arbitrarily snap a picture, or “see” what’s coming through the camera lens. Your application basically invokes a standard camera utility, the user points and shoots, and the picture is returned back to your program.

The classes you use for this job are in the *Microsoft.Phone.Tasks* namespace, which contains a number of classes referred to as *choosers* and *launchers*. Conceptually, these are rather similar, except that choosers return data to your program but launchers do not.

The *CameraCaptureTask* is derived from the generic *ChooserBase* class which defines a *Completed* event and a *Show* method. Your program attaches a handler for the *Completed* event and calls *Show*. When the *Completed* event handler is called, the *PhotoResult* event argument contains a *Stream* object to the photo. From there, you already know what to do. (In much the same way, the *PhotoChooserTask* class allows the user to access the photo library and return a selected photo to your program.)

Like the earlier programs in this chapter, the SilverlightTapToShoot program contains an *Image* element in the content grid of its MainPage.xaml file. Here's the entire code-behind file:

Silverlight Project: SilverlightTapToShoot   File: MainPage.xaml.cs

```
using System.Windows.Input;
using System.Windows.Media.Imaging;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Tasks;

namespace SilverlightTapToShoot
{
    public partial class MainPage : PhoneApplicationPage
    {
        CameraCaptureTask camera;

        public MainPage()
        {
            InitializeComponent();

            camera = new CameraCaptureTask();
            camera.Completed += OnCameraCaptureTaskCompleted;
        }

        protected override void OnManipulationStarted(ManipulationStartedEventArgs
args)
        {
            camera.Show();

            args.Complete();
            args.Handled = true;
            base.OnManipulationStarted(args);
        }

        void OnCameraCaptureTaskCompleted(object sender, PhotoResult args)
        {
            if (args.TaskResult == TaskResult.OK)
            {
                BitmapImage bmp = new BitmapImage();
                bmp.SetSource(args.ChosenPhoto);
                img.Source = bmp;
            }
        }
    }
}
```

Now listen up. This is important: When you run this program on the phone emulator from Visual Studio, I want you to choose Start Without Debugging from the Debug menu. *Do not run this program under the debugger.* (Don't worry — I'll tell you how to run it under the debugger in a moment.)

The program will come up on the phone emulator like normal, but you'll see that Visual Studio has let go of it. The program is running on its own while Visual Studio has returned to edit mode.

When you tap the emulator screen, the call to *Show* causes the camera task to start up. You can "shoot" a photo by tapping a sporadically appearing icon in the upper-right corner of the screen. The simulated "photo" just looks like a large white square with a small black square inside one of the edges. Then you need to click the Accept button. The camera goes away.

The *OnCameraCaptureTaskCompleted* method then takes over. It creates a *BitmapImage* object, sets the input stream from *args.ChoosenPhoto*, and then sets the *BitmapImage* object to the *Image* element, displaying the photo on the screen.

The whole process seems fairly straightforward. Conceptually it seems as if the program is spawning the camera process, and then resuming control when that camera process terminates.

But that's not it. There's something else going on that is not so evident at first and which you will probably find somewhat unnerving.

When the SilverlightTapToShoot program calls the *Show* method on the *CameraCaptureTask* object, the SilverlightTapToShoot program is terminated. (Not immediately, though. The *OnManipulationStarted* method is allowed to return back to the program, and a couple other events are fired, but then the program is definitely terminated.)

The camera utility then runs. When the camera utility has done its job, the SilverlightTapToShoot program is re-executed. It's a new instance of the program. The program starts up from the beginning, and then the *OnCameraCaptureTaskCompleted* method is called.

Want to be convinced? Try inserting the statement

```
PageTitle.Text = "tapped!";
```

at the beginning of the *OnManipulationStarted* handler. When you tap the screen, the title reading "main page" becomes "tapped!" right before the program gives way to the camera. When the program returns, the title is back to "main page." That only makes sense if the SilverlightTapToShoot program that the camera returns to is a new instance.

Or try this: Here's the statement that attaches the handler for the *Completed* event:

```
camera.Completed += OnCameraCaptureTaskCompleted;
```

Move that statement from the constructor of the program to the *OnManipulationStarted* override (before the call to the *Show* method, of course). Normally that wouldn't be a problem. But because there's a new instance of SilverlightTapToShoot starting up after the

camera utility finishes, the code to install the event handler doesn't get executed, and there's no place to return the camera result.

One important lesson: When you make use of a chooser or launcher such as *CameraCaptureTask*, you must create the chooser object and attach the *Completed* handler in your page's constructor.

Now let's try running SilverlightTapToShoot from Visual Studio again but using the debugger. From the Debug menu choose Start Debugging or press F5. The SilverlightTapToShoot program comes up on the emulator.

Tap the screen to invoke the camera. You'll notice that Visual Studio indicates that the program has terminated and retreats back into edit mode.

Take a photo, and tap the Accept button. Here's where something different happens. As a new instance of SilverlightTapToShoot starts up, you'll see a blank screen on the emulator. Now, without wasting a lot of time—you have 10 seconds for this little maneuver—press F5 in Visual Studio again. (Or, choose Start Debugging from the Debug menu.) Visual Studio magically attaches its debugger to this new instance. SilverlightTapToShoot comes up on the emulator displaying the beautiful photo you just took, and if necessary you can use Visual Studio to debug any subsequent code.

This termination and re-execution of your program is a characteristic of Windows Phone 7 programming call *tombstoning*. When the program is terminated as the camera task begins, sufficient information is retained by the phone operating system to start the program up again when the camera finishes. However, not enough information is retained to restore the program entirely to its pre-tombstone state (as you saw with setting the *PageTitle* text). That's your responsibility.

Running a launcher or chooser is one way tombstoning can occur. But it also occurs when the user leaves your program by pressing the Start button on the phone. Eventually the user could return to your program by pressing the Back button, and your program needs to be re-executed from its tombstoned state. Tombstoning also takes place when a lack of activity on the phone causes it to go into a lock state.

Tombstoning does *not* occur when your program is running and the user presses the Back button. The Back button simply terminates the program normally.

When tombstoning occurs, obviously you'll want to save some of the state of your program so you can restore that state when the program starts up again, and obviously Windows Phone 7 has facilities to help you out. That's in Chapter 6.

Chapter 5
Sensors and Services

This chapter covers two of the facilities in Windows Phone 7 that provide information about the outside world. With the user's permission, the location service lets your application obtain the phone's location on the earth in the traditional geographic coordinates of longitude and latitude, whereas the accelerometer tells your program which way is down.

The accelerometer and location service are related in that neither of them will work very well in outer space.

Although the accelerometer and the location service are ostensibly rather easy, this chapter also explores issues involved with working with secondary threads of execution, handling asynchronous operations, and accessing web services.

Accelerometer

Windows Phones contain an accelerometer — a small hardware device that essentially measures force, which elementary physics tells us is proportional to acceleration. When the phone is held still, the accelerometer responds to the force of gravity, so the accelerometer can tell your application the direction of the Earth relative to the phone.

A simulation of a bubble level is an archetypal application that makes use of an accelerometer, but the accelerometer can also provide a basis for interactive animations. For example, you might pilot a messenger bike through the streets of Manhattan by tilting the phone left or right to indicate steering.

The accelerometer also responds to sudden movements such as shakes or jerks, useful for simulations of dice or some other type of randomizing activity. Coming up with creative uses of the accelerometer is one of the many challenges of phone development.

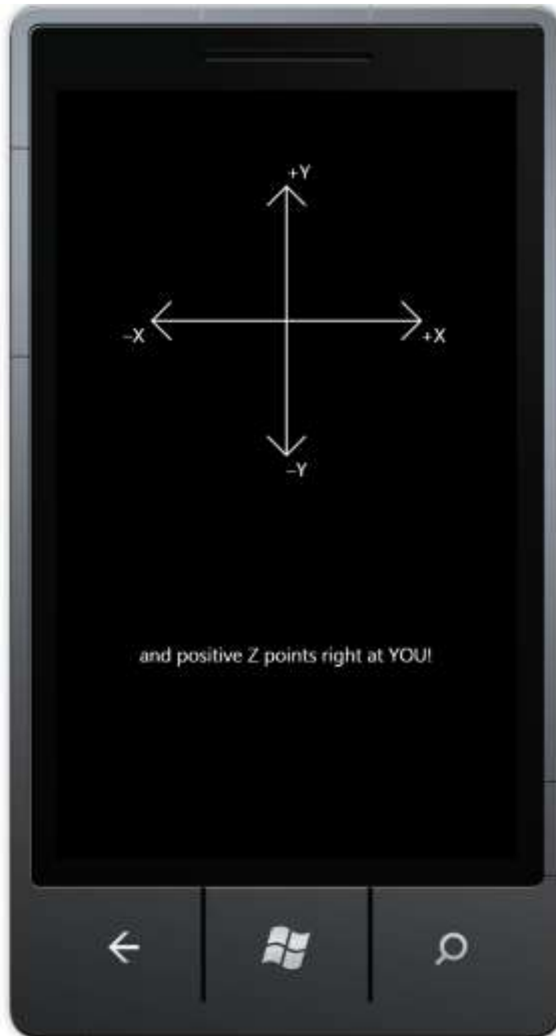
It is convenient to represent the accelerometer output as a vector in three-dimensional space. Vectors are commonly written in boldface, so the acceleration vector can be symbolized as ***(x, y, z)***. XNA defines a three-dimensional vector type; Silverlight does not.

While a three-dimensional point (*x*, *y*, *z*) indicates a particular location in space, the vector ***(x, y, z)*** encapsulates instead a direction and a magnitude. Obviously the point and the vector are related: The direction of the vector ***(x, y, z)*** is the direction from the point (0, 0, 0) to the point(*x*, *y*, *z*). But the vector ***(x, y, z)*** is definitely not the line from (0, 0, 0) to (*x*, *y*, *z*). It's only the direction of that line.

The magnitude of the vector ***(x, y, z)*** is calculable from the three-dimensional form of the Pythagorean Theorem:

$$Magnitude = \sqrt{x^2 + y^2 + z^2}$$

For working with the accelerometer, you can imagine the phone as defining a three-dimensional coordinate system. No matter how the phone is oriented, the positive Y axis points from the bottom of the phone (with the buttons) to the top, the positive X axis points from left to right,



This is a traditional three-dimensional coordinate system, the same coordinate system used in XNA 3D programming. It's termed a *right-hand* coordinate system. Point the index finger of your right hand to increasing X, the middle finger to increase Y, and your thumb points to increasing Z. Or, curve the fingers of your right hand from the positive X axis to the positive Y axis. Your thumb again points to increasing Z.

This coordinate system remains fixed relative to the phone regardless how you hold the phone, and regardless of the orientation of any programs running on the phone. In fact, as you might expect, the accelerometer is the basis for performing orientation changes of Windows Phone 7 applications.

When the phone is still, the accelerometer vector points towards the Earth. The magnitude is 1, meaning 1 *g*, which is the force of gravity on the earth's surface. When holding your phone in the upright position, the acceleration vector is **(0, –1, 0)**, that is, straight down.

Turn the phone 90° counter-clockwise (called landscape left) and the acceleration vector is **(–1, 0, 0)**, upside down it's **(0, 1, 0)**, and another 90° counter-clockwise turn brings you to the landscape right orientation and an accelerometer value of **(1, 0, 0)**. Sit the phone down on the desk with the display facing up, and the acceleration vector is **(0, 0, –1)**. (That final value is what the Windows Phone 7 emulator always reports.)

Of course, the acceleration vector will *rarely* be those exact values, and even the magnitude won't be exact. For a still phone, the magnitude may vary by 10% with different orientations. When you visit the Moon with your Windows Phone 7, you can expect acceleration vector magnitudes in the region of 0.17 but limited cell phone reception.

I've been describing values of the acceleration vector when the device is still. The acceleration vector can point in other directions (and the magnitude can become larger or smaller) when the phone is accelerating, that is, gaining or losing velocity. For example, if you jerk the phone to the left, the acceleration vector points to the right but only when the device is gaining velocity. As the velocity stabilizes, the acceleration vector again registers only gravity. When you decelerate this jerk to the left, the acceleration vector goes to the left briefly as the device comes to a stop.

If the phone is in free fall, the magnitude of the accelerometer vector should theoretically go down to zero.

To use the accelerometer, you'll need a reference to the Microsoft.Devices.Sensors library, and a *using* directive for the *Microsoft.Devices.Sensors* namespace. In WMAppManifest.xml, you need

```
<Capability Name="ID_CAP_SENSORS" />
```

You create an instance of the *Accelerometer* class, set an event handler for the *ReadingChanging* event, and call *Start*.

And then it gets a little tricky. Let's take a look at a project named SilverlightAccelerometer. project that simply displays the current reading in its content grid. A centered *TextBlock* is defined in the XAML file:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

This is a program that will display the accelerometer vector throughout its lifetime, so it creates the *Accelerometer* class in its constructor and calls *Start*:

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

```
public MainPage()
{
    InitializeComponent();

    Accelerometer acc = new Accelerometer();
    acc.ReadingChanged += OnAccelerometerReadingChanged;

    try
    {
        acc.Start();
    }
    catch (Exception exc)
    {
        txtblk.Text = exc.Message;
    }
}
```

The documentation warns that calling *Start* might raise an exception, so the program protects itself against that eventuality. The *Accelerometer* also supports *Stop* and *Dispose* methods, but this program doesn't make use of them. A *State* property is also available if you need to know if the accelerometer is available and what it's currently doing.

A *ReadingChanged* event is accompanied by the *AccelerometerReadingEventArgs* event arguments. The object has properties named *X*, *Y*, and *Z* of type *double* and *TimeStamp* of type *DateTimeOffset*. In the SilverlightAccelerometer program, the job of the event handler is to format this information into a string and set it to the *Text* property of the *TextBlock*.

The catch here is that the event handler (in this case *OnAccelerometerReadingChanged*) is called on a different thread of execution, and this means it must be handled in a special way.

A little background: All the user-interface elements and objects in a Silverlight application are created and accessed in a main thread of execution often called the *user interface thread* or the *UI thread*. These user-interface objects are not thread safe; they are not built to be accessed simultaneously from multiple threads. For this reason, Silverlight will not allow you to access a user-interface object from a non-UI thread.

This means that the *OnAccelerometerReadingChanged* method cannot directly access the *TextBlock* element to set a new value to its *Text* property.

Fortunately, there's a solution involving a class named *Dispatcher* defined in the *System.Windows.Threading* namespace. Through the *Dispatcher* class, you can post jobs from a non-UI thread on a queue where they are later executed by the UI thread. This process sounds complex, but from the programmer's perspective it's fairly easy because these jobs take the form of simple method calls.

An instance of this *Dispatcher* is readily available. The *DependencyObject* class defines a property named *Dispatcher* of type *Dispatcher*, and many Silverlight classes derive from *DependencyObject*. Instances of all of these classes can be accessed from non-UI threads because they all have *Dispatcher* properties. You can use any *Dispatcher* object from any *DependencyObject* derivative created in your UI thread. They are all the same.

The *Dispatcher* class defines a method named *CheckAccess* that returns *true* if you can access a particular user interface object from the current thread. (The *CheckAccess* method is also duplicated by *DependencyObject* itself.) If an object can't be accessed from the current thread, then *Dispatcher* provides two versions of a method named *Invoke* that you use to post the job to the UI thread.

The SilverlightAccelerometer project implements a syntactically elaborate version of the code, but then I'll show you how to chop it down in size.

The verbose version requires a delegate and a method defined in accordance with that delegate. The delegate (and method) should have no return value, but as many arguments as you need to do the job, in this case setting a string to the *Text* property of a *TextBlock*:

Project: SilverlightAccelerometer File: MainPage.xaml.cs (excerpt)

```
delegate void SetTextBlockTextDelegate(TextBlock txtblk, string text);

void SetTextBlockText(TextBlock txtblk, string text)
{
    txtblk.Text = text;
}
```

The *OnAccelerometerReadingChanged* is responsible for calling *SetTextBlockText*. It first makes use of *CheckAccess* to see if it can just call the *SetTextBlockText* method directly. If not, then the handler calls the *BeginInvoke* method. The first argument is an instantiation of the delegate with the *SetTextBlockText* method; this is followed by all the arguments that *SetTextBlockText* requires:

Project: SilverlightAccelerometer File: MainPage.xaml.cs (excerpt)

```
void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    string str = String.Format("X = {0:F2}\n" +
                               "Y = {1:F2}\n" +
                               "Z = {2:F2}\n\n" +
                               "Magnitude = {3:F2}\n\n" +
                               "{4}",
                               args.X, args.Y, args.Z,
                               Math.Sqrt(args.X * args.X + args.Y * args.Y +
                                                        args.Z * args.Z),
                               args.Timestamp);

    if (txtblk.CheckAccess())
    {
        SetTextBlockText(txtblk, str);
    }
    else
    {
        txtblk.Dispatcher.BeginInvoke(new
SetTextBlockTextDelegate(SetTextBlockText),
                                      txtblk, str);
    }
}
```

This is not too bad, but the need for the code to jump across threads has necessitated an additional method and a delegate. Is there a way to do the whole job right in the event handler?

Yes! The *BeginInvoke* method has an overload that accepts an *Action* delegate, which defines a method that has no return value and no arguments. You can create an anonymous method right in the *BeginInvoke* call. The complete code following the creation of the string object looks like this:

```
if (txtblk.CheckAccess())
{
    txtblk.Text = str;
}
else
{
    txtblk.Dispatcher.BeginInvoke(delegate()
    {
        txtblk.Text = str;
    });
}
```

The anonymous method begins with the keyword *delegate* and concludes with the curly brace following the method body. The empty parentheses following the *delegate* keyword are not required.

The anonymous method can also be defined using a lambda expression:

```
if (txtblk.CheckAccess())
{
    txtblk.Text = str;
}
else
{
    txtblk.Dispatcher.BeginInvoke(() =>
    {
        txtblk.Text = str;
    });
}
```

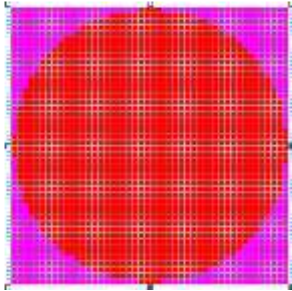
The duplicated code that sets *str* to the *Text* property of *TextBlock* looks a little ugly here (and would be undesirable if it involved more than just one statement), but don't really need to call *CheckAccess*. You can just call *BeginInvoke* and nothing bad will happen even if you are calling it from the UI thead.

The Windows Phone 7 emulator doesn't contain any actual accelerometer, so it always reports a value of (0, 0, –1), which indicates the phone is lying on a flat surface:

A Simple Bubble Level

One handy tool found in any workshop is a bubble level, also called a spirit level. A little bubble always floats to the top of a liquid, so it visually indicates whether something is parallel or orthogonal to the earth, or tilted in some way.

The XnaAccelerometer project includes a 48-by-48 pixel bitmap named Bubble.bmp that consists of a red circle:



The magenta on the corners makes those areas of the bitmap transparent when XNA renders it.

The fields in the *Game1* class mostly involve variables necessary to position that bitmap on the screen:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float BUBBLE_RADIUS_MAX = 25;
    const float BUBBLE_RADIUS_MIN = 12;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Vector2 screenCenter;
    float screenRadius;     // less BUBBLE_RADIUS_MAX

    Texture2D bubbleTexture;
    Vector2 bubbleCenter;
    Vector2 bubblePosition;
    float bubbleScale;

    Vector3 accelerometerVector;
    object accelerometerVectorLock = new object();
     …

}
```

Towards the bottom you'll see a field named *acclerationVector* of type *Vector3*. The *OnAccelerometerReadingChanged* event handler will store a new value in that field, and the *Update* method will utilize the value in calculating a position for a bitmap.

OnAccelerometerReadingChanged and *Update* run in separate threads. One is setting the field; the other is accessing the field. This is no problem if the field is set or accessed in a single machine code instruction. That would be the case if *Vector3* were a class, which is a reference type and basically referenced with something akin to a pointer. But *Vector3* is a structure (a

value type) consisting of three properties of type *float*, each of which occupies four bytes, for a total of 12 bytes or 96 bits. Setting or accessing this *Vector3* field requires this many bits to be transferred.

A Windows Phone 7 device contains at least a 32-bit ARM processor, and a brief glance at the ARM instruction set does not reveal any machine code that would perform a 12-byte memory transfer in one instruction. This means that the accelerometer thread storing a new *Vector3* value could be interrupted midway in the process by the *Update* method in the program's main thread when it retrieves that value. The resultant value might have *X*, *Y*, and *Z* values mixed up from two readings.

While that could hardly be classified as a catastrophe in this program, let's play it entirely safe and use the C# *lock* statement to make sure the *Vector3* value is stored and retrieved by the two threads without interruption. That's the purpose of the *accelerometerVectorLock* variable among the fields.

I chose to create the *Accelerometer* object and set the event handler in the *Initialize* method:

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

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try
    {
        accelerometer.Start();
    }
    catch
    {
    }

    base.Initialize();
}

void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accelerometerVectorLock)
    {
        accelerometerVector = new Vector3((float)args.X, (float)args.Y,
(float)args.Z);
    }
}
```

Notice that the event handler uses the *lock* statement to set the *accelerometerVector* field. That prevents code in the *Update* method from accessing the field during this short duration.

The **LoadContent** method loads the bitmap used for the bubble and initializes several variables used for positioning the bitmap:

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

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

    Viewport viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);
    screenRadius = Math.Min(screenCenter.X, screenCenter.Y) - BUBBLE_RADIUS_MAX;

    bubbleTexture = this.Content.Load<Texture2D>("Bubble");
    bubbleCenter = new Vector2(bubbleTexture.Width / 2, bubbleTexture.Height / 2);
}
```

When the *X* and *Y* properties of accelerometer are zero, the bubble is displayed in the center of the screen. That's the reason for both *screenCenter* and *bubbleCenter*. The *screenRadius* value is the distance from the center when the magnitude of the *X* and *Y* components is 1.

The Update method safely access the accelerometerVector field and calculates *bubblePosition* based on the *X* and *Y* components. It might seem like I've mixed up the *X* and *Y* components in the calculation, but that's because the default screen orientation is portrait in XNA, so it's opposite the coordinates of the acceleration vector.

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

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

    Vector3 accVector;

    lock (accelerometerVectorLock)
    {
        accVector = accelerometerVector;
    }

    bubblePosition = new Vector2(screenCenter.X + screenRadius * accVector.Y,
                                 screenCenter.Y + screenRadius * accVector.X);
    float bubbleRadius = BUBBLE_RADIUS_MIN + (1 - accVector.Z) / 2 *
                            (BUBBLE_RADIUS_MAX - BUBBLE_RADIUS_MIN);
    bubbleScale = bubbleRadius / (bubbleTexture.Width / 2);

    base.Update(gameTime);
}
```

In addition, a *bubbleScale* factor is calculated based on the *Z* component of the vector. The idea is that the bubble is largest when the screen is facing up and smallest when the screen is facing down, as if the screen is really one side of a rectangular pool of liquid that extends below the phone, and the size of the bubble indicates how far it is from the surface.

The *Draw* override uses a long version of the *Draw* method of *SpriteBatch*.

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

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

    spriteBatch.Begin();
    spriteBatch.Draw(bubbleTexture, bubblePosition, null, Color.White, 0,
                     bubbleCenter, bubbleScale, SpriteEffects.None, 0);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

Notice the *bubbleScale* argument, which scales the bitmap to a particular size. The center of scaling is provided by the previous argument to the method, *bubbleCenter*. That point is also aligned with the *bubblePosition* value relative to the screen.

The program doesn't look like much, and is even more boring running on the emulator:

Geographic Location

With the user's permission, a Windows Phone 7 program can obtain the geographic location of the phone using a technique called Assisted-GPS or A-GPS.

The most accurate method of determining location is accessing signals from Global Positioning System (GPS) satellites. However, GPS can be slow. It doesn't work well in cities, and doesn't work at all indoors, and it's considered expensive in terms of battery use. To work more cheaply and quickly, an A-GPS system can attempt to determine location from cell-phone towers or the network. These methods are faster and more reliable, but less accurate.

The core class involved in location detection is *GeoCoordinateWatcher*. You'll need a reference to the System.Device assembly and a *using* direction for the *System.Device.Location* namespace. The WMAppManifest.xml file requires the tag:

```
<Capability Name="ID_CAP_LOCATION" />
```

The *GeoCoordinateWatcher* constructor optionally takes a member of the *GeoPositionAccuracy* enumeration:

- *Default*
- *High*

After creating a *GeoCoordinateWatcher* object, you'll want to install a handler for the *PositionChanged* event and call *Start*. The *PositionChanged* event delivers a *GeoCoordinate* object that has eight properties:

- *Latitude*, a *double* between –90 and 90 degrees
- *Longitude*, a *double* between –180 and 180 degrees
- *Altitude* of type *double*
- *HorizontalAccuracy* and *VerticalAccuracy* of type *double*
- *Course*, a *double* between 0 and 360 degrees
- *Speed* of type *double*
- *IsUnknown*, a Boolean that is *true* if the *Latitude* or *Longitude* is not a number

If the application does not have permission to get the location, then *Latitude* and *Longitude* will be *Double.NaN*, and *IsUnknown* will be *true*.

In addition, *GeoCoordinate* has a *GetDistanceTo* method that calculates the distance between two *GeoCoordinate* objects.

I'm going to focus on the first two properties, which together are referred to as *geographic coordinates* to indicate a point on the surface of the Earth. Latitude is the angular distance from the equator. In common usage, latitude is an angle between 0 and 90 degrees and followed with either N or S meaning north or south. For example, the latitude of New York City is approximately 40°N. In the *GeoCoordinate* object, latitudes north of the equator are positive values and south of the equator are negative values, so that 90° is the North Pole and –90° is the South Pole.

All locations with the same latitude define a *line of latitude*. Along a particular line of latitude, longitude is the angular distance from the Prime Meridian, which passes through the Royal Observatory at Greenwich England. In common use, longitudes are either east or west. New

York City is 74°W because it's west of the Prime Meridian. In a *GeoCoordinate* object, positive longitude values denote east and negative values are west. Longitude values of 180 and –180 meet up at the International Date Line.

Although the *System.Device.Location* namespace includes classes that use the geographic coordinates to determine civic address (streets and cities), these are not implemented in the initial release of Windows Phone 7.

The XnaLocation project simply displays numeric values.

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont kootenay14;
    string text = "Obtaining location...";
    Viewport viewport;
    Vector2 textPosition;
    …
}
```

As with the accelerometer, I chose to create and initialize the *GeoCoordinateWatcher* in the *Initialize* override. The event handler is called in the same thread, so nothing special needs to be done to format the results in a string:

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

```
protected override void Initialize()
{
    GeoCoordinateWatcher geoWatcher = new GeoCoordinateWatcher();
    geoWatcher.PositionChanged += OnGeoWatcherPositionChanged;
    geoWatcher.Start();

    base.Initialize();
}

void OnGeoWatcherPositionChanged(object sender,
                                 GeoPositionChangedEventArgs<GeoCoordinate> args)
{
    text = String.Format("Latitude: {0}\r\n" +
                         "Longitude: {1}\r\n" +
                         "Altitude: {2}\r\n\r\n" +
                         "{3}",
                         args.Position.Location.Latitude,
                         args.Position.Location.Longitude,
                         args.Position.Location.Altitude,
```

```
                         args.Position.Timestamp);
}
```

The *LoadContent* method simply obtains the font and saves the *Viewport* for later text positioning:

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

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    viewport = this.GraphicsDevice.Viewport;

    System.Diagnostics.Debug.WriteLine(viewport);
}
```

The size of the displayed string could be different depending on different values. That's why the position of the string is calculated from its size and the *Viewport* values in the *Update* method:

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

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

    Vector2 textSize = kootenay14.MeasureString(text);
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);
    base.Update(gameTime);
}
```

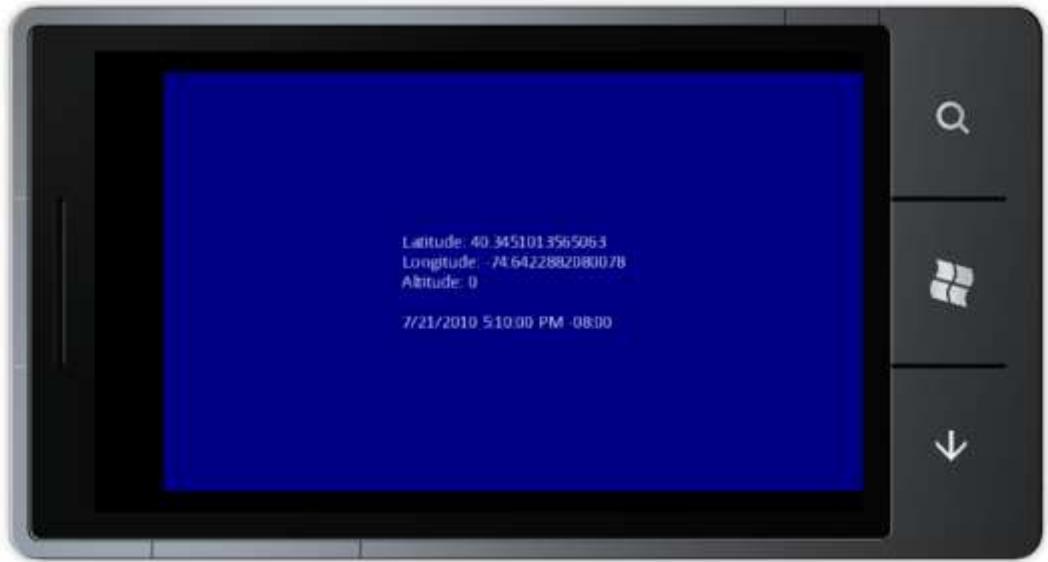
The *Draw* method is trivial:

XNA Project: XnaLocation   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);
}
```

Because the *GeoCoordinateWatcher* is left running for the duration of the program, it should update the location as the phone is moved. With the emulator, however, the *GeoCoordinateWatcher* always returns the coordinates of a spot in Princeton, New Jersey, perhaps as a subtle reference to the college where Alan Turing earned his PhD.



Using a Map Service

Of course, most people curious about their location prefer to see a map rather than numeric coordinates. The Silverlight demonstration of the location service displays a map that comes to the program in the form of bitmaps.

In a real phone application, you'd probably be using Bing Maps or perhaps another online mapping service. Unfortunately, making use of Bing Maps in a program involves opening a developer account, and getting a maps key and a credential token. This is all free and straightforward but it doesn't work well for a program that will shared among all the readers of a book.

For that reason, I'll be using an alternative that doesn't require keys or tokens. This alternative is Microsoft Research Maps, which you can learn all about at *msrmaps.com*. The aerial images are provided by the United States Geological Survey (USGS). Microsoft Research Maps makes these images available through a web service called MSR Maps Service, but still sometimes referred to by its old name of TerraService.

The downside is that the images are not quite state-of-the-art and the service doesn't seem entirely reliable.

MSR Maps Service is a SOAP (Simple Object Access Protocol) service with the transactions described in a WSDL (Web Services Description Language) file. Behind the scenes, all the transactions between your program and the web service are in the form of XML files. However, to avoid programmer anguish, generally the WSDL file is used to generate a *proxy*, which is a collection of classes and structures that allow your program to communicate with the web service with method calls and events.

You can generate this proxy right in Visual Studio. Here's how I did it: I first created an Windows Phone 7 project in Visual Studio called SilverlightLocationMapper. In the Solution Explorer, I right-clicked the project name and selected Add Service Reference. In the Address field I entered the URL of the MSR Maps Service WSDL file: *http://MSRMaps.com/TerraService2.asmx*.

(You might wonder if the URL should be *http://msrmaps.com/TerraService2.asmx?WSDL* because that's how WSDL files are often referenced. That address will actually seem to work at first, but you'll get files containing obsolete URLs.)

After you've entered the URL in the Address field, press Go. Visual Studio will access the site and report back what it finds. There will be one service, called by the old name of TerraService.

Next you'll want to enter a name in the Namespace field to replace the generic ServiceReference1. I used MsrMapsService and pressed OK.

You'll then see MsrMapsService show up under the project in the Solution Explorer. If you click the little Show All Files icon at the top of the Solution Explorer, you can view the generated files. In particular, you'll see Reference.cs, a big file (over 3000 lines) with a namespace of XnaLocationMapper.MsrMapsService, which combines the original project name and the name you selected for the web service.

This Reference.cs file contains all the classes and structures you need to access the web service, and which are documented on the *msrmaps.com* web site. To access these classes in your program, add a *using* direction:

```
using SilverlightLocationMapper.MsrMapsService;
```

You also need a using directive for *System.Device.Location* and a reference to the System.Device assembly.

In the MainPage.xaml file, I left the *SupportedOrientations* property at its default setting of *Portrait*, I removed the page title to free up more space, and I moved the title panel below the content grid just in case there was a danger of something spilling out of the content grid and

obscuring the title. Moving the title panel below the content grid in the XAML file ensures that it will be visually on top.

Here's the content grid:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="statusText"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               TextWrapping="Wrap" />

    <Image Source="Images/usgslogoFooter.png"
           Stretch="None"
           HorizontalAlignment="Right"
           VerticalAlignment="Bottom" />
</Grid>
```

The *TextBlock* is used to display status and (possibly) errors; the *Image* displays a logo of the United States Geological Survey.

The map bitmaps will be inserted between the *TextBlock* and *Image* so they obscure the *TextBlock* but the *Image* remains on top.

The code-behind file has just two fields, one for the *GeoCoordinateWatcher* that supplies the location information, and the other for the proxy class created when the web service was added:

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

```
public partial class MainPage : PhoneApplicationPage
{
    GeoCoordinateWatcher geoWatcher = new GeoCoordinateWatcher();
    TerraServiceSoapClient proxy = new TerraServiceSoapClient();
    …
}
```

You use the proxy by calling its methods, which make network requests. All these methods are asynchronous. For each method you call, you must also supply a handler for a completion event that is fired when the information you requested has been transferred to your application.

The completion event is accompanied by event arguments: a *Cancelled* property of type *bool*, an *Error* property that is *null* if there is no error, and a *Result* property that depends on the request.

I wanted the process to begin after the program was loaded and displayed, so I set a handler for the *Loaded* event. That handler sets the handlers for the two completion events I'll require of the proxy, and also starts up the *GeoCoordinateWatcher*:

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

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

void OnMainPageLoaded(object sender, RoutedEventArgs args)
{
    // Set event handlers for TerraServiceSoapClient proxy
    proxy.GetAreaFromPtCompleted += OnProxyGetAreaFromPtCompleted;
    proxy.GetTileCompleted += OnProxyGetTileCompleted;

    // Start GeoCoordinateWatcher going
    statusText.Text = "Obtaining geographic location...";
    geoWatcher.PositionChanged += OnGeoWatcherPositionChanged;
    geoWatcher.Start();
}
```

When coordinates are obtained, the following *OnGeoWatcherPositionChanged* method is called. This method begins by turning off the *GeoCoordinateWatcher*. The program is not equipped to continuously update the display, so it can't do anything with any additional location information. It appends the longitude and latitude to the *TextBlock* called *ApplicationTitle* displayed at the top of the screen.

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

```
void OnGeoWatcherPositionChanged(object sender,
                                 GeoPositionChangedEventArgs<GeoCoordinate> args)
{
    // Turn off GeoWatcher
    geoWatcher.PositionChanged -= OnGeoWatcherPositionChanged;
    geoWatcher.Stop();

    // Set coordinates to title text
    GeoCoordinate coord = args.Position.Location;
    ApplicationTitle.Text += ": " + String.Format("{0:F2}°{1} {2:F2}°{3}",
                                                  Math.Abs(coord.Latitude),
                                                  coord.Latitude > 0 ? 'N' : 'S',
                                                  Math.Abs(coord.Longitude),
                                                  coord.Longitude > 0 ? 'E' : 'W');
    // Query proxy for AreaBoundingBox
    LonLatPt center = new LonLatPt();
    center.Lon = args.Position.Location.Longitude;
```

```
    center.Lat = args.Position.Location.Latitude;

    statusText.Text = "Accessing Microsoft Research Maps Service...";
    proxy.GetAreaFromPtAsync(center, 1, Scale.Scale4m, (int)ContentGrid.ActualWidth,

(int)ContentGrid.ActualHeight);
}
```

The method concludes by making its first call to the proxy. The *GetAreaFromPtAsync* call requires a longitude and latitude as a center point, but some other information as well. The second argument is 1 to get an aerial view and 2 for a map (as you'll see at the end of this chapter). The third argument is the desired scale, a member of the *Scale* enumeration. The member I've chosen means that each pixel of the returned bitmaps is equivalent to 4 meters.

Watch out: Some scaling factors—in particular, *Scale2m*, *Scale8m*, and *Scale32m*—result in GIF files being returned. Remember, remember, remember that Silverlight doesn't do GIF! For the other scaling factors, JPEGS are returned.

The final arguments to GetAreaFromPtAsync are the width and height of the area you wish to cover with the map.

All the bitmaps you get back from the MSR Maps Service are 200 pixels square. Almost always, you'll need multiple bitmaps to tile a complete area. For example, if the last two arguments to *GetAreaFromPtAsync* are 400 and 600, you'll need 6 bitmaps to tile the area.

Well, actually not: An area of 400 pixels by 600 pixels will require 12 bitmaps, 3 horizontally and 4 vertically.

Here's the catch: These bitmaps aren't specially created when a program requests them. They already exist on the server in all the various scales. The geographic coordinates where these bitmaps begin and end are fixed. So if you want to cover a particular area of your display with a tiled map, and you want the center of this area to be precisely the coordinate you specify, the existing tiles aren't going to fit exactly. You want sufficient tiles to cover your area, but the tiles around the boundary are going to hang over the edges.

What you get back from the *GetAreaFromPtAsync* call (in the following *OnProxyGetAreaFromPtCompleted* method) is an object of type *AreaBoundingBox*. This is a rather complex structure that nonetheless has all the information required to request the individual tiles you need and then assemble them together in a grid.

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

```
void OnProxyGetAreaFromPtCompleted(object sender, GetAreaFromPtCompletedEventArgs
args)
{
    if (args.Error != null)
```

```
    {
        statusText.Text = args.Error.Message;
        return;
    }

    statusText.Text = "Getting map tiles...";

    AreaBoundingBox box = args.Result;
    int xBeg = box.NorthWest.TileMeta.Id.X;
    int yBeg = box.NorthWest.TileMeta.Id.Y;
    int xEnd = box.NorthEast.TileMeta.Id.X;
    int yEnd = box.SouthWest.TileMeta.Id.Y;

    // Loop through the tiles
    for (int x = xBeg; x <= xEnd; x++)
        for (int y = yBeg; y >= yEnd; y--)
        {
            // Create Image object to display tile
            Image img = new Image();
            img.Stretch = Stretch.None;
            img.HorizontalAlignment = HorizontalAlignment.Left;
            img.VerticalAlignment = VerticalAlignment.Top;
            img.Margin = new Thickness((x - xBeg) * 200 -
box.NorthWest.Offset.XOffset,
                                       (yBeg - y) * 200 -
box.NorthWest.Offset.YOffset,
                                       0, 0);

            // Insert after TextBlock but before Image with logo
            ContentGrid.Children.Insert(1, img);

            // Define the tile ID
            TileId tileId = box.NorthWest.TileMeta.Id;
            tileId.X = x;
            tileId.Y = y;

            // Call proxy to get the tile (Notice that Image is user object)
            proxy.GetTileAsync(tileId, img);
        }
}
```

I won't discuss the intricacies of *AreaBoundingBox* because it's more or less documented on the *msrmaps.com* web site, and I was greatly assisted by some similar logic on the site written for Windows Forms (which I suppose dates it a bit).

Notice that the loop creates each *Image* object to display each title. Each of these *Image* objects has the same *Stretch*, *HorizontalAlignment*, and *VerticalAlignment* properties, but a different *Margin*. This *Margin* is how the individual tiles are positioned within the content grid. The *XOffset* and *YOffset* values cause the tiles to hang off the top and left edges of the content grid. The content grid doesn't clip its contents, so these tiles possibly extend to the top of the program's page.

Notice also that each *Image* object is passed as a second argument to the proxy's *GetTileAsync* method. This is called the *UserState* argument. The proxy doesn't do anything with this argument except return it as the *UserState* property of the completion arguments, as shown here:

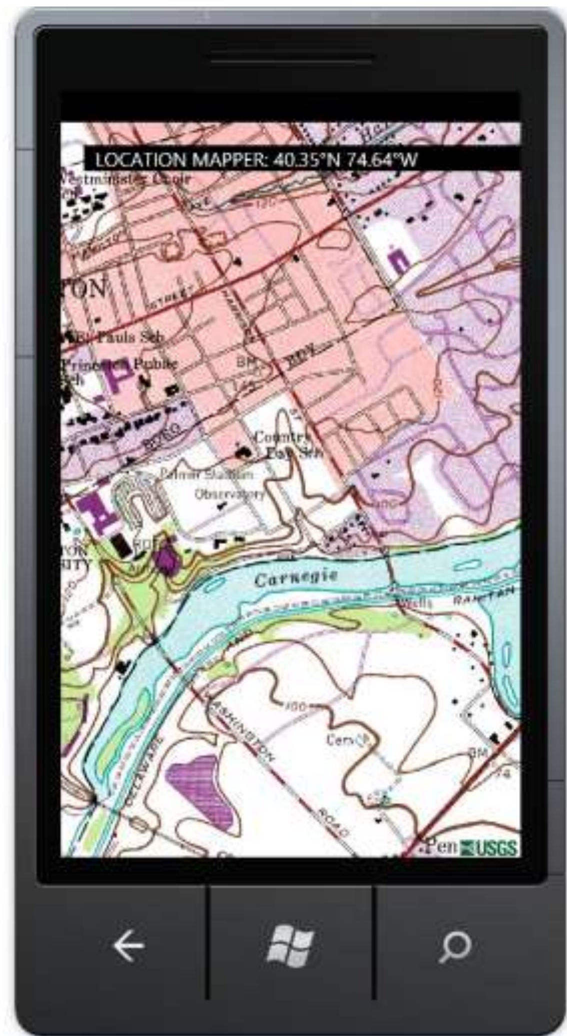
Silverlight Project: SilverlightLocationManager   File: MainPage.xaml.cs (excerpt)

```
void OnProxyGetTileCompleted(object sender, GetTileCompletedEventArgs args)
{
    if (args.Error != null)
    {
        return;
    }

    Image img = args.UserState as Image;
    BitmapImage bmp = new BitmapImage();
    bmp.SetSource(new MemoryStream(args.Result));
    img.Source = bmp;
}
```

That's how the method links up the particular bitmap tile with the particular *Image* element already in place in the content grid.

It is my experience that in most cases, the program doesn't get all the tiles it requests. If you're very lucky, your display might look like this:

If you change the second argument of the *proxy.GetAreaFromPtAsync* call from a 1 to a 2, you get back images of an actual map rather than an aerial view:



It has a certain retro charm—and I love the watercolor look—but I'm afraid that modern users are accustomed to something just a little more 21st century.

Chapter 6

Issues in Application Architecture

Over the past few decades, it's been a common desire that our personal computers be able to do more than one thing at a time. But when user interfaces are involved, multitasking is never quite as seamless as we'd like. The Terminate-and-Stay-Resident (TSR) programs of MS-DOS and the cooperative multitasking of early Windows were only the first meager attempts in an ongoing struggle. In theory, process switching is easy. But sharing resources—including the screen and a handful of various input devices—is very hard.

While the average user might marvel at the ability of modern Windows to juggle many different applications at once, we programmers still wrestle with the difficulties of multitasking— carefully coding our UI threads to converse amicably with our non-UI threads, always on the lookout for the hidden treachery of asynchronous operations.

Every new application programming interface we encounter makes a sort of awkward accommodation with the ideals of multitasking, and as we become familiar with the API we also become accustomed to this awkward accommodation, and eventually we might even consider this awkward accommodation to be a proper solution to the problem.

On Windows Phone 7, that awkward accommodation is known as *tombstoning*.

Task Switching on the Phone

We want our phones to be much like our other computers. We want to have a lot of applications available. We want to start up a particular application as soon as we conceive a need for it. While that application is running, we want it to be as fast as possible and have access to unlimited resources. But we want this application to coexist with other running applications because we want to be able to jump among multiple applications running on the machine.

Arbitrarily jumping among multiple running applications is impractical on the phone. It would require some kind of display showing all the currently running applications, much like the Windows taskbar. Either this taskbar would have to be constantly visible—taking valuable screen space away from the active applications—or a special button or command would need to be assigned to display the taskbar or task list.

Instead, Windows Phone 7 manages multiple active applications by implementing a stack. You can think of the phone as an old-fashioned web browser with no tab feature and no Forward button. But it does have a Back button and it also has a Start button, which brings you to Start screen and allows you to launch a program.

Suppose you choose to launch a program called Analyze. You work a little with Analyze and then decide you're finished. You press the Back button. The Analyze program is terminated and you're back at the Start screen. That's the simple scenario.

Later you decide you need to run Analyze again. While you're using Analyze, you need to check something on the Web. You press the Start button to get to the Start screen and select Internet Explorer. While you're browsing, you remember you haven't played any games recently. You press the Start button, select Backgammon and play a little of that. While playing Backgammon, you wonder about the odds of a particular move, so you press the Start button again and run Calc. Then you feel guilty about not doing any work, so you press the Start button again and run Draft.

Draft is a Silverlight program with multiple pages. From the main page, you navigate to several other pages.

Now start pressing the Back button. You go backwards through all the pages you visited in the Draft program, then Draft is terminated as you go back to Calc. Calc still displays the remnants of your work, and Calc is terminated as you go back to Backgammon, which shows a game in progress, and Backgammon is terminated as you go back to Internet Explorer, and again you go backwards through any Web pages you may have visited, and IE is terminated as you go back to Analyze, and Analyze is terminated as you go back to the Start screen. The stack is now empty.

This type of navigation is a good compromise for small devices, and it's consistent with users' experiences in web browsing. The stack is conceptually very simple: The Start button pushes the current application on the stack so a new application can be run; the Back button terminates the current application and pops the next one off the stack.

However, the limited resources of the phone convinced the Windows Phone 7 developers that applications on the stack should consume as few resources as possible. For this reason, an application put on the stack does not continue plugging away at work. It's not even put into a suspended state of some sort. Something more severe than that happens. The process is actually terminated. When this terminated program comes off the stack, it is then re-executed from scratch.

This is tombstoning. The application is killed but then allowed to come back to life.

You've probably seen enough movies to know that reanimating a corpse can be a very scary proposition. Almost always the hideous thing that arises out of the filthy grave is not the clean and manicured loved one who went in.

The trick here is to persuade the disinterred program to look and feel much the same as when it was last alive and the user interacted with it. This process is a collaboration between you and Windows Phone 7. The phone gives you the tools (a place to put some data); your job is

to use the tools to restore your program to a presentable state. Ideally the user should have no idea that it's a completely new process.

For some applications, resurrection doesn't have to be 100% successful. We all have experience with navigating among Web pages to know what's acceptable and what's not. For example, suppose you visit a long Web page, and you scroll down a ways, then you navigate to another page. When you go back to the original page, it's not too upsetting if it's lost your place and you're back at the top of the page.

On the other hand, if you've just spent 10 minutes filling out a large form, you definitely do *not* want to see all your work gone after another page tells you that you've made one tiny error.

Let's nail down some terminology that's consistent with some events I'll discuss later:

When an application is run from the Start screen, it is said to be *launched*. When it is terminated as a result of the Back button, it is *closed*. When the program is running and the user presses the Start button, the program is said to be *deactivated*, even though it really is quite dead. This is the tombstoned state. When a program comes out of tombstoning as the user navigates back to it, it is said to be *activated*, even though it's really starting up from scratch.

Page State

The *SilverlightFlawedTombstoning* project is a simple Silverlight program that responds to taps on the screen by changing the background of *ContentGrid* to a random color, and displaying the total number of taps in its page title. Everything of interest happens in the code-behind file:

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

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    int numTaps = 0;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentGrid.Background =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
```

```
                                                    (byte)rand.Next(256)));
        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }
}
```

The little *UpdatePageTitle* method is called from both the program's constructor (where it always results in displaying a value of 0) and from the *OnManipulationStarted* override.

You probably didn't get enough practice using Visual Studio with tombstoned programs in Chapter 4, so here's the routine: Build and deploy the program to the phone emulator by pressing F5 (or selecting Start Debugging from the Debug menu). When the program comes up, tap the screen several times to change the color and bump up the tap count. Now press the Start button. You can see from Visual Studio that the program has terminated, but to the phone it's actually been deactivated and tombstoned.

Now press the Back button to return to the program. You'll see a blank screen, and you have 10 seconds to press F5 in Visual Studio again to reconnect the debugger with the program.

However, when the program comes back into view, you'll see that the color and the number of taps have been lost. All your hard work! Totally gone! This is not a good way for a program to emerge from tombstoning. It is this state data that we want to preserve when the program is flat-lined.

An excellent opportunity to save and reload state data for a page is through overrides of the *OnNavigatedTo* and *OnNavigatedFrom* methods defined by the *Page* class from which *PhoneApplicationPage* derives. These methods are called when a page is brought into view by being loaded by the frame, and when the page is detached from the frame.

Using these methods is particularly appropriate if your Silverlight application will have multiple pages that the user can navigate among. You'll find that a new instance of *PhoneApplicationPage* is created every time a user navigates to a page, so you'll probably want to save and reload page state data for normal navigation anyway. By overriding *OnNavigatedTo* and *OnNavigatedFrom* you're effectively solving two problems with one solution. When overriding these methods, you'll want a *using* directive for *System.Windows.Navigation* because that's where the event arguments are defined.

Although Windows Phone 7 leaves much of the responsibility for restoring a tombstoned application to the program itself, it will cause the correct page to be loaded on activation, so

it's possible that a page-oriented Silverlight program that saves and restores page state data during *OnNavigatedTo* and *OnNavigatedFrom* will need no special processing for tombstoning.

Windows Phone 7 provides a special way to save page state data during tombstoning. This is the *State* property of *PhoneApplicationService*, a class defined in the *Microsoft.Phone.Shell* namespace. This *State* property is of type *IDictionary<string*, *object>*. You store objects in this dictionary using text keys. The phone operating system preserves this *State* property during the time a program is deactivated and tombstoned, but gets rid of it when the program closes and is terminated for real.

Do not create a *PhoneApplicationService* object. One is already created for your application and is accessible through the static *PhoneApplicationService.Current* property.

Any object you store in this *State* dictionary must be serializable, that is, it must be possible to convert the object into XML, and recreate the object from XML. It must have a public parameterless constructor, and all its public properties must either be serializable or be of types that have *Parse* methods to convert the strings back to objects.

It's not always obvious what objects are serializable and which ones are not. When I first wrote the SilverlightBetterTombstoning project (shown below), I tried to store both the *numTaps* value and the *SolidColorBrush* in the *State* dictionary. The program raised an exception that said "Type 'System.Windows.Media.Transform' cannot be serialized." It took awhile to remember that *Brush* has a property named *Transform* of type *Transform*, an abstract class. I had to serialize the *Color* instead.

Here's the complete class:

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

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    int numTaps = 0;
    PhoneApplicationService appService = PhoneApplicationService.Current;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentGrid.Background =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
```

```
        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        appService.State["numTaps"] = numTaps;

        if (ContentGrid.Background is SolidColorBrush)
        {
            appService.State["backgroundColor"] =
                                (ContentGrid.Background as SolidColorBrush).Color;
        }

        base.OnNavigatedFrom(args);
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Load numTaps
        if (appService.State.ContainsKey("numTaps"))
        {
            numTaps = (int)appService.State["numTaps"];
            UpdatePageTitle(numTaps);
        }

        // Load background color
        object obj;

        if (appService.State.TryGetValue("backgroundColor", out obj))
            ContentGrid.Background = new SolidColorBrush((Color)obj);

        base.OnNavigatedTo(args);
    }
}
```

Notice the *appService* field set to *PhoneApplicationService.Current*. That's just for convenience for accessing the *State* property. You can use the long *PhoneApplicationService.Current.State* instead if you prefer.

Storing items in the *State* dictionary is easier than getting them out. The syntax:

```
appService.State["numTaps"] = numTaps;
```

replaces an existing item if the "numTaps" key exists, or adds a new item if the key does not exist. Saving the background color is a little trickier: By default the *Background* property of

ContentGrid is *null*, so the code checks for a non-*null* value before attempting to save the *Color* property.

To get items out of the dictionary, you can't use similar syntax. You'll raise an exception if the key does not exist. (And these keys will *not* exist when the application is launched.) The *OnNavigatedTo* method shows two different standard ways of accessing the items: The first checks if the dictionary contains the key; the second uses *TryGetValue*, which returns *true* if the key exists.

In a real program, you'll probably want to use *string* variables for the keys to avoid accidently typing inconsistent values. (If your typing is impeccable, don't worry about the multiple identical strings taking up storage: Strings are interned, and identical strings are consolidated into one.) You'll probably also want to write some standard routines that perform these jobs.

Try running this program like you ran the earlier one: Press F5 to deploy it to the emulator from Visual Studio. Tap the screen a few times. Press the Start button as if you're going to start a new program. Visual Studio reports that the process has terminated. Now press the Back button. When you see the blank screen, press F5 again in Visual Studio to reconnect the debugger. The settings have been saved and the corpse looks as good as new!

As you experiment, you'll discover that the settings are saved when the application is tombstoned (that is, when you navigate away from the application with the Start button and then return) but not when a new instance starts up from the Start list. This is correct behavior. The operating system discards the *State* dictionary when the program terminates for real.

This page state data is sometimes referred to as "transient" data. It's not data that affects other instances of the application.

If you want some similar data shared among all instances of a program, you probably want to implement what's often called *application settings*. You can do that as well.

Isolated Storage

Every program installed on Windows Phone 7 has access to its own area of permanent disk storage referred to as *isolated storage*, which the program can access using classes in the *System.IO.IsolatedStorage* namespace. Although whole files can be read and written to in isolated storage, I'm going to focus instead on a special use of isolated storage for storing application settings. The *IsolatedStorageSettings* class exists specifically for this purpose.

For application settings, you should be thinking in terms of the whole application rather than a particular page. Perhaps some of the application settings apply to multiple pages. Hence, a good place to deal with these application settings is in the program's *App* class, which derives from *Application*.

Not coincidently, it is the App. xaml file that creates a *PhoneApplicationService* object (the same *PhoneApplicationService* object used for saving transient data) and assigns event handlers for four events:

```
<shell:PhoneApplicationService Launching="Application_Launching"
                               Closing="Application_Closing"
                               Activated="Application_Activated"
                               Deactivated="Application_Deactivated"/>
```

The *Launching* event is fired when the program is first executed from the Start screen. The *Deactivated* event occurs when the program is tombstoned, and the *Activated* event occurs when the program is resurrected from tombstoning. The *Closing* event occurs when the program is really terminated, probably by the user pressing the Back button.

So, when a program starts up, it gets either a *Launching* event or an *Activated* event (but never both), depending whether it's being started from the Start screen or coming out of a tombstoned state. When a program ends, it gets either a *Deactivated* event or a *Closing* event, depending whether it's being tombstoned or terminated for real.

A program should load application settings during the *Launching* event and save them in response to the *Closing* event. That much is obvious. But a program should also save application settings during the *Deactivated* event because the program really doesn't know if it will ever be resurrected. And if it is resurrected, it should load application settings during the *Activated* event because otherwise it won't know about those settings.

Conclusion: application settings should be loaded during the *Launching* and *Activated* events and saved during the *Deactivated* and *Closing* events.

For the SilverlightIsolatedStorage program, I decided that the number of taps should continue to be treated as transient data—part of the state of the page. But the background color should be an application setting and shared among all instances.

In App.xaml.cs—a file that hasn't yet been altered for any of the programs in this book—I defined the following public property:

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

```
public partial class App : Application
{
    // Application settings
    public Brush BackgroundBrush { set; get; }
    …
}
```

Conceivably this can be one of many application settings that are accessible throughout the application.

App.xaml.cs already has empty event handlers for all the *PhoneApplicationService* events. Each handler was given a body consisting of a single method call:

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

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    LoadSettings();
}

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    LoadSettings();
}

private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    SaveSettings();
}

private void Application_Closing(object sender, ClosingEventArgs e)
{
    SaveSettings();
}
```

Here are the *LoadSettings* and *SaveSettings* methods. Both methods obtain an *IsolatedStorageSettings* object. One loads (and the other saves) the *Color* property of the *BackgroundBrush* property with code that is similar to what you saw before.

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

```
void LoadSettings()
{
    IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

    Color clr;

    if (settings.TryGetValue<Color>("backgroundColor", out clr))
        BackgroundBrush = new SolidColorBrush(clr);
}

void SaveSettings()
{
    IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

    if (BackgroundBrush is SolidColorBrush)
    {
        settings["backgroundColor"] = (BackgroundBrush as SolidColorBrush).Color;
    }
}
```

And finally, here's the new MainPage.xaml.cs file. This file—and any other class in the program—can get access to the *App* object using the static *Application.Current* property and casting it to an *App*. The constructor of *MainPage* obtains the *BackgroundBrush* property from the *App* class, and the *OnManipulationStarted* method sets that *BackgroundBrush* property.

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

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    int numTaps = 0;
    PhoneApplicationService appService = PhoneApplicationService.Current;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);

        // Access App class for isolated storage setting
        Brush brush = (Application.Current as App).BackgroundBrush;

        if (brush != null)
            ContentGrid.Background = brush;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        SolidColorBrush brush =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
        ContentGrid.Background = brush;

        // Save to App class for isolated storage setting
        (Application.Current as App).BackgroundBrush = brush;

        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        appService.State["numTaps"] = numTaps;

        base.OnNavigatedFrom(args);
```

```
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Load numTaps
        if (appService.State.ContainsKey("numTaps"))
        {
            numTaps = (int)appService.State["numTaps"];
            UpdatePageTitle(numTaps);
        }
    }
}
```

Because that background color has been upgraded from transient page data to an application setting, references to it have been removed in the *OnNavigatedFrom* and *OnNavigatedTo* overrides.

Obviously as programs get more complex, transient page data and application settings also get more complex, but these techniques illustrate a good starting point.

Part II

Silverlight

Chapter 7
XAML Power and Limitations

As you've seen, a Silverlight program is generally a mix of code and XAML. Most often, you'll use XAML for defining the layout of 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.

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#—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 *almost* all of your Silverlight application entirely in code. However, page navigation is based around the existence of XAML files for classes that derive from *PhoneApplicationPage*, and there also is a very important type of job that *must* be done in XAML. 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 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 an 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 another 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 7!"
               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 TapForTextBlock project creates a new *TextBlock* in code every time you tap the screen. The MainPage.xaml file contains a TextBlock centered with the content grid:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

The code-behind file for *MainPage* creates an additional *TextBlock* whenever you tap the screen. It uses the dimensions of the existing *TextBlock* to set a *Margin* property on the new *TextBlock* elements to randomly position them within the content grid:

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

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

    public MainPage()
    {
        InitializeComponent();
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        TextBlock newTextBlock = new TextBlock();
        newTextBlock.Text = "Hello, Windows Phone 7!";
        newTextBlock.HorizontalAlignment = HorizontalAlignment.Left;
        newTextBlock.VerticalAlignment = VerticalAlignment.Top;
        newTextBlock.Margin = new Thickness(
            (ContentGrid.ActualWidth - txtblk.ActualWidth) * rand.NextDouble(),
            (ContentGrid.ActualHeight - txtblk.ActualHeight) * rand.NextDouble(),
            0, 0);

        ContentGrid.Children.Add(newTextBlock);

        args.Complete();
        args.Handled = true;
        base.OnManipulationStarted(args);
    }
}
```

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.

But this is the type of thing you simply can't do in XAML. XAML can't respond to events, it can't create arbitrarily create new instances of elements, it can't make calls to the *Random* class, and it certainly can't perform arbitrary calculations.

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 newTextBlock = new TextBlock
{
    Text = "Hello, Windows Phone 7!",
    HorizontalAlignment = HorizontalAlignment.Left,
    VerticalAlignment = VerticalAlignment.Top,
    Margin = new Thickness(
            (ContentGrid.ActualWidth - txtblk.ActualWidth) * rand.NextDouble(),
            (ContentGrid.ActualHeight - txtblk.ActualHeight) * rand.NextDouble(),
            0, 0)
};

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

That makes the code look a *little* more like the XAML (except for the calculations and method calls to *rand.NextDouble*), 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.

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 content grid:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone 7!" />
</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 7!"
           FontStyle="Italic" />
```

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

```
<phone:PhoneApplicationPage … FontStyle="Italic" …
```

This *FontStyle* attribute can go anywhere in the *PhoneApplicationPage* tag. Notice that setting the property in this tag affects *all* the *TextBlock* elements on 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 (and now the *FontStyle* property) 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 complain. 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 7!"
           FontStyle="Normal" />
```

Now the text in this particular *TextBlock* 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.

Property-Element Syntax

Let's set the *TextBlock* attributes to these values:

```
<TextBlock Text="Hello, Windows Phone 7!"
           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 7!"
           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 7!"
           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 7!"
           Foreground="Red">
   <!-- Not a legal property-element tag! -->
    <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 7!"
           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 two more:

```
<Application.Resources>
<Application.ApplicationLifetimeObjects>
```

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

Colors and Brushes

Let's return the *TextBlock* to its pristine condition:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone 7!" />
</Grid>
```

The text shows up as white (or black, depending on the theme your selected) because the *Foreground* property is set on the root element in MainPage.xaml. You can override the user's preferences by setting *Background* for the *Grid* and *Foreground* for the *TextBlock*:

```
<Grid x:Name="ContentGrid" Background="Blue" Grid.Row="1">
    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</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*.

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 numerically 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 makes the color seem 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 7!"
               Foreground="Red" />
</Grid>
```

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

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        Red
    </TextBlock.Foreground>
</TextBlock>
```

When you specify a *Foreground* property in XAML, a *SolidColorBrush* is created for the element behind the scenes. You can also explicitly create the *SolidColorBrush* in XAML:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <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 7!">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                Red
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

And you can go even further:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <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.

This excessive 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 7!"
               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 7!"
               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 7!"
               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 7!"
               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 7!"
               Foreground="Red" />
</Grid>
```

And with the help of property elements, that is how 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 7!"
               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 7!"
               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 originally 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 to 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 7!" />
    </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*. You can 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" />

    <!-- Not a legal place for the property element! -->
    <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>

    <!-- Not a legal place for the property element! -->
    <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.

The Resources Collection

In one sense, computer programming is all about the avoidance of repetition. (Or at least repetition by us humans. We don't mind if our machines engage in repetition. We just want it to be efficient repetition.) XAML would seem to be a particularly treacherous area for repetition because it's just markup and not a real programming language, You can easily imagine situations where a bunch of elements have the same *HorizontalAlignment* or *VerticalAlignment* or *Margin* settings, and it would certainly be convenient if there were a way to avoid a lot of repetitive markup. If you ever needed to change one of these properties, changing it just once is much better than changing it scores or hundreds of times.

Fortunately XAML has been designed by programmers who (like the rest of us) prefer not to type in the same stuff over and over again.

The most generalized solution to repetitive markup is the Silverlight *style*. But a prerequisite to styles is a more generalized sharing mechanism. This is called the *resource*, and right away we need to distinguish between the resources I'll be showing you here, and the resources encountered in Chapter 4 when embedding images into the application. Whenever there's a chance of confusion, I will refer to the resources in this chapter as XAML resources, even though they can exist in code as well.

XAML resources are always instances of a particular .NET class or structure, either an existing class or structure or a custom class. When a particular class is defined as a XAML resource, only one instance is made, and that one instance is shared among everybody using that resource.

The sharing of resources immediately disqualifies many classes from being defined as XAML resources. For example, a single instance of *TextBlock* can't be used more than once because the *TextBlock* must have a unique parent and a unique location within that parent. And that goes for any other element as well. Anything derived from *UIElement* is probably not going to show up as a resource because it can't be shared.

However, it is very common to share brushes, this is a typical way to give a particular application a certain consistent and distinctive visual appearance. Animations are also candidates for sharing. It's also possible to share text strings and numbers. Think of these as the XAML equivalents of string or numeric constants in a C# program. When you need to change one of them, you can just change the single resource rather than hunting through the XAML to change a bunch of individual occurrences.

To support the storage of resources, *FrameworkElement* defines a property named *Resources* of type *ResourceDictionary*. On any element that derives from *FrameworkElement*, you can define *Resources* as a property element. Usually this appears right under the start tag. Here's a *Resources* collection for a page class that derives from *PhoneApplicationPage*:

```
<phone:PhoneApplicationPage … >

    <phone:PhoneApplicationPage.Resources>
        …
    </phone:PhoneApplicationPage.Resources>

    …
</phone:PhoneApplicationPage>
```

The collection of resources within those *Resources* tags is sometimes called a *resource section*, and anything in that particular *PhoneApplicationPage* can then use those resources.

The *Application* class also defines a *Resources* property, and the App.xaml file that Visual Studio creates for you in a new Silverlight application already includes an empty resource section:

```
<Application … >
    <Application.Resources>
    </Application.Resources>
    …
</Application>
```

The resources defined in the *Resources* collection on a *FrameworkElement* are available only within that element and nested elements; the resources defined in the *Application* class are available throughout the application.

Sharing Brushes

Let's suppose your page contains several *TextBlock* elements, and you want to apply the same *LinearGradientBrush* to the *Foreground* of each of them. This is an ideal use of a resource.

The first step is to define a *LinearGradientBrush* in a resource section of a XAML file. If you're defining the resource in a *FrameworkElement*-derivative, the resource must be defined before is used, and it can only be used by the same element or a nested element.

```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Key="brush">
        <GradientStop Offset="0" Color="Pink" />
        <GradientStop Offset="1" Color="SkyBlue" />
    </LinearGradientBrush>
</phone:PhoneApplicationPage.Resources>
```

Notice the *x:Key* attribute. Every resource must have a key name. There are only four keywords that must be prefaced with "x" and you've already seen three of them: Besides *x:Key* they are *x:Class*, *x:Name* and *x:Null*.

Making use of that resource is possible with a couple kinds of syntax. The rather verbose way is to break out the *Foreground* property of the *TextBlock* as a property element and set it to an object of type *StaticResource* referencing the key name:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        <StaticResource ResourceKey="brush" />
    </TextBlock.Foreground>
</TextBlock>
```

There is, however, a shortcut syntax that makes use of what is called a *XAML markup extension*. A markup extension is always delimited by curly braces. Here's what the *StaticResource* markup extension looks like:

```
<TextBlock Text="Hello, Windows Phone 7!"
           Foreground="{StaticResource brush}" />
```

Notice that within the markup extension the word "brush" is not in quotation marks. Quotation marks within a markup extension are always prohibited.

Suppose you want to share a *Margin* setting. The *Margin* is of type *Thickness*, and in XAML you can specify it with 1, 2, or 4 numbers. Here's a *Thickness* resource:

```
<Thickness x:Key="margin">
    12 96
</Thickness>
```

Suppose you want to share a *FontSize* property. That's of type *double*, and you're going to need a little help. The *Double* structure, which is the basis for the *double* C# data type, is defined in the *System* namespace, but the XML namespace declarations in a typical XAML file only refer to Silverlight classes in Silverlight namespaces. What's needed is an XML namespace declaration for the *System* namespace in the root element of the page, and here it is:

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

This is the standard syntax for associating an XML namespace with a .NET namespace. First, come up with an XML namespace name that reminds you of the .NET namespace. The word "system" is good for this one; some programmers use "sys" or just "s." The hyphenated "clr-namespace" is followed by a colon and the .NET namespace name. If you're interested in referencing objects that are in the current assembly, you're done. Otherwise you need a semicolon followed by "assembly=" and the assembly, in this case the standard mscorlib.lib ("Microsoft Common Runtime Library").

Now you can have a resource of type *double*:

```
<system:Double x:Key="fontsize">
    48
</system:Double>
```

The ResourceSharing project defines all three of these resources and references them in two *TextBlock* elements. Here's the complete resource section:

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

```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Key="brush">
        <GradientStop Offset="0" Color="Pink" />
        <GradientStop Offset="1" Color="SkyBlue" />
    </LinearGradientBrush>

    <Thickness x:Key="margin">
        12 96
    </Thickness>

    <system:Double x:Key="fontsize">
        48
    </system:Double>
</phone:PhoneApplicationPage.Resources>
```

The content grid contains the two *TextBlock* elements:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Whadayasay?"
               Foreground="{StaticResource brush}"
               Margin="{StaticResource margin}"
               FontSize="{StaticResource fontsize}"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    <TextBlock Text="Fuhgedaboudit!"
               Foreground="{StaticResource brush}"
               Margin="{StaticResource margin}"
               FontSize="{StaticResource fontsize}"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom" />
</Grid>
```

The screen shot demonstrates that it works:

The *Resources* property is a dictionary, so within any resource section the key names must be unique. However, you can re-use key names in different resource collections. For example, try inserting the following markup right after the start tag of the content grid:

```
<Grid.Resources>
    <Thickness x:Key="margin">96</Thickness>
</Grid.Resources>
```

This resource will override the one defined on *MainPage*. Resources are searched going up the visual tree for a matching key name, and then the *Resources* collection in the application class is searched. For this reason, the *Resources* collection in App.xaml is an excellent place to put resources that are used throughout the app.

If you put that little piece of markup in the *Grid* named “LayoutRoot” it will also be accessible to the *TextBlock* elements because this *Grid* is an ancestor. But if you put the markup in the *StackPanel* entitled “TitlePanel,” (and changing *Grid* to *StackPanel* in the process) it will be ignored. Resources are searched going up the visual tree, and that’s another branch.

This little piece of markup will also be ignored if you put it in the content grid but *after* the two *TextBlock* elements. Now it’s not accessible because it’s lexicographically after the reference.

x:Key and *x:Name*

If you need to reference a XAML resource from code, you can simply index the *Resources* property with the resource name. For example, in the MainPage.xaml.cs code-behind file, this code will retrieve the resource named “brush” stored in the *Resources* collection of *MainPage*:

```
this.Resources["brush"]
```

You would then probably cast that object to an appropriate type, in this case either *Brush* or *LinearGradientBrush*. Because the *Resources* collection isn't built until the XAML is processed, you can't access the resource before the *InitializeComponent* call in the constructor of the code-behind file.

If you have resources defined in other *Resource* collections in the same XAML file, you can retrieve those as well. For example, if you've defined a resource named "margin" in the *Resources* collection of the content grid, you can access that resource using:

```
ContentGrid.Resources["margin"]
```

If no resource with that name is found in the *Resources* collection of an element, then the *Resources* collection of the *App* class is searched. If the resource is not found there, then the indexer returns *null*.

Due to a legacy issue involving Silverlight 1.0, you can use *x:Name* rather than using *x:Key* to identify a resource:

```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Name="brush">
    …
</phone:PhoneApplicationPage.Resources>
```

There is one big advantage to this: The name is stored as a field in the generated code file so you can reference the resource in the code-behind file just like any other field:

```
txtblk.Foreground = brush;
```

This is a much better syntax for sharing resources between XAML and code. However, if you use *x:Name* for a resource, that name must be unique in the XAML file.

An Introduction to Styles

One very common item in a *Resources* collection is a *Style*, which is basically a collection of property assignments for a particular element type. Besides a key, the *Style* also requires a *TargetType*:

```
<Style x:Key="txtblkStyle"
       TargetType="TextBlock">
       …
</Style>
```

Between the start and end tags go one or more *Setter* definitions. *Setter* has two properties: One is actually called *Property* and you set it to a property name. The other is *Value*. A few examples:

```
<Style x:Key="txtblkStyle"
       TargetType="TextBlock">
```

```
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="Margin" Value="12 96" />
    <Setter Property="FontSize" Value="48" />
</Style>
```

Suppose you also want to include a *Setter* for the *Foreground* property but it's a *LinearGradientBrush*. There are two ways to do it. If you have a previously defined resource with a key of "brush" (as in the ResourceSharing project) you can reference that:

```
<Setter Property="Foreground" Value="{StaticResource brush}" />
```

Or, you can use property-element syntax with the *Value* property to embed the brush right in the *Style* definition. That's how it's done in the *Resources* collection of the StyleSharing project:

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

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="txtblkStyle"
           TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Margin" Value="12 96" />
        <Setter Property="FontSize" Value="48" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Pink" />
                    <GradientStop Offset="1" Color="SkyBlue" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</phone:PhoneApplicationPage.Resources>
```

To apply this style to an element of type *TextBlock*, set the *Style* property (which is defined by *FrameworkElement* so every kind of element has it):

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Whadayasay?"
               Style="{StaticResource txtblkStyle}"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    <TextBlock Text="Fuhgedaboudit!"
               Style="{StaticResource txtblkStyle}"
               HorizontalAlignment="Right"
```

```
                VerticalAlignment="Bottom" />
</Grid>
```

The display looks the same as the previous program, which teaches an important lesson. Notice that values of *HorizontalAlignment* and *VerticalAlignment* are defined in the *Style*, yet these are overridden by local settings in the two *TextBlock* elements. But the *Foreground* set in the *Style* overrides the value normally inherited through the visual tree.

That means that the little chart I started earlier in this chapter can now be enhanced slightly.

Local Settings have precedence over

Style Settings, which have precedence over

Property Inheritance, which has precedence over

Default Values

Style Inheritance

Styles can enhance or modify other styles through the process of inheritance. Set the *Style* property *BasedOn* to a previously defined *Style*. Here's the *Resources* collection of the StyleInheritance project:

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

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="txtblkStyle"
           TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Margin" Value="12 96" />
        <Setter Property="FontSize" Value="48" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Pink" />
                    <GradientStop Offset="1" Color="SkyBlue" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>

    <Style x:Key="upperLeftStyle"
           TargetType="TextBlock"
           BasedOn="{StaticResource txtblkStyle}">
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="VerticalAlignment" Value="Top" />
    </Style>
```

```
    <Style x:Key="lowerRightStyle"
           TargetType="TextBlock"
           BasedOn="{StaticResource txtblkStyle}">
        <Setter Property="HorizontalAlignment" Value="Right" />
        <Setter Property="VerticalAlignment" Value="Bottom" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

The two new *Style* definitions at the end override the *HorizontalAlignment* and *VerticalAlignment* properties set in the earlier style. This allows the two *TextBlock* elements to reference these two different styles:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="Whadayasay?"
               Style="{StaticResource upperLeftStyle}" />

    <TextBlock Text="Fuhgedaboudit!"
               Style="{StaticResource lowerRightStyle}" />
</Grid>
```

Implicit styles, which were introduced into Silverlight 4, are not supported in Silverlight for Windows Phone.

Themes

Windows Phone 7 predefines many resources that you can use throughout your application with the *StaticResource* markup extension. There are predefined colors, brushes, font names, font sizes, margins, and text styles. Some of them show up in the root element of MainPage.xaml to supply the defaults for the whole page:

```
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
```

You can find all these predefined themes in the Themes section of the Windows Phone 7 documentation. You should try to use these resources particularly for foreground and background brushes so you comply with the user's wishes, and you don't inadvertently cause your text to become invisible. Some of the predefined font sizes may be different when the small-screen phone is released, and these differences might help you port your large-screen programs to the new device.

What happens if the user sets a different theme while your program is running? Well, the only way this can happen is if your program is tombstoned at the time, and when your program is reactivated, it starts up from scratch and hence references the new colors automatically.

The color theme that the user selects includes a foreground and background (either white on a black background or black on a white background) but also an accent color: magenta, purple, teal, lime, brown, pink, orange, blue (the default), red, or green. This color is available as the PhoneAccentColor resource, and a brush based on this color is available as PhoneAccentBrush.

Gradient Accents

You might want to use the user's preferred accent color in your program, but as a gradient brush. In other words, you want the same hue, but you want to get darker or lighter versions. In code, this is fairly easy by manipulating the red, green, and blue components of the color.

It's also fairly easy in XAML, as the GradientAccent project demonstrates:

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

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Grid.Background>
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <GradientStop Offset="0" Color="White" />
            <GradientStop Offset="0.5" Color="{StaticResource PhoneAccentColor}" />
            <GradientStop Offset="1" Color="Black" />
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

Here it is:

Chapter 8

Elements and Properties

Although *TextBlock* and *Image* are surely two of the most important elements supported by Silverlight, there are a couple more elements you should see before the next chapter introduces the *Panel* elements that provide the basis of Silverlight's dynamic layout system. I'll also describe some important properties you can apply to all these elements, including transforms.

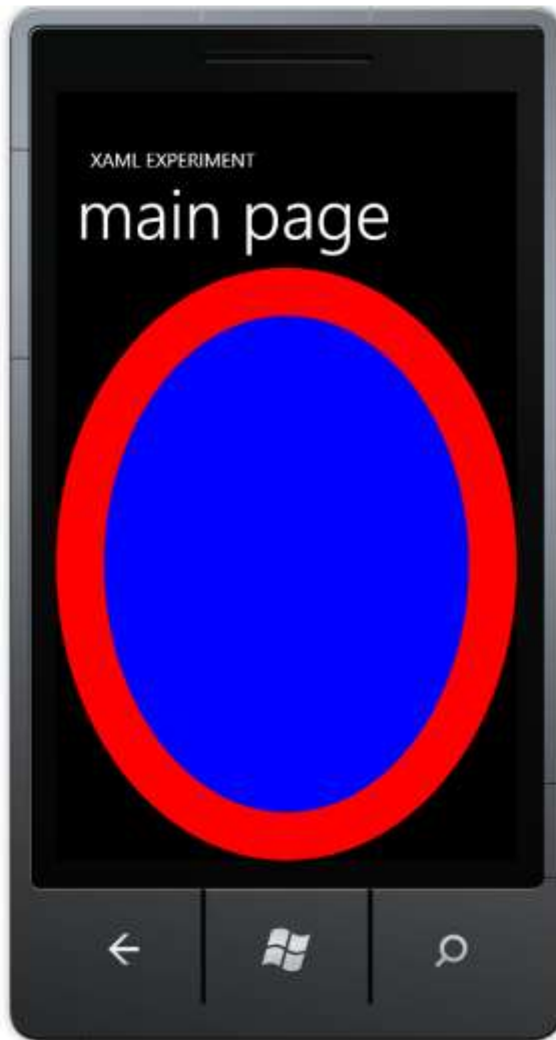
Basic Shapes

The *System.Windows.Shapes* namespace includes elements used for the display of vector graphics—the use of straight lines and curves for drawing and defining filled areas. Although the subject of vector graphics awaits us in a later chapter, two of the classes in this namespace—*Ellipse* and *Rectangle*—are a little different from the others in that you can use them without specifying any coordinate points.

Go back to the XamlExperiment program from the Chapter 7 and insert this *Ellipse* element into the content grid:

```
<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?

This *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. This is one case where explicitly setting *Width* and *Height* properties is appropriate.

Both the *Fill* property and the *Stroke* property of *Ellipse* are of type *Brush*, so you can set either or both to gradient brushes. It is very common to make the *Width* and *Height* of an *Ellipse* the same so it displays as a circle. The *Fill* can then be set to a *RadialGradientBrush* that starts at White in the center and then goes to a gradient color at the perimeter. Normally the

gradient center is the point (0.5, 0.5) relative to the ball's dimension, but you can offset that like so:

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Ellipse Width="300"
             Height="300">
        <Ellipse.Fill>
            <RadialGradientBrush Center="0.4 0.4"
                                 GradientOrigin="0.4 0.4">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="Red" />
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
</Grid>
```

The offset white spot looks like reflection from a light source, suggesting a three dimensional shape:

The *Rectangle* is similar to the *Ellipse* except that it has *RadiusX* and *RadiusY* properties for rounding the corners.

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, bitmaps, movies, panels, 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 that 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 multiplied 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!

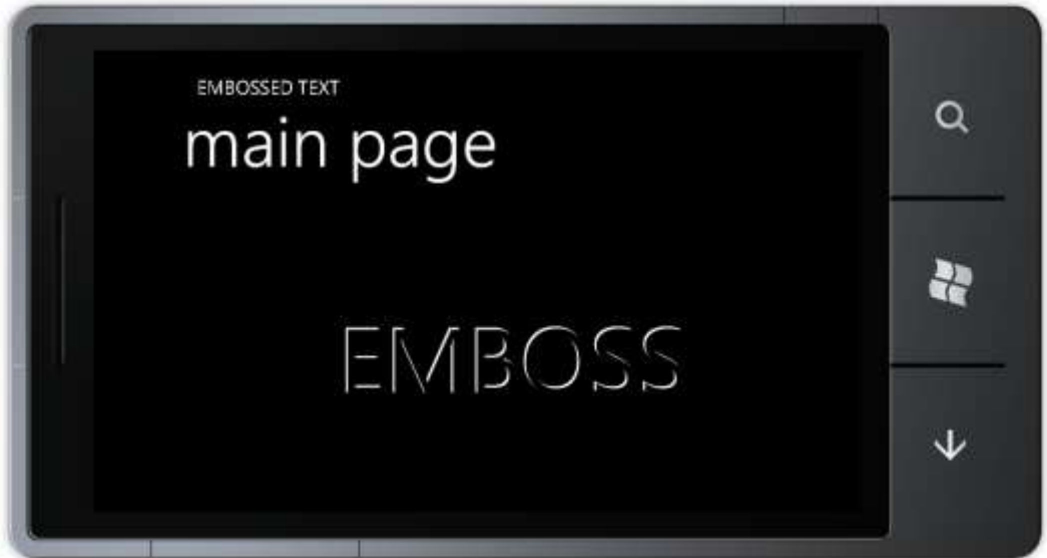
TranslateTransform is useful for making drop shadows. and effects where the text seems embossed or engraved. Simply put two *TextBlock* elements in the same location with the same text, and all the same text properties, but different *Foreground* properties. Without any transforms, the second *TextBlock* sits on top of the first *TextBlock*. On one or the other, apply a small *ScaleTransform* and the result is magic. 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="{StaticResource PhoneForegroundBrush}"
                FontSize="96"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />

    <TextBlock Text="EMBOSS"
                Foreground="{StaticResource PhoneBackgroundBrush}"
                FontSize="96"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock.RenderTransform>
            <TranslateTransform X="2" Y="2" />
```

```
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Notice I've used theme colors for the two *Foreground* properties. With the default dark theme, the *TextBlock* underneath is white, and the one on top is black like the background but shifted a little to let the white one peak 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* in the *ScaleTransform* 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 infinite") 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 then scaled, but the scaling applies to the original translation factors as well, so the text is effectively translated by 400 pixels.

It is sometimes tempting to put a *Transform* in a *Style*, like so:

```
<Setter Property="RenderTransform">
    <Setter.Value>
        <TranslateTransform />
    </Setter.Value>
</Setter>
```

You can then manipulate that transform from code, perhaps. But watch out: resources are shared. There will be only one instance of the *TranslateTransform* that is shared among all elements that use the *Style*. Hence, changing the transform for one element will also affect the others!

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 in one convenient class.

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 of the content grid. Here's the XAML:

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

```
<Grid Name="ContentGrid" Grid.Row="1" SizeChanged="OnContentGridSizeChanged">
    <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>
```

Notice the *SizeChanged* handler on the *Grid*. The code-behind file will use this to make calculation adjustments based on the size of the *Grid*, which will depend on the orientation.

Of the four *TextBlock* elements in the same *Grid*, the first is transparent and used solely by the code part of the program for measurement. The other three *TextBlock* elements are colored 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, and the constructor sets up a *DispatcherTimer*:

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

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

    public MainPage()
    {
        InitializeComponent();

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

```
        tmr.Start();
    }

    void OnContentGridSizeChanged(object sender, SizeChangedEventArgs args)
    {
        gridCenter = new Point(args.NewSize.Width / 2,
                               args.NewSize.Height / 2);

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

        scale = Math.Min(gridCenter.X, gridCenter.Y) / textSize.Width;

        UpdateClock();
    }

    void OnTimerTick(object sender, EventArgs e)
    {
        UpdateClock();
    }

    void UpdateClock()
    {
        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;
    }
}
```

HybridClock uses the *SizeChanged* handler to determine the center of the *ContentGrid*, and the size of the *TextBlock* named *referenceText*. (The latter item won't change for the duration of the program.) 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 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.

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 height of the text. 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.

Animating at the Speed of Video

The use of the *DispatcherTimer* for the HybridClock program makes sense because the positions of the clock hands need to be updated only once per second. But switching to a

sweep second hand immediately raises the question: How often should the clock hands be updated? Considering that the second hand only needs to move a few pixels per second, setting the timer for 250 milliseconds would probably be fine, and 100 milliseconds would be more than sufficient.

It's helpful to keep in mind that the video display of Windows Phone 7 devices is refreshed 30 times per second, or once every 33-1/3 milliseconds. Therefore, the use of a timer with a tick rate shorter than 33-1/3 milliseconds makes no sense whatsoever for video animations.

Ideal for animations is a timer that is synchronous with the video refresh rate, and Silverlight provides one in the very easy-to-use *CompositionTarget.Rendering* event. The event handler looks something like this:

```
void OnCompositionTargetRendering(object sender, EventArgs args)
{
    TimeSpan renderingTime = (args as RenderingEventArgs).RenderingTime;
    ...
}
```

Although the event handler must be defined with an *EventArgs* argument, the argument is actually a *RenderingEventArgs* object. If you cast the argument to a *RenderingEventArgs*, you can get a *TimeSpan* object that indicates the elapsed time since the application began running.

CompositionTarget is a static class with only one public member, which is the *Rendering* event. Install the event handler like so:

```
CompositionTarget.Rendering += OnCompositionTargetRendering;
```

Unless you're coding a very animation-laden game, you probably don't want this event handler installed for the duration of your program, so uninstall it when you're done:

```
CompositionTarget.Rendering -= OnCompositionTargetRendering;
```

The RotatingText project contains a *TextBlock* in the center of its content grid:

Project: RotatingText File: MainPage.xaml (excerpt)

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock Text="ROTATE!"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate" />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Notice the *x:Name* attribute on the *RotateTransform*. You can't use *Name* here because that's defined by *FrameworkElement*. The code-behind file starts a *CompositionTarget.Rendering* event going in its constructor:

Project: RotatingText File: MainPage.xaml.cs (except)

```
public partial class MainPage : PhoneApplicationPage
{
    TimeSpan startTime;

    public MainPage()
    {
        InitializeComponent();
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, EventArgs args)
    {
        TimeSpan renderingTime = (args as RenderingEventArgs).RenderingTime;

        if (startTime.Ticks == 0)
        {
            startTime = renderingTime;
        }
        else
        {
            TimeSpan elapsedTime = renderingTime - startTime;
            rotate.Angle = 180 * elapsedTime.TotalSeconds % 360;
        }
    }
}
```

The event handler uses the *renderingTime* to pace the animation so there's one revolution every two seconds.

For simple repetitive animations like this, the use of Silverlight's built-in animation facility is greatly preferred over *CompositionTarget.Rendering*.

Handling Manipulation Events

Transforms are also a good way to handle manipulation events. Here's a ball sitting in the middle of the content grid:

Silverlight Project: DragAndScale File: Page.xaml

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <Ellipse Width="200"
             Height="200"
```

```
             RenderTransformOrigin="0.5 0.5"
             ManipulationDelta="OnEllipseManipulationDelta">
        <Ellipse.Fill>
            <RadialGradientBrush Center="0.4 0.4"
                                 GradientOrigin="0.4 0.4">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="{StaticResource PhoneAccentColor}"
/>
            </RadialGradientBrush>
        </Ellipse.Fill>

        <Ellipse.RenderTransform>
            <CompositeTransform  />
        </Ellipse.RenderTransform>
    </Ellipse>
</Grid>
```

Notice the *CompositeTransform*. It has no name so the code will have to reference it throug the *Ellipse* element. (This is a good strategy to use if you're handling more than one element in a single event handler.)

The code-behind file just handles the *ManipulationDelta* event from the *Ellipse*:

```
void OnEllipseManipulationDelta(object sender, ManipulationDeltaEventArgs args)
{
    Ellipse ellipse = sender as Ellipse;
    CompositeTransform xform = ellipse.RenderTransform as CompositeTransform;

    if (args.DeltaManipulation.Scale.X > 0 || args.DeltaManipulation.Scale.Y > 0)
    {
        double maxScale = Math.Max(args.DeltaManipulation.Scale.X,
                                   args.DeltaManipulation.Scale.Y);
        xform.ScaleX *= maxScale;
        xform.ScaleY *= maxScale;
    }

    xform.TranslateX += args.DeltaManipulation.Translation.X;
    xform.TranslateY += args.DeltaManipulation.Translation.Y;

    args.Handled = true;
}
```

For handling anything other than taps, the *ManipulationDelta* event is crucial. This is the event that consolidates one or more fingers on an element into translation and scaling information. The *ManipulationDeltaEventArgs* has two properties named *CumulativeManipulation* and *DeltaManipulation*, both of type *ManipulationDelta*, which has two properties named *Translation* and *Scale*.

Using *DeltaManipulation* is often easier than *CumulativeManipulation*. If only one finger is manipulating the element, then only the *Translation* factors are valid, and these can just be

added to the *TranslateX* and *TranslateY* properties of the *CompositeTransform*. If two fingers are touching the screen, then the *Scale* values are non-zero, although they could be negative and they're often unequal. To keep the circle a circle, I use the maximum and multiply by the existing scaling factors of the transform. This enables "pinch" and "stretch" manipulations.

The XAML file sets the transform center to the center of the ellipse; in theory it should be based on the position and movement of the two fingers, but this is a rather more difficult thing to determine.

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 load the XamlExperiment program from the last chapter into Visual Studio, 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 7!" />
        </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 7!"
                   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 7!" />
    </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* or *Padding* 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 7!"
                   RenderTransformOrigin="0.5 0.5">
            <TextBlock.RenderTransform>
                <RotateTransform Angle="45" />
            </TextBlock.RenderTransform>
        </TextBlock>
    </Border>
</Grid>
```

Here's what you get:

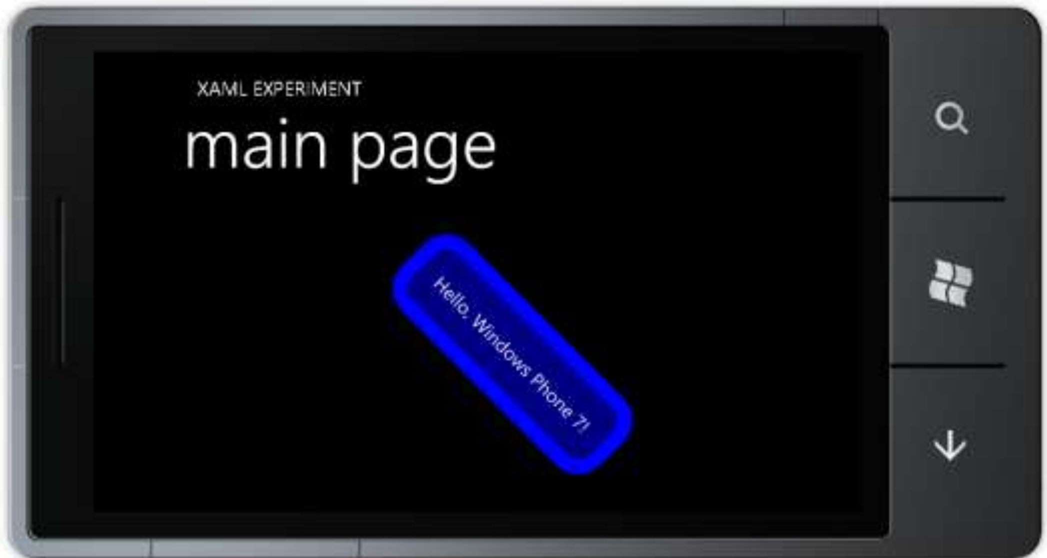
The *RenderTransform* property is called a *render* transform for a reason: It only affects rendering. It does *not* affect how the element is perceived in the layout system. The Windows Presentation Foundation has a second property named *LayoutTransform* that does affect layout. If you were coding in WPF and set the *LayoutTransform* in this case, the *Border* would expand to fit the rotated text, although it wouldn't be rotated itself. 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 7!" />
    </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.

TextBlock Properties and Inlines

Although I've been talking about the TextBlock since the early pages of this book, it's time to look at it in just a little bit more detail. The *TextBlock* element has five font-related properties: *FontFamily*, *FontSize*, *FontStretch*, *FontStyle*, and *FontWeight*.

As you saw earlier, you can set *FontStyle* to either *Normal* or *Italic*. In theory, you can set *FontStretch* to values such as *Condensed* and *Expanded* but I've never seen these work in Silverlight. Generally you'll set *FontWeight* to *Normal* or *Bold*, although there are other options like *Black*, *SemiBold*, and *Light*.

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 (
 or
) 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:

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.

The use of the *Inlines* property allows us to write a program that explores the *FontFamily* property. 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 maps seems to map to Segoe WP—a Windows Phone variant of the Segoe font that is a frequently found in Microsoft products and printed material, including this very book.

The FontFamilies program lists all the *FontFamily* values that Visual Studio's Intellisense tells us are valid:

Silverlight Project: FontFamilies   File: MainPage.xaml

```
<Grid x:Name="ContentGrid" Grid.Row="1">
    <TextBlock FontSize="24">
        <Run FontFamily="Arial">Arial</Run><LineBreak />
        <Run FontFamily="Arial Black">Arial Black</Run><LineBreak />
        <Run FontFamily="Calibri">Calibri</Run><LineBreak />
```

```
        <Run FontFamily="Comic Sans MS">Comic Sans MS</Run><LineBreak />
        <Run FontFamily="Courier New">Courier New</Run><LineBreak />
        <Run FontFamily="Georgia">Georgia</Run><LineBreak />
        <Run FontFamily="Lucida Sans Unicode">Lucida Sans Unicode</Run><LineBreak />
        <Run FontFamily="Portable User Interface">Portable User
Interface</Run><LineBreak />
        <Run FontFamily="Segoe WP">Segoe WP</Run><LineBreak />
        <Run FontFamily="Segoe WP Black">Segoe WP Black</Run><LineBreak />
        <Run FontFamily="Segoe WP Bold">Segoe WP Bold</Run><LineBreak />
        <Run FontFamily="Segoe WP Light">Segoe WP Light</Run><LineBreak />
        <Run FontFamily="Segoe WP Semibold">Segoe WP Semibold</Run><LineBreak />
        <Run FontFamily="Segoe WP SemiLight">Segoe WP SemiLight</Run><LineBreak />
        <Run FontFamily="Tahoma">Tahoma</Run><LineBreak />
        <Run FontFamily="Times New Roman">Times New Roman</Run><LineBreak />
        <Run FontFamily="Trebuchet MS">Trebuchet MS</Run><LineBreak />
        <Run FontFamily="Verdana">Verdana</Run><LineBreak />
        <Run FontFamily="Webdings">Webdings</Run> (Webdings)
    </TextBlock>
</Grid>
```

Here’s the result:

If you misspell a name that you assign to *FontFamily*, nothing bad will happen; you'll just get the default.

The predefined resources include four keys that return objects of type *FontFamily*: PhoneFontFamilyNormal, PhoneFontFamilyLight, PhoneFontFamilySemiLight, and PhoneFontFamilySemiBold. These return the corresponding Segoe WP fonts.

More on Images

As you saw in Chapter 4, a Silverlight program can display bitmaps in the JPEG and PNG formats with the *Image* element. Let's explore the *Image* element a little more.

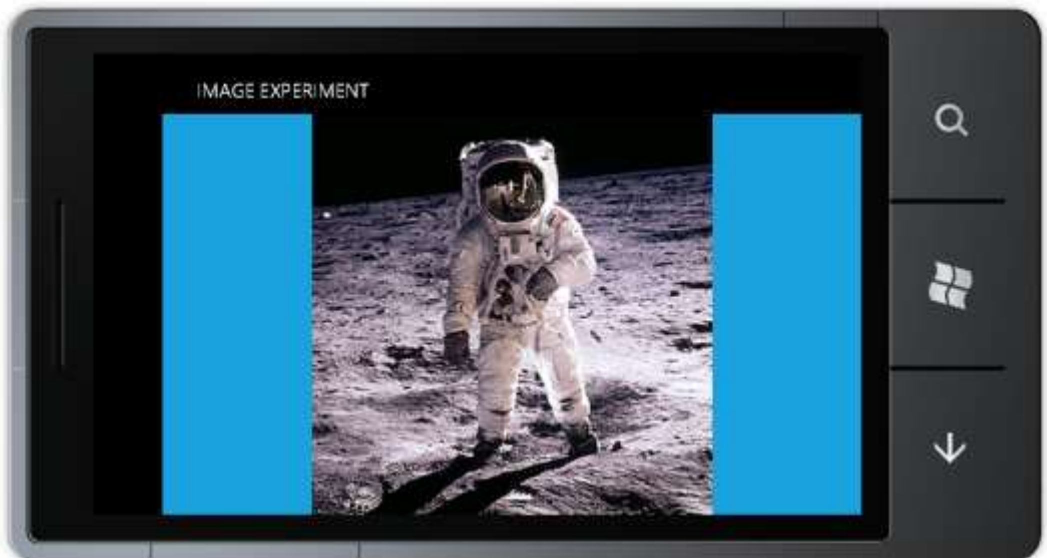
The ImageExperiment project contains a folder named Images containing a file named BuzzAldrinOnTheMoon.png, which is the famous photograph taken with a Hasselblad camera by Neil Armstrong on July 21st, 1969. The photo is 288 pixels square.

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="Images/BuzzAldrinOnTheMoon.png" />
</Grid>
```

I've also give the content grid a *Background* brush of the accent color just to make the photo stand out a little better. Here's how it appears in landscape mode:



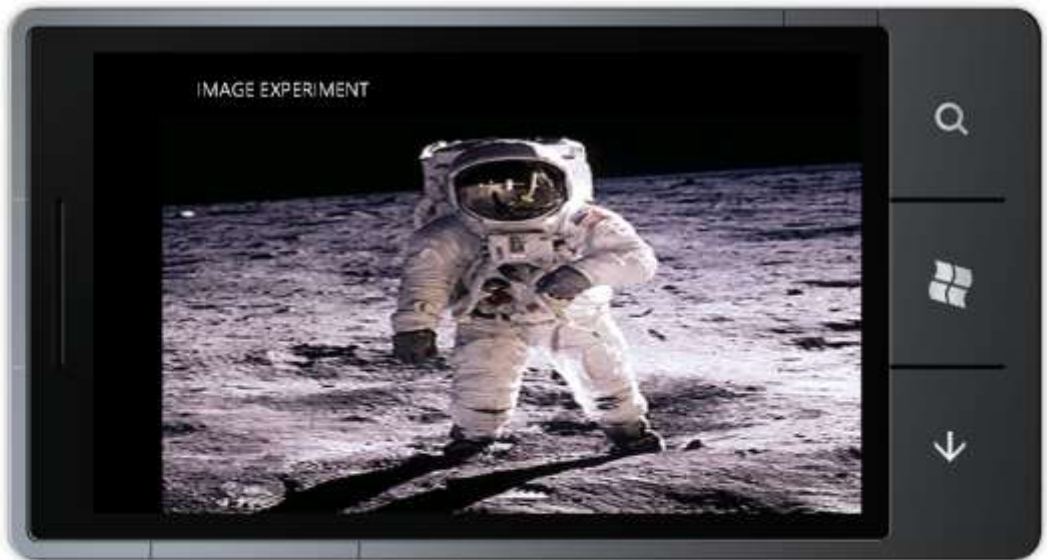
By default, the bitmap expands to the size of its container (the content grid in this case) while maintaining the correct aspect ratio. Depending on the dimensions and aspect ratio of the container, the image is centered either horizontally or vertically. You can move it to one side or the other with 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="Images/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.

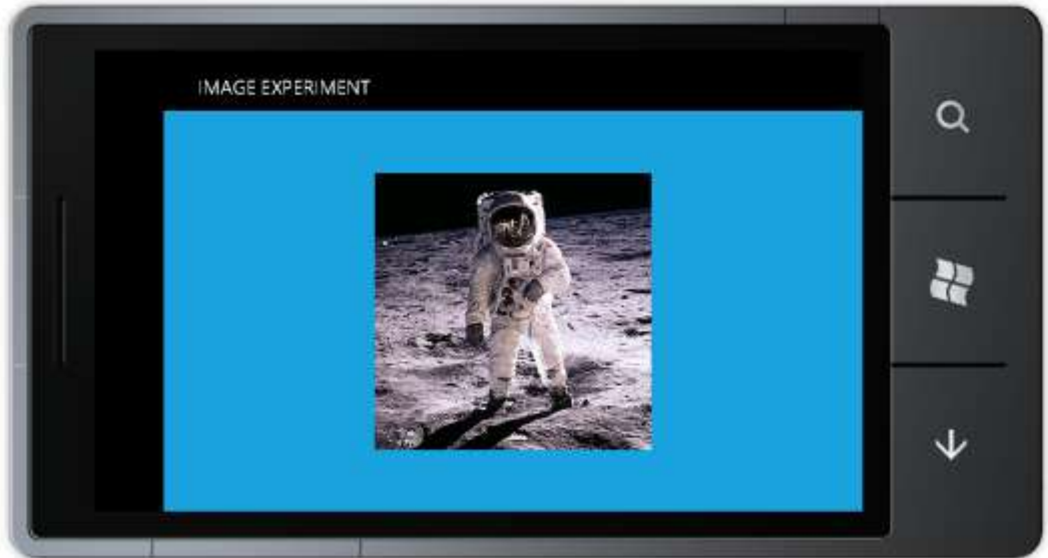


A compromise is *UniformToFill*:

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.

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

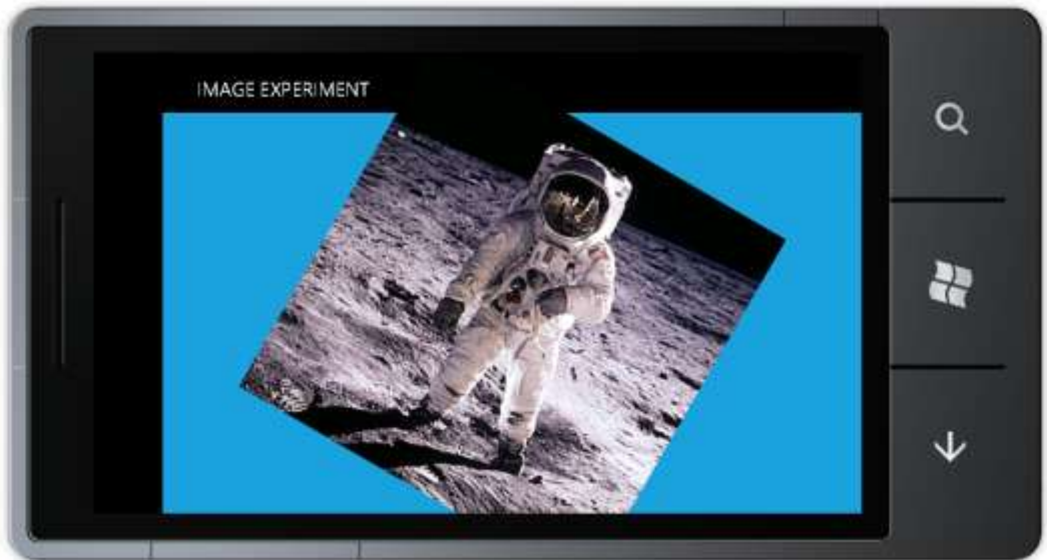


If you want to display the image in a particular size at the correct aspect ratio, you can set either an explicit *Width* or *Height* property. If you want to stretch non-uniformly to a particular dimension, specify both *Width* and *Height* and set *Stretch* to *Fill*.

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

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

Here it is:



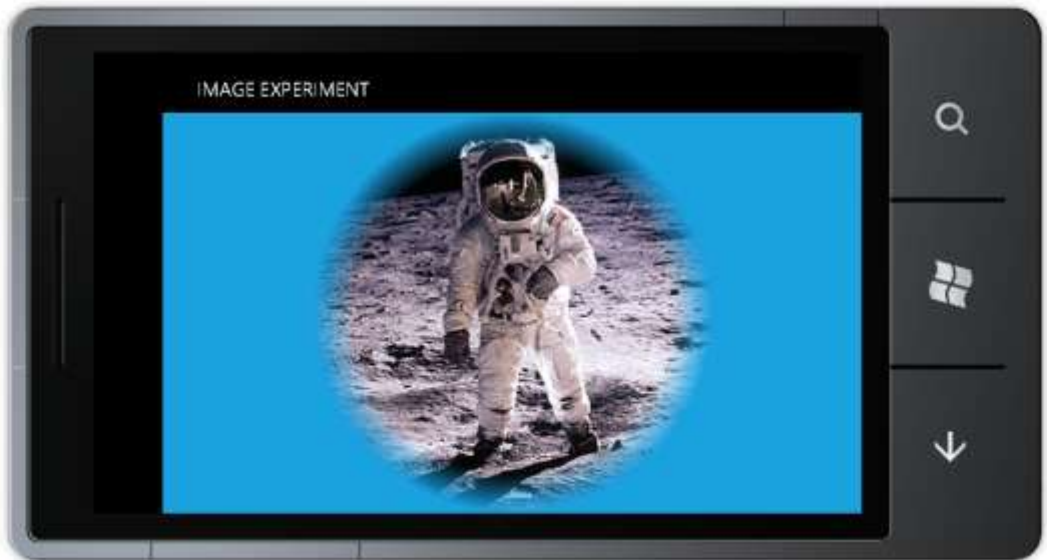
Modes of Opacity

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="Images/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:



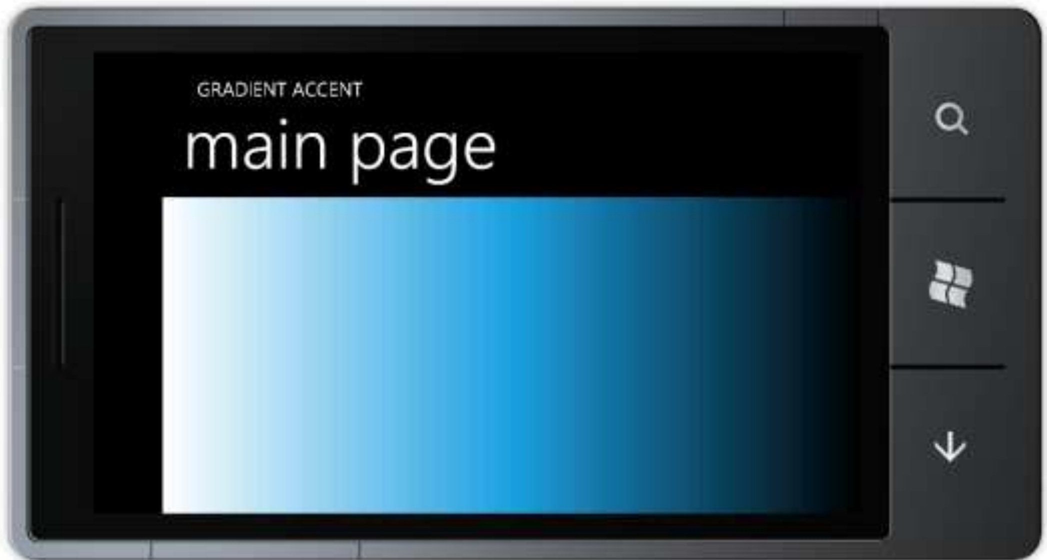
Here's a popular technique that uses two identical elements but one of them gets both a *ScaleTransform* to flip it upside down, and an *OpacityMask* to make it fade out:

```
<Image Source="Images/BuzzAldrinOnTheMoon.png"
       Stretch="None"
       VerticalAlignment="Top" />
<Image Source="Images/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>
```

The two *Image* elements are 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 (0.5, 1), which is the bottom of the element. This causes the scaling to flip the image 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.

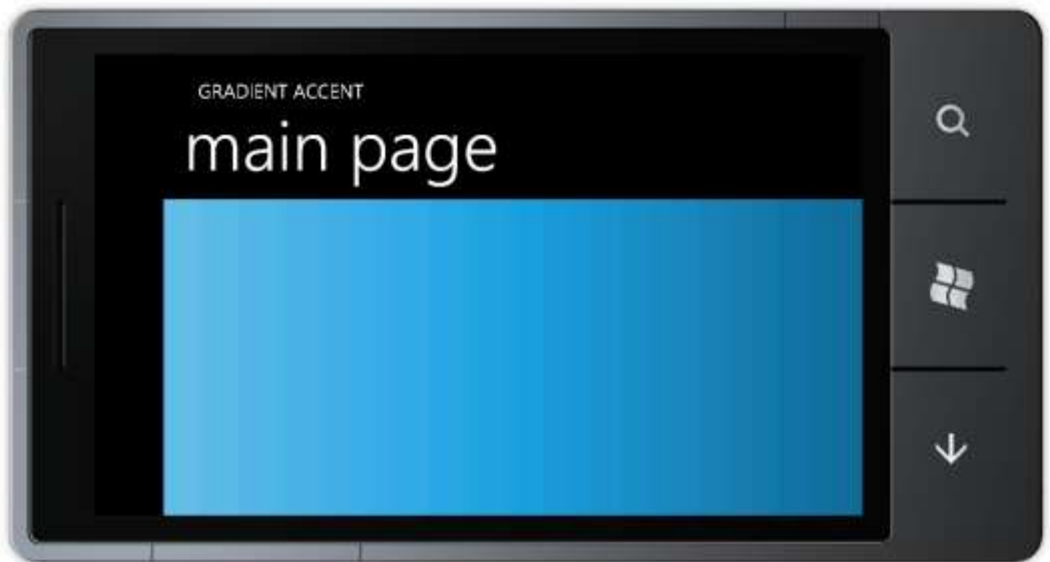
But indiscriminate use of *OpacityMask*—particularly in combination with complex animations—is discouraged because it sometimes tends to cripple performance. The general rule is: Only use it if the effect is really, really cool.



You can get a more subtle affect by changing the gradient offsets. These can actually be set outside the range of 0 to 1, perhaps like this:

```
<LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
    <GradientStop Offset="-1" Color="White" />
    <GradientStop Offset="0.5" Color="{StaticResource PhoneAccentColor}" />
    <GradientStop Offset="2" Color="Black" />
</LinearGradientBrush>
```

Now the gradient goes from White at an offset of –1 to the accent color at 0.5 to Black at 2. But you're only seeing the section of the gradient between 0 and 1, so the White and Black extremes are not here:



It’s just another little suggestion that XAML can be more powerful than it might at first seem.

Part III

XNA

Chapter 20
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 fixed value during the *LoadContent* method. This code puts the text in the center of the screen:

```
Vector2 textSize = kootenay14.MeasureString(text);
Viewport viewport = this.GraphicsDevice.Viewport;
textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                           (viewport.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 at first 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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // pixels per second
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont kootenay14;
    Viewport viewport;
    Vector2 textSize;
    Vector2 textPosition;
    bool isGoingUp = false;
        …
    }
}
```

Nothing should be too startling here. I've defined both the SPEED and TEXT as constants. The SPEED is set at 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);
    viewport = this.GraphicsDevice.Viewport;
    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    textSize = kootenay14.MeasureString(TEXT);
    textPosition = new Vector2(viewport.X + (viewport.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.TotalSeconds;
        if (textPosition.Y + textSize.Y > viewport.Height)
```

```
        {
            float excess = textPosition.Y + textSize.Y - viewport.Height;
            textPosition.Y -= 2 * excess;
            isGoingUp = true;
        }
    }
    else
    {
        textPosition.Y -= SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;

        if (textPosition.Y < 0)
        {
            float excess = - 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 second) and the total seconds that have elapsed since the last *Update* call, which will actually be 1/30th second.

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)

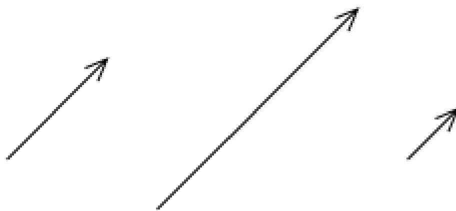
```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    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.

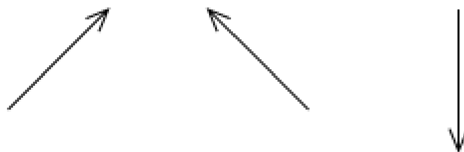
What's missing from the NaiveTextMovement program is any concept of direction that would allow escaping from horizontal or vertical movement. What we need are vectors.

A Brief Review of Vectors

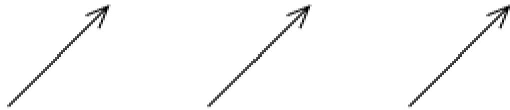
A vector is a mathematical entity that encapsulates both a direction and a magnitude. Very often a vector is symbolized by a line with an arrow. These three vectors have the same direction but different magnitudes:



These three vectors have the same magnitude but different directions:

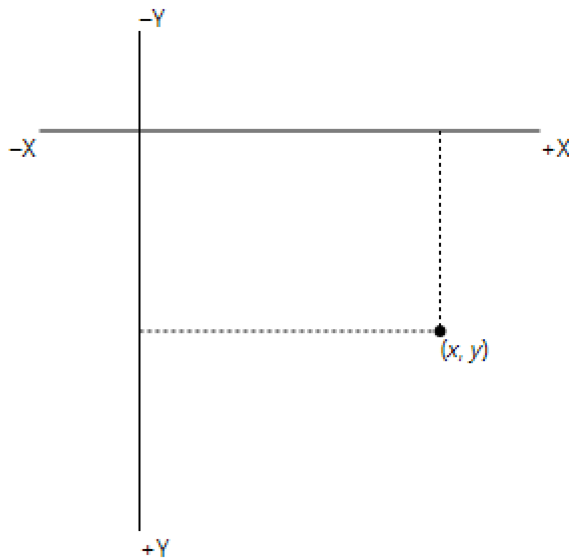


These three vectors have the same magnitude and the same direction, and hence are considered to be identical:



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.

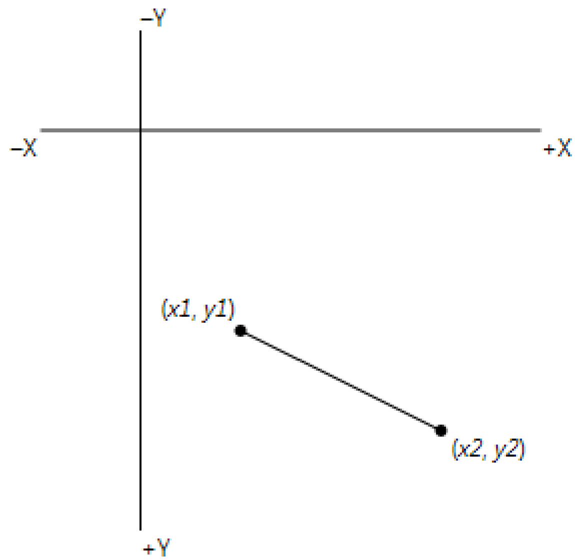
A point has no magnitude and no dimension. A point is *just* location. In two-dimensional space, a point is represented by a number pair (x, y) to represent a horizontal distance and a vertical distance from an origin (0, 0):



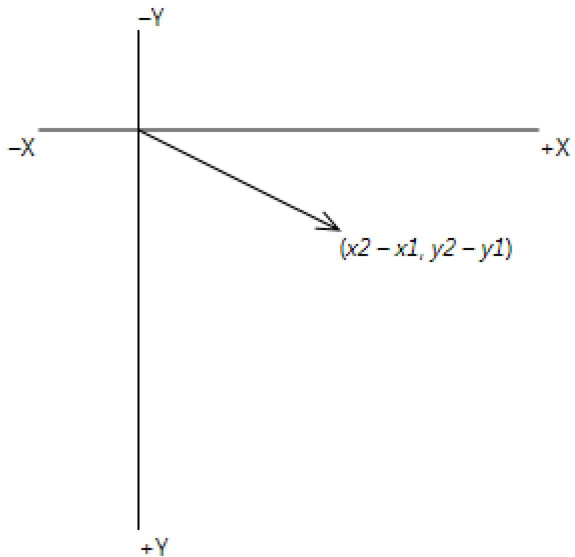
The figure shows increasing values of Y going down for consistency with the two-dimensional coordinate system in XNA. (XNA 3D is different.)

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. (There is also a *Point* structure in XNA but the *X* and *Y* fields are integers.)

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:

```
float angle = (float)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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // pixels per second
    const string TEXT = "Hello, Windows Phone 7!";

    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);
    Viewport viewport = this.GraphicsDevice.Viewport;

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

    Vector2 position1 = new Vector2(viewport.Width - textSize.X, 0);
    Vector2 position2 = new Vector2(0, viewport.Height - 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.TotalSeconds;
    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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // pixels per second
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont kootenay14;
    Vector2 position1;
    Vector2 pathVector;
    Vector2 textPosition;
    float lapSpeed;                     // laps per second
    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);
    Viewport viewport = this.GraphicsDevice.Viewport;

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);
    position1 = new Vector2(viewport.Width - textSize.X, 0);
    Vector2 position2 = new Vector2(0, viewport.Height - textSize.Y);
    pathVector = position2 - position1;
```

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

}
```

In the calculation of *lapSpeed*, the numerator is in units of pixels-per-second. 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-second by pixels-per-lap give you a speed in units of laps-per-second.

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.TotalSeconds;
    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 seconds. 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.TotalSeconds) % 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:

SpriteEffects.None

SpriteEffects.FlipHorizontally

SpriteEffects.FlipVertically

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((viewport.Width - textSize.X) / 2, (viewport.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(viewport.Width / 2, viewport.Height / 2);
origin = new Vector2(textSize.X / 2, textSize.Y / 2);
```

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(viedwport.Width, viewport.Height);
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. An earlier version of the program rotated the text to align it with the longest dimension of the screen; this version assumes that the screen is in landscape mode. As with the other programs, it includes a font. Here are the fields:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.5f;           // laps per second
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont kootenay14;
    Vector2 textPosition;
    Vector2 origin;
    Vector2 maxScale;
    Vector2 scale;
    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, the *origin* field to the center of the text, and *maxScale* to the maximum scaling factor necessary to fill the screen with the text. 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);
    Viewport viewport = this.GraphicsDevice.Viewport;

    kootenay14 = this.Content.Load<SpriteFont>("Kootenay14");
    Vector2 textSize = kootenay14.MeasureString(TEXT);
    textPosition = new Vector2(viewport.Width / 2, viewport.Height / 2);
    origin = new Vector2(textSize.X / 2, textSize.Y / 2);
    maxScale = new Vector2(viewport.Width / textSize.X, viewport.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.TotalSeconds) % 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

Let's conclude this chapter with two programs that rotate text.

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 second 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:

$$x(t) = \frac{1}{2}at^2$$

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 second and acceleration in units of revolutions per second squared:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float ACCELERATION = 1;       // revs per second squared
    const float MAXSPEED = 30;          // revs per second
    const string TEXT = "Hello, Windows Phone 7!";

    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. As the spinning text reaches that speed, it should appear to stop. 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);
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2(viewport.Width / 2, viewport.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(viewport.Width - textSize.X, viewport.Height -
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.TotalSeconds;
        speed = Math.Min(MAXSPEED, speed);
        angle += MathHelper.TwoPi * speed *
(float)gameTime.ElapsedGameTime.TotalSeconds;
        angle %= MathHelper.TwoPi;
    }
    else
    {
        if (speed == 0)
            SuppressDraw();

        speed = 0;
    }
    strBuilder.Remove(0, strBuilder.Length);
    strBuilder.AppendFormat(" {0:F1} revolutions/second", 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 the fields should look mostly familiar at this point:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.1f;          // laps per second
    const string TEXT = "Hello, Windows Phone 7!";

    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.TotalSeconds) % 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 21

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 right in Visual Studio or 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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    static readonly TimeSpan ANIMATION_DURATION = TimeSpan.FromSeconds(5);
    const int CHAR_SPACING = 5;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Viewport viewport;
    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 *LoadContent* 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);
    viewport = this.GraphicsDevice.Viewport;

    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;
```

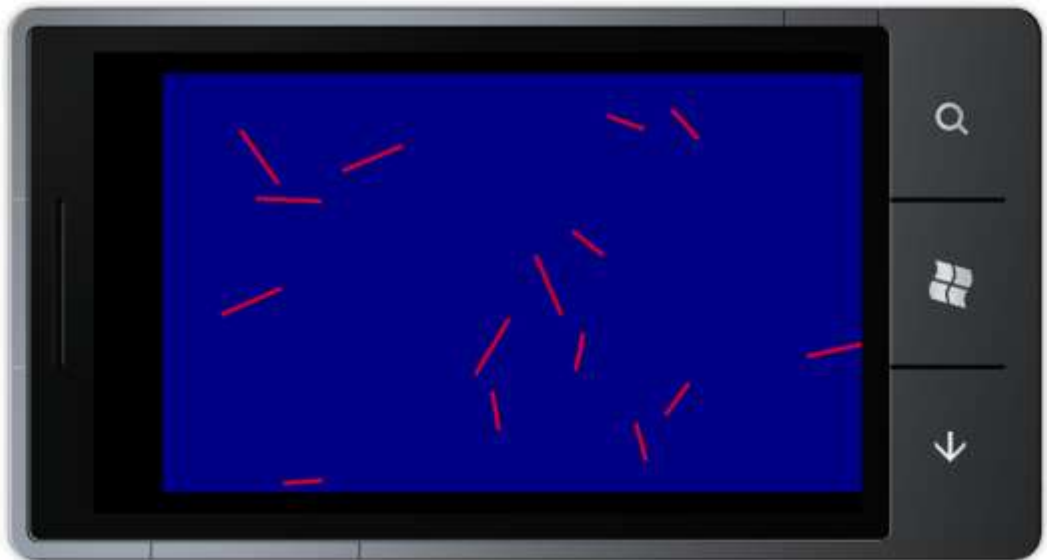
```
        if (atLeastOneTouchPointPressed)
        {
            foreach (SpriteInfo spriteInfo in spriteInfos)
            {
                float r1 = (float)rand.NextDouble() - 0.5f;
                float r2 = (float)rand.NextDouble() - 0.5f;
                float r3 = (float)rand.NextDouble();

                spriteInfo.PositionOffset = new Vector2(r1 * viewport.Width,
                                                        r2 * viewport.Height);
                spriteInfo.MaximumRotation = 2 * (float)Math.PI * r3;
            }
            animationStartTime = gameTime.TotalGameTime;
            isAnimationGoing = true;
        }
    }
    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);
}
```

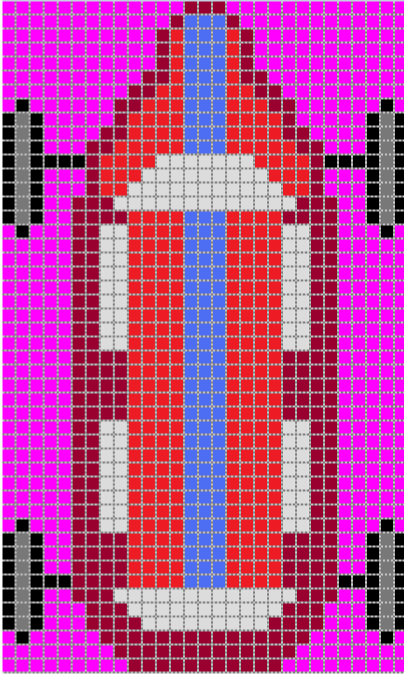
The *Draw* call also uses *SpriteInfo.InterpolationFactor* to interpolate between blue and red for coloring the bars. Notice that the *Color* structure also has a *Lerp* method. The text is normally blue but changes to red as the pieces fly apart.



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*. This configuration would allow *SpriteBatch* to have fewer public properties.

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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 100;            // 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;
    Viewport viewport = this.GraphicsDevice.Viewport;
    turnPoints[0] = new Vector2(margin, margin);
    turnPoints[1] = new Vector2(viewport.Width - margin, margin);
    turnPoints[2] = new Vector2(viewport.Width - margin, viewport.Height - margin);
    turnPoints[3] = new Vector2(margin, viewport.Height - 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.TotalSeconds;

    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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.25f;          // 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;
    Viewport viewport = this.GraphicsDevice.Viewport;

    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(viewport.Width - car.Width, car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, viewport.Height - car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(viewport.Width - car.Width, viewport.Height - car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, 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.TotalSeconds) % 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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.25f;          // laps per second
    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);
    Viewport viewport = this.GraphicsDevice.Viewport;
    ellipseCenter = viewport.Bounds.Center;
    ellipseRadiusX = viewport.Width / 2 - car.Width;
    ellipseRadiusY = viewport.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.TotalSeconds) % 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);
}
```

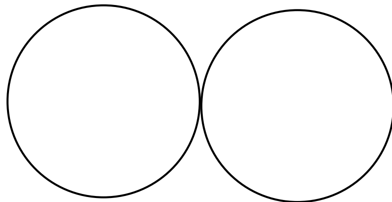
A Generalized Curve Solution

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:

Chapter 22

Touch and Play

Often when learning a new programming environment, a collection of techniques are acquired that don't necessary add up to the skills required to create a complete program. This chapter is intended to compensate for that problem by presenting two rather archetypal programs for the phone called PhingerPaint and PhreeCell. The first is a simple drawing program; the second is a version of the classic solitaire game.

Both programs use three powerful tools:

- The dynamic manipulation of *Texture2D* objects.
- The use of components in architecting a game.
- The processing of touch input.

In addition, the PhingerPaint game will demonstrate the use of a dynamic link library (DLL). These programs — while certainly not of commercial quality — will at least give you a little better sense of what a "real program" looks like.

Dynamic *Texture2D* Objects

In the previous chapter you saw how to create bitmaps in programs like Paint and load them into your game as *Texture2D* objects. It is also possible to create *Texture2D* objects directly in your program and to dynamically manipulate the pixel bits. The first step is to use one of the *Texture2D* constructors:

```
Texture2D texture = new Texture2D(this.GraphicsDevice, width, height);
```

The *width* and *height* arguments indicate the size of the *Texture2D* in pixels; this size cannot be changed after the *Texture2D* is created. The total number of pixels in the bitmap is easily calculated as *width* * *height*.

Each pixel in the bitmap is encoded with a particular color; how the bits of each pixel correspond to a particular color is often referred to as the bitmap's color format or, in XNA, with a member of the *SurfaceFormat* enumeration. A *Texture2D* created with the simple constructor shown above will have a *Format* property set to *SurfaceFormat.Color*, which means that every pixel consists of 4 bytes (or 32 bits) of data, one byte each for the red, green, and blue values (in this order), and another byte for the alpha channel, which is the opacity of that pixel.

It is also possible (and very convenient) to treat each pixel as a 32-bit unsigned integer, which in C# is a *uint*. In accordance with the "little-endian" for of byte ordering, where the least

significant byte of a multibyte value comes first, the colors appear in the 8-digit hexadecimal value of this *uint* like so:

AABBGGRR

If you have a *Texture2D* that you either loaded as content or created as shown above, and it has a *Format* property of *SurfaceFormat.Color*, you can obtain all the pixel bits of the bitmap by first creating an array of type *uint*:

```
uint[] pixels = new uint[width * height];
```

You then transfer all the pixels of the *Texture2D* into the array like so:

```
texture.GetData<uint>(pixels);
```

GetData is a generic method and you simply need to indicate the data type of the array. Overloads of *GetData* allow you to get pixels corresponding to a rectangular subset of the bitmap, or starting at an offset into the *pixels* array. You can also go the other way to transfer the data in the *pixels* array back into the bitmap:

```
texture.SetData<uint>(pixels);
```

The pixels in the *pixels* array are arranged by row beginning with the topmost row. The pixels in each row are arranged left by right. For a particular row *y* and column *x* in the bitmap, you can index the *pixels* array using a single formula:

```
pixels[y * width + x]
```

One exceptionally convenient property of the *Color* structure is *PackedValue*. This converts a *Color* object into a *uint* of the precise format required for this array, for example:

```
pixels[y * width + x] = Color.Fuchsia.PackedValue;
```

Let's look at a simple example. Suppose you want a background to your game that consists of a gradient from blue at the left to red at the right. The GradientBackground project demonstrates this technique. Here are the fields:

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

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Rectangle viewportBounds;
    Texture2D background;
    …
}
```

All the work is done in the *Initialize* method. The method creates a bitmap based on the *Viewport* size (but using the *Bounds* property which has convenient integer dimensions), and fills it with data. The interpolation for the gradient is accomplished by the *Color.Lerp* method based on the *x* value:

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

```
protected override void Initialize()
{
    viewportBounds = this.GraphicsDevice.Viewport.Bounds;
    background = new Texture2D(this.GraphicsDevice, viewportBounds.Width,
                                                    viewportBounds.Height);

    uint[] pixels = new uint[background.Width * background.Height];

    for (int x = 0; x < background.Width; x++)
    {
        uint clr = Color.Lerp(Color.Blue, Color.Red,
                              (float)x / background.Width).PackedValue;

        for (int y = 0; y < background.Height; y++)
            pixels[y * background.Width + x] = clr;
    }
    background.SetData<uint>(pixels);

    base.Initialize();
}
```

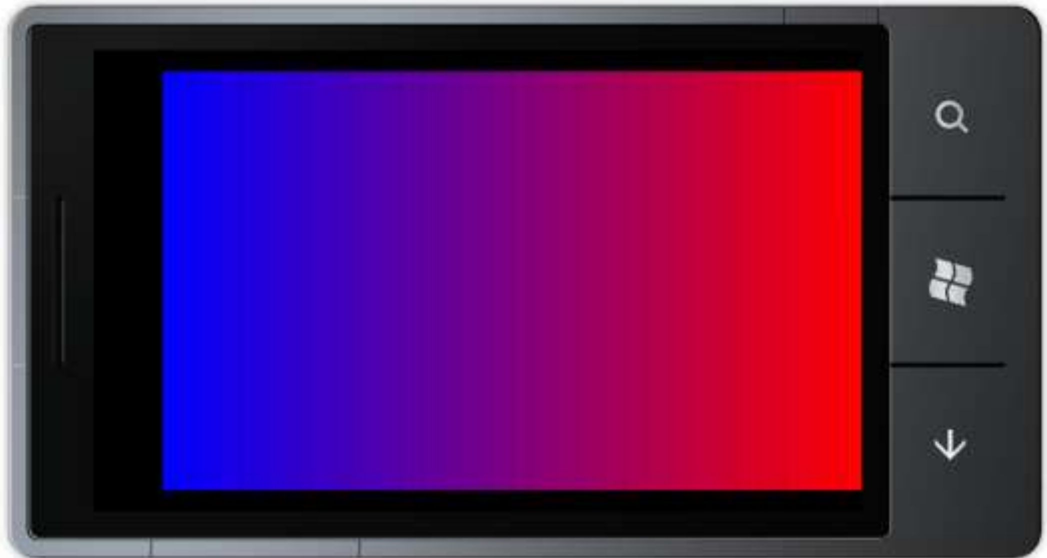
Don’t forget to call *SetData* after filling the *pixels* array with data! It’s pleasant to assume that there’s some kind of behind-the-scenes binding between the *Texture2D* and the array, but there’s really no such thing.

The *Draw* method simply draws the *Texture2D* like normal:

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

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(background, viewportBounds, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Here’s the gradient:



For this particular example, where the *Texture2D* is the same from top to bottom, it's not necessary to have quite so many rows. In fact, you can create the *background* object with just one row:

```
background = new Texture2D(this.GraphicsDevice, viewportBounds.Width, 1);
```

Because the other code in *Initialize* was based on the *background.Width* and *background.Height* properties, nothing else needs to be changed (although the loops could certainly be simplified. In the *Draw* method, the bitmap is then stretched to fill the *Rectangle*:

```
spriteBatch.Draw(background, viewportBounds, Color.White);
```

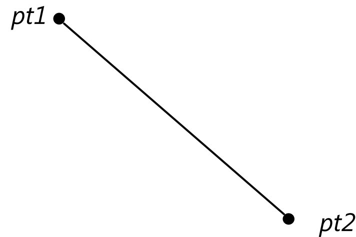
The Geometry of PhingerPaint

The PhingerPaint program allows you to draw on the screen with your phingers (or rather, *fingers*) to create masterpieces like this:

The squares at the bottom are instances of a *ColorBlock* class, which derives from *DrawableGameComponent*. These allow you to select the paint color. A bitmap "canvas" covers the rest of the *Viewport*. The program examines touch input, translates that into "brushstrokes" of sorts, sets pixel bits corresponding to each stroke, updates the bitmap canvas with *SetData*, and draws that canvas onto the display.

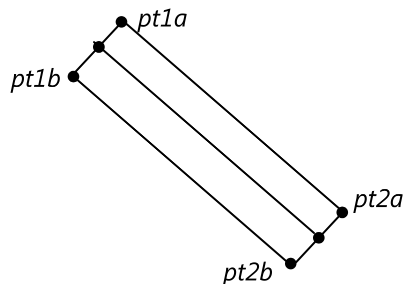
Perhaps the biggest challenge of this program is translating touch data into geometric objects that can be drawn as individual pixels. As you'll recall, touch input comes into an XNA program in the form of *TouchLocation* objects where the *State* property is a member of the *TouchLocationState* enumeration: *Pressed*, *Moved*, *Released*.

Interestingly enough, the PhingerPaint program doesn't require tracking particular fingers from touch down to touch up. It only needs look at *Moved* events. For each of these, the *Position* property indicates the current position of the finger. Calling the *TryGetPreviousLocation* method of *TouchLocation* provides access to another *TouchLocation* object containing the previous position of the finger. For purposes of analysis and illustrations, let's call these two positions *pt1* and *pt2*. The PhingerPaint program should respond by drawing a line between these two points:



However, this geometric line has zero width. You really need to draw a thick line. In theory, the thickness should correspond to the pressure that the finger is touching the screen, or the area that the finger makes in contact with the screen. In XNA 3.1, this was available in the *Pressure* property of *TouchLocation* — a float ranging in value from 0 to 1 — and it worked on the Zune HD, but it's been removed in XNA 4.0 and Windows Phone 7.

For this reason, the program should be enhanced so the user selection can select a desired brush widths. This has not been done yet. But let's assume the desired line thickness is 2*R* pixels. (*R* stands for *radius*, and you'll understand why I'm thinking of it in those terms shortly.) You really want to draw a rectangle, where the *pt1* and *pt2* are extended on each side by *R* pixels:



How are these corner points calculated? Well, it's really rather easy using vectors. Let's calculate the vector from *pt1* to *pt2* and normalize it:

```
Vector2 vector = pt2 - pt1;
vector.Normalize();
```

This vector must be rotated in increments of 90 degrees, and that's a snap. To rotate *vector* by 90 degrees clockwise, switch the X and Y coordinates while negating the Y coordinate:

```
Vector2 vect90 = new Vector2(-vector.Y, vector.X)
```

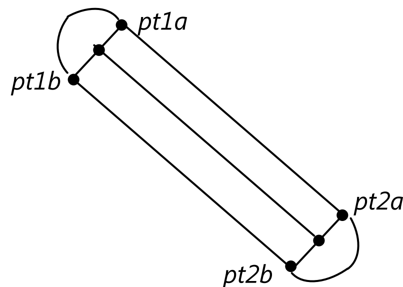
A vector rotated –90 degrees from *vector* is the negation of *vect90*.

If *vector* points from *pt1* to *pt2*, then the vector from *pt1* to *pt1a* (for example) is that vector rotated –90 degrees with a length of *R*. Then add that vector to *pt1* to get *pt1a*.

```
Vector2 pt1a = pt1 - R * vect90;
```

In a similar manner, you can also calculate *pt1b*, *pt2a*, and *pt2b*.

But you'll find that the rectangle is not sufficient. As you move your finger across the display, the program will be drawing multiple rectangles but they won't connect correctly. They will meet at angles, and slivers will appear between them. You really need to draw rounded caps on these rectangles:



These are semi-circles of radius R centered on *pt1* and *pt2*.

At this point, we have derived an overall outline of the shape to draw for two successive touch points: A line from *pt1a* to *pt2a*, a semi-circle from *pt2a* to *pt2b*, another line from *pt2b* to *pt1b*, and another semi-circle from *pt1b* to *pt1a*. The goal is to find all pixels (x, y) in the interior of this outline.

When drawing vector outlines, parametric equations are ideal. When filling areas, it's best to go back to the standard equations that we learned in high school. You probably remember the equations for a line in slope-intercept form:

$$y = mx + b$$

where *m* is the slope of the line ("rise over run") and *b* is the value of *y* where the line intercepts the X axis.

In computer graphics, however, traditionally areas are filled based on horizontal scan lines, also known as raster lines. (The terms come from television displays.) The equations represent *x* as a function of *y*:

$$x = ay + b$$

For a line from *pt1* to *pt2*,

$$a = \frac{pt2.X - pt1.X}{pt2.Y - pt1.Y}$$

$$b = pt1.X - a \cdot pt1.Y$$

For any *y*, there is a point on the line that connects *pt1* and *pt2* if *y* is between *pt1.Y* and *pt2.Y*. The *x* value can then be calculated from the equations of the line.

Look at the previous diagram and imagine a horizontal scan line that crosses the two lines from *pt1a* to *pt2a*, and from *pt1b* to *pt2b*. For any *y*, we can calculate *xa* on the line from *pt1a* to *pt2a*, and *xb* on the line from *pt1b* to *pt2b*. For that scan line, the pixels that must be colored are those between (*xa*, *y*) and (*xb*, *y*). This can be repeated for all *y*.

This process gets a little messier for the rounded caps but not much messier. A circle of radius *R* centered on the origin consists of all points (*x*, *y*) that satisfy the equation:

$$x^2 + y^2 = R^2$$

For a circle centered on (*xc*, *yc*), the equation is:

$$(x - xc)^2 + (y - yc)^2 = R^2$$

Or for any *y*:

$$x = xc \pm \sqrt{R^2 - (y - yc)^2}$$

If the expression in the square root is negative, then *y* is outside the circle entirely. Otherwise, there are (in general) two values of *x* for each *y*. The only exception is when the square root is zero, that is, when *y* is exactly *R* units from *yc*, which are the top and bottom points of the circle.

We're dealing with a semicircle so it's a little more complex, but not much. Consider the semicircle at the top of the diagram. The center is *pt1*, and the semicircle goes from *pt1b* to *pt1a*. The line from *pt1* to *pt1b* forms an angle *angle1* that can be calculated with *Math.Atan2*. Similarly for the line from *pt1* to *pt1a* there is an *angle2*. If the point (*x*, *y*) is on the circle as calculated above, it too forms an *angle* from the center *pt1*. If that angle is between *angle1* and *angle2*, then the point is on the semicircle. (This determination of "between" gets just a little messier because angles returned from *Math.Atan2* wrap around from π to –π.

Now for any *y* we can examine both the two lines and the two semicircles and determine all points (*x*, *y*) that are on these four figures. At most, there will be only two such points — one where the scan line enters the interior and the other where it exits. For that scan line, all pixels between those two points can be filled.

Let's start writing code. In the PhingerPaint solution, I created a new project named GeometryHelper of type Windows Game Library. This project creates a dynamic link library that will help with some of the mathematics. Because I intend to frequently instantiate the objects from GeometryHelper during the *Update* method, I made them all structures rather than classes. The project begins with a little interface:

XNA Project: GeometryHelper   File: IGeometrySegment.cs (complete)

```
using System.Collections.Generic;

namespace GeometryHelper
{
    public interface IGeometrySegment
    {
        void GetAllX(float y, IList<float> xCollection);
    }
}
```

All the structures will implement this interface. For any *y* value the method returns a collection of *x* values. In actual practice, with the structures in this library, often this collection will be returned empty. Sometimes it will contain one value, and sometimes two.

Here's the *LineSegment* structure:

XNA Project: GeometryHelper   File: LineSegment.cs (complete)

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace GeometryHelper
{
    public struct LineSegment : IGeometrySegment
    {
        readonly float a, b;         // as in x = ay + b

        public LineSegment(Vector2 point1, Vector2 point2) : this()
        {
            Point1 = point1;
            Point2 = point2;

            a = (Point2.X - Point1.X) / (Point2.Y - Point1.Y);
            b = Point1.X - a * Point1.Y;
        }
```

```
        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            if ((Point2.Y > Point1.Y && y >= Point1.Y && y < Point2.Y) ||
                (Point2.Y < Point1.Y && y <= Point1.Y && y > Point2.Y))
            {
                xCollection.Add(a * y + b);
            }
        }
    }
}
```

Notice that the *if* statement in *GetAllX* checks that *y* is between *Point1.Y* and *Point2.Y*; it allows *y* values that equal *Point1.Y* but not those that equal *Point1.Y*. In other words, it defines the line to be all points from *Point1* (inclusive) up to but not including *Point2*. This caution about what points are included and excluded comes into play when multiple lines and arcs are connected; it helps avoid the possibility of having duplicate *x* values in the collection.

Also notice that no special consideration is given to horizontal lines, that is, lines where *Point1.Y* equals *Point2.Y* and where *a* equals infinity. If that is the case, then the *if* statement in the method is never satisfied. A scan line never crosses a horizontal boundary line.

The next structure is similar but for a generalized arc on the circumference of a circle:

XNA Project: GeometryHelper   File: ArcSegment.cs (complete)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace GeometryHelper
{
    public struct ArcSegment : IGeometrySegment
    {
        readonly double angle1, angle2;

        public ArcSegment(Vector2 center, float radius, Vector2 point1, Vector2
point2) :
            this()
        {
            Center = center;
            Radius = radius;
            Point1 = point1;
            Point2 = point2;
            angle1 = Math.Atan2(point1.Y - center.Y, point1.X - center.X);
            angle2 = Math.Atan2(point2.Y - center.Y, point2.X - center.X);
        }
```

```
        public Vector2 Center { private set; get; }
        public float Radius { private set; get; }
        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            double sqrtArg = Radius * Radius - Math.Pow(y - Center.Y, 2);

            if (sqrtArg >= 0)
            {
                double sqrt = Math.Sqrt(sqrtArg);
                TryY(y, Center.X + sqrt, xCollection);
                TryY(y, Center.X - sqrt, xCollection);
            }
        }

        public void TryY(double y, double x, IList<float> xCollection)
        {
            double angle = Math.Atan2(y - Center.Y, x - Center.X);

            if ((angle1 < angle2 && (angle1 <= angle && angle < angle2)) ||
                (angle1 > angle2 && (angle1 <= angle || angle < angle2)))
            {
                xCollection.Add((float)x);
            }
        }
    }
}
```

The rather complex (but symmetrical) *if* clause in *TryY* accounts for the wrapping of angle values from π to –π. Notice also that the comparison of *angle* with *angle1* and *angle2* allows cases where *angle* equals *angle1* but not when *angle* equals *angle2*. It's allowing all angles from *angle1* (inclusive) up to but not including *angle2*.

For now, the final structure in the library is exactly the type of figure that PhingerPaint needs to draw: a line with rounded caps:

XNA Project: GeometryHelper   File: RoundCappedLines.cs (complete)

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace GeometryHelper
{
    public class RoundCappedLine : IGeometrySegment
    {
        LineSegment lineSegment1;
        ArcSegment arcSegment1;
        LineSegment lineSegment2;
```

```
        ArcSegment arcSegment2;

        public RoundCappedLine(Vector2 point1, Vector2 point2, float radius)
        {
            Point1 = point1;
            Point2 = point2;
            Radius = radius;

            Vector2 vector = point2 - point1;
            Vector2 normVect = vector;
            normVect.Normalize();

            Vector2 pt1a = Point1 + radius * new Vector2(normVect.Y, -normVect.X);
            Vector2 pt2a = pt1a + vector;
            Vector2 pt1b = Point1 + radius * new Vector2(-normVect.Y, normVect.X);
            Vector2 pt2b = pt1b + vector;

            lineSegment1 = new LineSegment(pt1a, pt2a);
            arcSegment1 = new ArcSegment(point2, radius, pt2a, pt2b);
            lineSegment2 = new LineSegment(pt2b, pt1b);
            arcSegment2 = new ArcSegment(point1, radius, pt1b, pt1a);
        }

        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }
        public float Radius { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            arcSegment1.GetAllX(y, xCollection);
            lineSegment1.GetAllX(y, xCollection);
            arcSegment2.GetAllX(y, xCollection);
            lineSegment2.GetAllX(y, xCollection);
        }
    }
}
```

This structure includes two *LineSegment* objects and two *ArcSegment* objects and defines them all based on the arguments to its own constructor. Implementing *GetAllX* is just a matter of calling the same method on the four components. It is the responsibility of the code calling *GetAllX* to ensure that the collection has previously been cleared. For *RoundCappedLines*, this method will return a collection with either one *x* value — a case that can be ignored for filling purposes — or two *x* values, in which case the pixels between those two *x* values can be filled.

The program itself is the PhingerPaint project that's part of the same PhingerPaint solution that also includes the GeometryHelper project. PhingerPaint must be given a reference to GeometryHelper. (I right-clicked References under the PhingerPaint project, selected Add Reference, clicked the Projects tab, and there it was.)

Game Components

To help you modularize your games, XNA supports a concept of game “components.” These can derive from the *GameComponent* class but very often they derive from *DrawableGameComponent* so they can display something on the screen in addition to what goes out in the *Draw* method of your *Game* class. To add a new component class to your project, you can right-click the project name, select Add and then New Item, and then pick *GameComponent* from the list. You’ll need to change the base class to *DrawableGameComponent* if you want the component to participate in drawing.

For PhingerPaint, I wanted a collection of little colored boxes that would allow the user to select a color to do some painting. Implementing these little boxes as *DrawableGameComponent* objects allowed this feature to be added fairly easily. Like the *Game* class, a *DrawableGameComponent* can override *Initialize*, *LoadComponent*, *Update*, and *Draw* methods, and use them in a very similar way.

The *ColorBlock* class isn’t long. Here it is in its entirety:

XNA Project: PhingerPaint   File: ColorBlock.cs (complete)

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input.Touch;

namespace PhingerPaint
{
    public class ColorBlock : DrawableGameComponent
    {
        SpriteBatch spriteBatch;
        Texture2D block;

        public ColorBlock(Game game) : base(game)
        {
        }

        public Color Color { set; get; }
        public Rectangle Destination { set; get; }
        public bool IsSelected { set; get; }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(this.GraphicsDevice);
            block = new Texture2D(this.GraphicsDevice, 1, 1);
```

```
            block.SetData<uint>(new uint[] { Color.White.PackedValue });

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            Rectangle rect = Destination;

            spriteBatch.Begin();
            spriteBatch.Draw(block, rect, IsSelected ? Color.White :
Color.DarkGray);
            rect.Inflate(-4, -4);
            spriteBatch.Draw(block, rect, Color);
            spriteBatch.End();

            base.Draw(gameTime);
        }
    }
}
```

ColorBlock relies on three public properties — *Color*, *Destination*, and *IsSelected* — to govern its appearance. Notice during the *LoadContent* method that it creates a *Texture2D* that is exactly one pixel in size! This *block* object is drawn twice in the *Draw* method. First it's drawn to the entire dimensions of the *Destination* rectangle as either dark gray or white, depending on the value of *IsSelected*. Then it's contracted in size by four pixels on all sides and drawn again based on *Color*.

PhingerPaint Concluded

The normal *Game1* class defines several fields:

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

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

    Viewport viewport;
    List<ColorBlock> colorBlocks = new List<ColorBlock>();
    Color drawingColor = Color.Blue;

    Texture2D canvas;
    uint[] pixels;
```

```
    List<float> xCollection = new List<float>();

    bool ignoreTouchId;
    int touchIdToIgnore;
    …
}
```

The *List* stores the 12 *ColorBlock* components, and *drawingColor* is the currently selected color. The main canvas is, of course, the *Texture2D* object called *canvas* and *pixels* are its pixels. The *xCollection* object is repeatedly reused in calls to the *RoundCappedLine* class.

The constructor sets the screen for portrait mode:

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

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    // Set to portrait mode
    graphics.PreferredBackBufferWidth = 480;
    graphics.PreferredBackBufferHeight = 800;
}
```

The *Initialize* override has several jobs to perform. It is responsible for creating the 12 *ColorBlock* objects and adding them to the *Components* collection of the *Game* class. This ensures that they get their own calls to *LoadContent*, *Update*, and *Draw*. The *Initialize* method is also responsible for setting the size and location of these components, finally creating the *canvas* object and *pixels* array for drawing.

```
protected override void Initialize()
{
    Color[] colors = { Color.Red, Color.Green, Color.Blue,
                       Color.Cyan, Color.Magenta, Color.Yellow,
                       Color.Black, new Color(0.2f, 0.2f, 0.2f),
                                    new Color(0.4f, 0.4f, 0.4f),
                                    new Color(0.6f, 0.6f, 0.6f),
                                    new Color(0.8f, 0.8f, 0.8f), Color.White };

    foreach (Color clr in colors)
    {
        ColorBlock colorBlock = new ColorBlock(this)
        {
            Color = clr,
            IsSelected = drawingColor == clr
```

```
        };
        colorBlocks.Add(colorBlock);
        this.Components.Add(colorBlock);
    }

    viewport = this.GraphicsDevice.Viewport;
    int colorBlockSize = viewport.Width / (colorBlocks.Count / 2) - 2;
    int xColorBlock = 2;
    int yColorBlock = viewport.Height - 2 * colorBlockSize - 2;

    foreach (ColorBlock colorBlock in colorBlocks)
    {
        colorBlock.Destination = new Rectangle(xColorBlock, yColorBlock,
                                               colorBlockSize, colorBlockSize);
        xColorBlock += colorBlockSize + 2;

        if (xColorBlock + colorBlockSize > viewport.Width)
        {
            xColorBlock = 2;
            yColorBlock += colorBlockSize + 2;
        }
    }

    int canvasHeight = viewport.Height - 2 * colorBlockSize - 2;
    canvas = new Texture2D(this.GraphicsDevice, viewport.Width, canvasHeight);
    pixels = new uint[canvas.Width * canvas.Height];
    canvas.GetData<uint>(pixels);

    for (int y = 0; y < canvas.Height; y++)
        for (int x = 0; x < canvas.Width; x++)
        {
            pixels[x + canvas.Width * y] = Color.Black.PackedValue;
        }

    canvas.SetData<uint>(pixels);

    base.Initialize();
}
```

The *LoadContent* method does nothing except its default job of creating the *SpriteBatch* object. But *Update* needs to handle touch input, both on the *ColorBlock* objects and the *canvas*:

XNA Project: PhingerPaint 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();
```

```
bool canvasNeedsUpdate = false;
int yMin = 0, yMax = 0;

foreach (TouchLocation touchLocation in touchLocations)
{
    if (ignoreTouchId && touchLocation.Id == touchIdToIgnore)
        continue;

    // Check for tap on ColorBlock
    if (touchLocation.State == TouchLocationState.Pressed)
    {
        Vector2 position = touchLocation.Position;
        ColorBlock newSelectedColorBlock = null;

        foreach (ColorBlock colorBlock in colorBlocks)
        {
            Rectangle rect = colorBlock.Destination;

            if (position.X >= rect.Left && position.X < rect.Right &&
                position.Y >= rect.Top && position.Y < rect.Bottom)
            {
                drawingColor = colorBlock.Color;
                newSelectedColorBlock = colorBlock;
            }
        }

        if (newSelectedColorBlock != null)
        {
            foreach (ColorBlock colorBlock in colorBlocks)
                colorBlock.IsSelected = colorBlock == newSelectedColorBlock;

            ignoreTouchId = true;
            touchIdToIgnore = touchLocation.Id;
        }
        else
        {
            ignoreTouchId = false;
        }
    }

    // Check for drawing movement
    else if (touchLocation.State == TouchLocationState.Moved)
    {
        TouchLocation prevTouchLocation;
        touchLocation.TryGetPreviousLocation(out prevTouchLocation);

        Vector2 point1 = prevTouchLocation.Position;
        Vector2 point2 = touchLocation.Position;

        // Sure hope touchLocation.Pressure comes back!
        float radius = 12; //  48 * touchLocation.Pressure;

        RoundCappedLine line = new RoundCappedLine(point1, point2, radius);

        yMin = (int)(Math.Min(point1.Y, point2.Y) - radius);
```

```
            yMax = (int)(Math.Max(point1.Y, point2.Y) + radius);

            yMin = Math.Max(0, Math.Min(canvas.Height, yMin));
            yMax = Math.Max(0, Math.Min(canvas.Height, yMax));

            for (int y = yMin; y < yMax; y++)
            {
                xCollection.Clear();
                line.GetAllX(y, xCollection);

                if (xCollection.Count == 2)
                {
                    int xMin = (int)(Math.Min(xCollection[0], xCollection[1]) +
0.5f);
                    int xMax = (int)(Math.Max(xCollection[0], xCollection[1]) +
0.5f);

                    xMin = Math.Max(0, Math.Min(canvas.Width, xMin));
                    xMax = Math.Max(0, Math.Min(canvas.Width, xMax));

                    for (int x = xMin; x < xMax; x++)
                    {
                        pixels[y * canvas.Width + x] = drawingColor.PackedValue;
                    }
                    canvasNeedsUpdate = true;
                }
            }
        }
    }

    if (canvasNeedsUpdate)
        canvas.SetData<uint>(pixels);

    base.Update(gameTime);
}
```

It's always very satisfying when everything has prepared the *Draw* override for a very simple job. The *ColorBlock* components draw themselves, so the *Draw* method here need only render the *canvas*:

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

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(canvas, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

PhreeCell and a Deck of Cards

I originally thought that my PhreeCell solitaire game would have no features beyond what was strictly necessary to play the game. My wife — who has played FreeCell under Windows and can usually win a deal — made it clear that PhreeCell would need two features that I hadn't planned on implementing: First and most importantly, there had to be some kind of positive feedback from the program acknowledging that the player has won. I implemented this as a *DrawableGameComponent* derivative called *CongratulationsComponent*.

The second essential feature was something I called "auto move." If a card can be legally moved to the suit piles at the upper right of the board, and there was no reason to do otherwise, then the card is automatically moved. Other than that, PhreeCell has no amenities. There is no animated "deal" at the beginning of play, you cannot simply "click" to indicate a destination spot, and there is no way to move multiple cards in one shot.

My coding for PhreeCell began not with an XNA program but with a Windows Presentation Foundation program to generate a single 1040 × 448 bitmap containing all 52 playing cards, each of which is 96 pixels wide and 112 pixels tall. This PlayingCardCreator program is included among the source code for this book; it uses mostly *TextBlock* objects to adorn a *Canvas* with numbers, letters, and suit symbols. It then passes the *Canvas* to a *RenderTargetBitmap* and saves the result out to a file named cards.png. In the XNA PhreeCell project, I added this file to the program's content.

Within the PhreeCell project, each card is an object of type CardInfo:

XNA Project: PhreeCell   File: CardInfo.cs

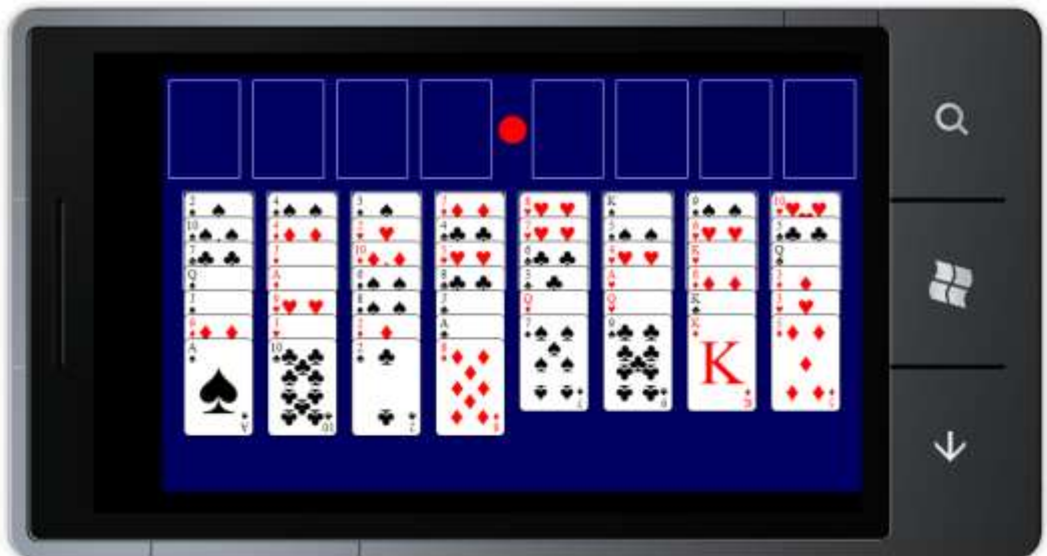
```
using System;
using Microsoft.Xna.Framework;

namespace PhreeCell
{
    class CardInfo
    {
        static string[] ranks = { "Ace", "Deuce", "Three", "Four",
                                  "Five", "Six", "Seven", "Eight",
                                  "Nine", "Ten", "Jack", "Queen", "King" };
        static string[] suits = { "Spades", "Clubs", "Hearts", "Diamonds" };

        public int Suit { protected set; get; }
        public int Rank { protected set; get; }

        public Vector2 AutoMoveOffset { set; get; }
        public TimeSpan AutoMoveTime { set; get; }
        public float AutoMoveInterpolation { set; get; }

        public CardInfo(int suit, int rank)
        {
```

```
            Suit = suit;
            Rank = rank;
        }

        // used for debugging purposes
        public override string ToString()
        {
            return ranks[Rank] + " of " + suits[Suit];
        }
    }
}
```

At first, this class simply had *Rank* and *Value* properties. I added the static *string* arrays and *ToString* for display purposes while debugging, and I added the three *AutoMove* fields when I implemented that feature. *CardInfo* itself has no information about where the card is actually located during play. That's retained elsewhere.

The Playing Field

Here's the opening screen of the completed PhreeCell program:



I'll assume you're familiar with the rules. All 52 cards are dealt face up in 8 columns that I refer to in the program as "piles." At the upper left are four spots for holding individual cards. I refer to these four card spots as "holds." At the upper-right are four spots for stacking ascending cards of the same suit; these are called "finals." The red dot in the middle is the replay button.

For convenience, I split the *Game1* class into two files. The first is the normal Game1.cs file; the second is named Game1.Helpers.cs. The Helpers file has no instance fields—just *const* and *static readonly*. The Game1.cs file has one *static* field and all the instance fields:

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

```
public partial class Game1 : Microsoft.Xna.Framework.Game
{
    static readonly TimeSpan AutoMoveDuration = TimeSpan.FromSeconds(0.25);

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    CongratulationsComponent congratsComponent;

    Texture2D cards;
    Texture2D surface;
    Rectangle[] cardSpots = new Rectangle[16];

    public Matrix DisplayMatrix { set; get; }
    Matrix inverseMatrix;

    CardInfo[] deck = new CardInfo[52];
    List<CardInfo>[] piles = new List<CardInfo>[8];
    CardInfo[] holds = new CardInfo[4];
    List<CardInfo>[] finals = new List<CardInfo>[4];

    CardInfo touchedCard;
    Vector2 touchedCardPosition;
    object touchedCardOrigin;
    int touchedCardOriginIndex;
    int touchedCardTouchId;
    …
}
```

The program uses only two *Texture2D* objects: The *cards* object is the bitmap containing all 52 cards; individual cards are displaying by defining rectangular subsets of this bitmap. The *surface* is the dark blue area you see in the screen shot that also includes the white rectangles and the red button. The coordinates of those 16 white rectangles — there are eight more under the top card in each pile — are stored in the *cardSpots* array.

The program was originally developed when portrait mode was the default. Because the program really wants landscape mode, it treats the display as if it were landscape mode, and then uses the *DisplayMatrix* to twist it sideways if necessary. The *inverseMatrix* is the inverse of that matrix and is useful for processing touch input. Landscape mode is now the default, but this logic has remained because it does some extra work that I'll discuss later.

The next block of fields are the basic data structures used by the program. The *deck* array contains all 52 *CardInfo* objects created early in the program and re-used until the program is

terminated. During play, copies of those cards are also in *piles*, *holds*, and *finals*. I originally thought *finals* would be an array like *holds* because only the top card need be displayed, but I discovered that the auto-move feature potentially required more cards to be visible.

The other fields are connected with touching and moving cards with the fingers. The *touchedCardPosition* field is the current position of the moving card. The *touchedCardOrigin* field stores the object where the moving card came from and is either the *holds* or *piles* array, while *touchedCardOriginIndex* is the array index. These are used to return the card to its original spot if the user tries to move the card illegally.

The *Game1* constructor does its normal stuff and loads the *CongratulationsComponent*:

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

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.IsFullScreen = true;
    Content.RootDirectory = "Content";

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    congratsComponent = new CongratulationsComponent(this);
    congratsComponent.Enabled = false;
    this.Components.Add(congratsComponent);
}
```

The *Initialize* method creates the *CardInfo* objects for the *decks* array, and initializes the *piles* and *finals* arrays with *List* objects.

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

```
protected override void Initialize()
{
    // Initialize deck
    for (int suit = 0; suit < 4; suit++)
        for (int rank = 0; rank < 13; rank++)
        {
            CardInfo cardInfo = new CardInfo(suit, rank);
            deck[suit * 13 + rank] = cardInfo;
        }

    // Create the List objects for the 8 piles
    for (int pile = 0; pile < 8; pile++)
        piles[pile] = new List<CardInfo>();

    // Create the List objects for the 4 finals
    for (int final = 0; final < 4; final++)
```

```
        finals[final] = new List<CardInfo>();

    base.Initialize();
}
```

The *LoadContent* method loads the bitmap containing the card images, and also calls three methods in the portion of the *Game1* class implemented in Game1.Helpers.cs:

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

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

    // Load large bitmap containing cards
    cards = this.Content.Load<Texture2D>("cards");

    // Create the 16 rectangular areas for the cards and the bitmap surface
    CreateCardSpots(cardSpots);
    surface = CreateSurface(this.GraphicsDevice, cardSpots);

    // Start the game!
    Replay();
}
```

The Game1.Helpers.cs file begins with a bunch of constant fields that define all the pixel dimensions of the playing field:

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

```
public partial class Game1 : Microsoft.Xna.Framework.Game
{
    const int wCard = 80;       // width of card
    const int hCard = 112;      // height of card

    // Horizontal measurements
    const int wSurface = 800;   // width of surface
    const int xGap = 16;        // space between piles
    const int xMargin = 8;      // margin on left and right

    // gap between "holds" and "finals"
    const int xMidGap = wSurface - (2 * xMargin + 8 * wCard + 6 * xGap);

    // additional margin on second row
    const int xIndent = (wSurface - (2 * xMargin + 8 * wCard + 7 * xGap)) / 2;

    // Vertical measurements
    const int yMargin = 8;      // vertical margin on top row
    const int yGap = 16;        // vertical margin between rows
    const int yOverlay = 28;    // visible top of cards in piles
```

```
    const int hSurface = 2 * yMargin + yGap + 2 * hCard + 19 * yOverlay;

    // Replay button
    const int radiusReplay = xMidGap / 2 - 8;
    static readonly Vector2 centerReplay =
                        new Vector2(wSurface / 2, xMargin + hCard / 2);
    …
}
```

Notice that *wSurface* — the width of the playing field — is defined to be 800 pixels because that's the width of the large phone display. However, the vertical dimension might need to be greater than 480. It is possible for there to be 20 overlapping cards in the *piles* area. To accommodate that possibility, *hSurface* is calculated as a maximum possible height based on these 20 overlapping cards.

The *CreateCardSpots* method uses those constants to calculate 16 *Rectangle* objects indicating where the cards are positioned on the playing fields. The top row has the *holds* and *finals*, and the bottom row is for the *piles*:

XNA Project: PhreeCell   File: Game1.Helper.cs (excerpt)

```
static void CreateCardSpots(Rectangle[] cardSpots)
{
    // Top row
    int x = xMargin;
    int y = yMargin;

    for (int i = 0; i < 8; i++)
    {
        cardSpots[i] = new Rectangle(x, y, wCard, hCard);
        x += wCard + (i == 3 ? xMidGap : xGap);
    }

    // Bottom row
    x = xMargin + xIndent;
    y += hCard + yGap;

    for (int i = 8; i < 16; i++)
    {
        cardSpots[i] = new Rectangle(x, y, wCard, hCard);
        x += wCard + xGap;
    }
}
```

The *CreateSurface* method creates the bitmap used for the playing field. The size of the bitmap is based on *hSurface* (set as a constant 800) and *wSurface*, which is much more than 480. To draw the white rectangles and red replay button, it directly manipulates pixels and sets those to the bitmap:

XNA Project: PhreeCell File: Game1.Helper.cs (excerpt)

```
static Texture2D CreateSurface(GraphicsDevice graphicsDevice, Rectangle[] cardSpots)
{
    uint backgroundColor = new Color(0, 0, 0x60).PackedValue;
    uint outlineColor = Color.White.PackedValue;
    uint replayColor = Color.Red.PackedValue;
    Texture2D surface = new Texture2D(graphicsDevice, wSurface, hSurface);
    uint[] pixels = new uint[wSurface * hSurface];

    for (int i = 0; i < pixels.Length; i++)
    {
        if ((new Vector2(i % wSurface, i / wSurface) - centerReplay).LengthSquared()
<
                                radiusReplay * radiusReplay)
            pixels[i] = replayColor;
        else
            pixels[i] = backgroundColor;
    }

    foreach (Rectangle rect in cardSpots)
    {
        // tops of rectangles
        for (int x = 0; x < wCard; x++)
        {
            pixels[(rect.Top - 1) * wSurface + rect.Left + x] = outlineColor;
            pixels[rect.Bottom * wSurface + rect.Left + x] = outlineColor;
        }
        // sides of rectangles
        for (int y = 0; y < hCard; y++)
        {
            pixels[(rect.Top + y) * wSurface + rect.Left - 1] = outlineColor;
            pixels[(rect.Top + y) * wSurface + rect.Right] = outlineColor;
        }
    }

    surface.SetData<uint>(pixels);
    return surface;
}
```

The other static methods in the *Game1* class are fairly self-explanatory.

XNA Project: PhreeCell File: Game1.Helper.cs (excerpt)

```
static void ShuffleDeck(CardInfo[] deck)
{
    Random rand = new Random();

    for (int card = 0; card < 52; card++)
    {
        int random = rand.Next(52);
        CardInfo swap = deck[card];
```

```
        deck[card] = deck[random];
        deck[random] = swap;
    }
}

static bool IsWithinRectangle(Vector2 point, Rectangle rect)
{
    return point.X >= rect.Left &&
           point.X <= rect.Right &&
           point.Y >= rect.Top &&
           point.Y <= rect.Bottom;
}

static Rectangle GetCardTextureSource(CardInfo cardInfo)
{
    return new Rectangle(wCard * cardInfo.Rank,
                         hCard * cardInfo.Suit, wCard, hCard);
}

static CardInfo TopCard(List<CardInfo> cardInfos)
{
    if (cardInfos.Count > 0)
        return cardInfos[cardInfos.Count - 1];

    return null;
}
```

GetCardTextureSource is used in conjunction with the large *cards* bitmap. It simply returns a *Rectangle* object corresponding to a particular card. *TopCard* actually returns the last item in a *List<CardInfo>* collection, which is useful for obtaining the topmost card in one of the *piles* or *finals* collections.

The *LoadContent* method in Card1.cs concluded by calling *Replay*. Here it is:

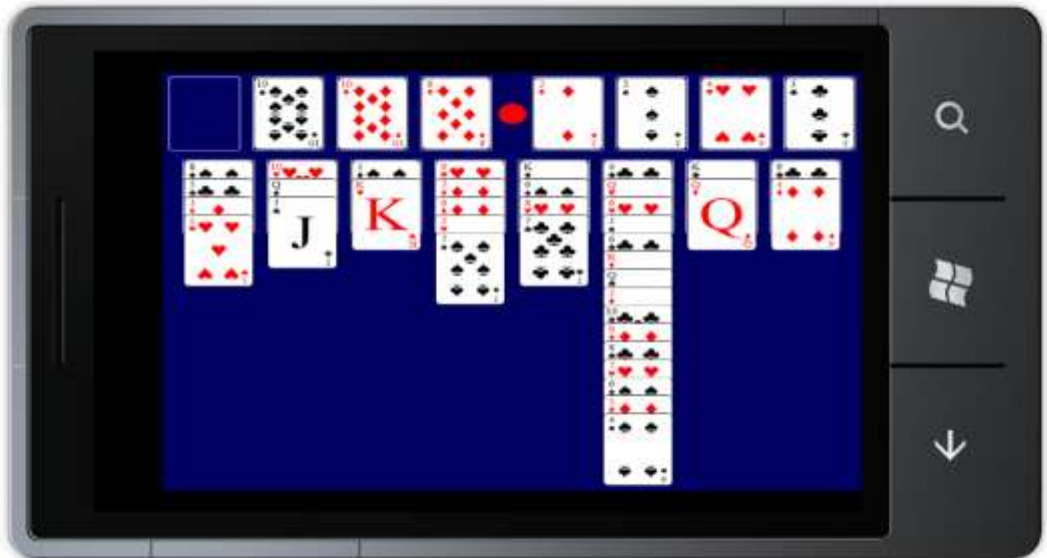
XNA Project: PhreeCell File: Game1.Helper.cs (excerpt)

```
void Replay()
{
    for (int i = 0; i < 4; i++)
        holds[i] = null;

    foreach (List<CardInfo> final in finals)
        final.Clear();

    foreach (List<CardInfo> pile in piles)
        pile.Clear();

    ShuffleDeck(deck);

    // Apportion cards to piles
    for (int card = 0; card < 52; card++)
    {
```

```
            piles[card % 8].Add(deck[card]);
    }
    CalculateDisplayMatrix();
}
```

The method clears out the *holds* array, and the *finals* and *piles* collections, randomizes the deck of cards, and apportions them into the eight collections in *piles*. The method is concluded with a call to *CalculateDisplayMatrix*. This is not the only time this method is called. Any time a card is moved from, or added to, one of the piles collection, the display matrix is re-calculated just in case.

This matrix serves several purposes. It was originally responsible for turning the playing field sideways for landscape mode. That's no longer an issue. The matrix also scales the playing field if the program runs on a smaller display. And for a large landscape display, it still has a important function, which is shortening the height if more space is required for viewing all the cards in the *piles* area. The program doesn't handle this issue very elegantly. It merely makes the entire playing field a little shorter, including all the cards and even the replay button:



I'm not entirely happy with this solution, but here's the *CalculateDisplayMatrix* method that makes it all possible:

XNA Project: PhreeCell File: Game1.Helper.cs (excerpt)

```
void CalculateDisplayMatrix()
{
    Viewport viewport = this.GraphicsDevice.Viewport;
    int largeDimension = Math.Max(viewport.Width, viewport.Height);
```

```
    int smallDimension = Math.Min(viewport.Width, viewport.Height);

    // Initialize the matrix
    DisplayMatrix = Matrix.Identity;
    float scale = 1;

    // Find basic scaling based on the widest dimension
    if (largeDimension != wSurface)
    {
        scale = (float)largeDimension / wSurface;
        DisplayMatrix *= Matrix.CreateScale(scale);
    }

    // Figure out the total required height and scale vertically
    int maxCardsInPiles = 0;

    foreach (List<CardInfo> pile in piles)
        maxCardsInPiles = Math.Max(maxCardsInPiles, pile.Count);
        int requiredHeight = 2 * yMargin + yGap + 2 * hCard +
                                        yOverlay * (maxCardsInPiles - 1);

    if (scale * requiredHeight > smallDimension)
        DisplayMatrix *= Matrix.CreateScale(1, smallDimension /
                                         (scale * requiredHeight), 1);

    // Rotate if display is in portrait mode
    if (largeDimension != viewport.Width)
    {
        DisplayMatrix *= Matrix.CreateRotationZ(MathHelper.PiOver2);
        DisplayMatrix *= Matrix.CreateTranslation(smallDimension, 0, 0);
    }

    // Find the inverse matrix for hit-testing
    inverseMatrix = Matrix.Invert(DisplayMatrix);
}
```

DisplayMatrix is defined as a public property rather than a field because it is also accessed by the *CongratulationsComponent* class. In a commercial program, I would definitely design a second set of cards for the small display; these would certainly be more attractive than cards that are scaled to 60% of their designed size.

Most crucially, the *DisplayMatrix* is used in the *Begin* call of *SpriteBatch* so it's applied to everything in one grand swoop. Although just a little bit out of my customary sequence, you are now ready to look at the *Draw* method in the *Game1* class.

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

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin(SpriteSortMode.Immediate, null, null, null, null, null,
                      DisplayMatrix);
```

```
    spriteBatch.Draw(surface, Vector2.Zero, Color.White);

    // Draw holds
    for (int hold = 0; hold < 4; hold++)
    {
        CardInfo cardInfo = holds[hold];

        if (cardInfo != null)
        {
            Rectangle source = GetCardTextureSource(cardInfo);
            Vector2 destination = new Vector2(cardSpots[hold].X, cardSpots[hold].Y);
            spriteBatch.Draw(cards, destination, source, Color.White);
        }
    }

    // Draw piles
    for (int pile = 0; pile < 8; pile++)
    {
        Rectangle cardSpot = cardSpots[pile + 8];

        for (int card = 0; card < piles[pile].Count; card++)
        {
            CardInfo cardInfo = piles[pile][card];
            Rectangle source = GetCardTextureSource(cardInfo);
            Vector2 destination = new Vector2(cardSpot.X, cardSpot.Y + card *
yOverlay);
            spriteBatch.Draw(cards, destination, source, Color.White);
        }
    }

    // Draw finals including all previous cards (for auto-move)
    for (int pass = 0; pass < 2; pass++)
    {
        for (int final = 0; final < 4; final++)
        {
            for (int card = 0; card < finals[final].Count; card++)
            {
                CardInfo cardInfo = finals[final][card];

                if (pass == 0 && cardInfo.AutoMoveInterpolation == 0 ||
                    pass == 1 && cardInfo.AutoMoveInterpolation != 0)
                {
                    Rectangle source = GetCardTextureSource(cardInfo);
                    Vector2 destination =
                            new Vector2(cardSpots[final + 4].X,
                                        cardSpots[final + 4].Y) +
                                  cardInfo.AutoMoveInterpolation *
cardInfo.AutoMoveOffset;
                    spriteBatch.Draw(cards, destination, source, Color.White);
                }
            }
        }
    }

    // Draw touched card
```

```
    if (touchedCard != null)
    {
        Rectangle source = GetCardTextureSource(touchedCard);
        spriteBatch.Draw(cards, touchedCardPosition, source, Color.White);
    }

    spriteBatch.End();

    base.Draw(gameTime);
}
```

After calling *Begin* on the *SpriteBatch* object and displaying the *surface* bitmap for the playing field, the method is ready for drawing cards. It begins with the easy one — the four possible cards in the *holds* array. The little *GetCardTextureSource* method returns a *Rectangle* for the position of the card within the cards bitmap, and the *cardSpot* array provides the point where each card is to appear.

The next section is a little more complicated. When displaying the cards in the *piles* area, the *cardSpot* location must be overset to accommodate the overlapping cards. The problematic area is the *finals*, and it's problematic because of the auto-move feature. As you'll see, when a card is eligible for auto-move, it is removed from its previous *holds* array or *piles* collection and put into a *finals* collection. However, the location of the card must be animated from its previous position to its new position. This is the purpose of the *AutoMoveOffset* and *AutoMoveInterpolation* properties that are part of *CardInfo*.

However, the *Draw* method wants to display each of the four *finals* collections sequentially from left to right, and then within each collection from the beginning (which is always an ace) to the end, which is the topmost card. I discovered this didn't always work, and an animated card sometimes seemed briefly to slide under a card in one of the other *finals* stacks. That's why the loop to display the *finals* collections has two passes — one for the non-animated cards and another for any animated auto-move cards. (Although the program only animates one card at a time, an earlier version animated multiple cards.)

Draw finishes with the card that the user might be currently dragging with touch.

The *Update* method is concerned almost exclusively with implementing the animation for the auto-move feature and processing touch, but methods that *Update* calls implicitly enforce all the rules of the game.

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

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

```
    // Process auto-move card and perhaps initiate next auto-move
    bool checkForNextAutoMove = false;

    foreach (List<CardInfo> final in finals)
        foreach (CardInfo cardInfo in final)
        {
            if (cardInfo.AutoMoveTime > TimeSpan.Zero)
            {
                cardInfo.AutoMoveTime -= gameTime.ElapsedGameTime;

                if (cardInfo.AutoMoveTime <= TimeSpan.Zero)
                {
                    cardInfo.AutoMoveTime = TimeSpan.Zero;
                    checkForNextAutoMove = true;
                }
                cardInfo.AutoMoveInterpolation = (float)cardInfo.AutoMoveTime.Ticks
                                                                    /
AutoMoveDuration.Ticks;
            }
        }

    if (checkForNextAutoMove && !AnalyzeForAutoMove() && HasWon())
    {
         congratsComponent.Enabled = true;
    }

    TouchCollection touchLocations = TouchPanel.GetState();

    // Try to pick up a card
    if (touchedCard == null)
    {
        foreach (TouchLocation touchLocation in touchLocations)
        {
            // Finger pressed to screen
            if (touchLocation.State == TouchLocationState.Pressed)
            {
                Vector2 position = Vector2.Transform(touchLocation.Position,
inverseMatrix);

                // Replay pressed?
                if ((position - centerReplay).Length() < radiusReplay)
                {
                    congratsComponent.Enabled = false;
                    Replay();
                }
                // Now try to pick up a card
                else if (TryPickUpCard(position))
                {
                    touchedCardTouchId = touchLocation.Id;
                }
            }
        }
    }
    else
```

```
    {
        foreach (TouchLocation touchLocation in touchLocations)
        {
            // If a card is being moved, only process IDs for that card
            if (touchLocation.Id == touchedCardTouchId)
            {
                if (touchLocation.State == TouchLocationState.Moved)
                {
                    TouchLocation previousTouchLocation;
                    touchLocation.TryGetPreviousLocation(out previousTouchLocation);
                    Vector2 moveVector = touchLocation.Position -

previousTouchLocation.Position;

                    Vector2 position = Vector2.Transform(touchedCardPosition,
                                                         DisplayMatrix);
                    position += moveVector;
                    touchedCardPosition = Vector2.Transform(position,
inverseMatrix);
                }
                // The finger has been lifted from the screen: Try to set down the
card
                else
                {
                    if (TryPutDownCard(touchedCard))
                    {
                        CalculateDisplayMatrix();

                        // This will not happen!
                        if (!AnalyzeForAutoMove() && HasWon())
                        {
                            congratsComponent.Enabled = true;
                        }
                    }
                    touchedCard = null;
                }
            }
        }
    }
    base.Update(gameTime);
}
```

Following the animation for auto-move cards, the method checks if the user is trying to “pick up” a card by touching it. If a card is already being moved and being “set down,” then other methods must be called in the Game1.Helpers.cs file.

The logic to determine what cards (if any) should be auto-moved turned out to be one of the lengthier parts of the program:

XNA Project: PhreeCell   File: Game1.Helpers.cs (excerpt)

```
bool AnalyzeForAutoMove()
{
    for (int hold = 0; hold < 4; hold++)
    {
        CardInfo cardInfo = holds[hold];

        if (cardInfo != null && CheckForAutoMove(cardInfo))
        {
            holds[hold] = null;
            cardInfo.AutoMoveOffset += new Vector2(cardSpots[hold].X,
cardSpots[hold].Y);
            cardInfo.AutoMoveInterpolation = 1;
            cardInfo.AutoMoveTime = AutoMoveDuration;
            return true;
        }
    }

    for (int pile = 0; pile < 8; pile++)
    {
        CardInfo cardInfo = TopCard(piles[pile]);

        if (cardInfo != null && CheckForAutoMove(cardInfo))
        {
            piles[pile].Remove(cardInfo);
            cardInfo.AutoMoveOffset += new Vector2(cardSpots[pile + 8].X,
                                  cardSpots[pile + 8].Y + piles[pile].Count *
yOverlay);
            cardInfo.AutoMoveInterpolation = 1;
            cardInfo.AutoMoveTime = AutoMoveDuration;
            return true;
        }
    }
    return false;
}

bool CheckForAutoMove(CardInfo cardInfo)
{
    if (cardInfo.Rank == 0)     // ie, ace
    {
        for (int final = 0; final < 4; final++)
            if (finals[final].Count == 0)
            {
                finals[final].Add(cardInfo);
                cardInfo.AutoMoveOffset = -new Vector2(cardSpots[final + 4].X,
                                                       cardSpots[final + 4].Y);
                return true;
            }
    }
    else if (cardInfo.Rank == 1)    // ie, deuce
    {
        for (int final = 0; final < 4; final++)
        {
            CardInfo topCardInfo = TopCard(finals[final]);

            if (topCardInfo != null &&
```

```
                topCardInfo.Suit == cardInfo.Suit &&
                topCardInfo.Rank == 0)
            {
                finals[final].Add(cardInfo);
                cardInfo.AutoMoveOffset = -new Vector2(cardSpots[final + 4].X,
                                                       cardSpots[final + 4].Y);
                return true;
            }
        }
    }
    else
    {
        int slot = -1;
        int count = 0;

        for (int final = 0; final < 4; final++)
        {
            CardInfo topCardInfo = TopCard(finals[final]);

            if (topCardInfo != null)
            {
                if (topCardInfo.Suit == cardInfo.Suit &&
                    topCardInfo.Rank == cardInfo.Rank - 1)
                {
                    slot = final;
                }
                else if (topCardInfo.Suit < 2 != cardInfo.Suit < 2 &&
                        topCardInfo.Rank >= cardInfo.Rank - 1)
                {
                    count++;
                }
            }
        }
        if (slot >= 0 && count == 2)
        {
            cardInfo.AutoMoveOffset = -new Vector2(cardSpots[slot + 4].X,
                                                   cardSpots[slot + 4].Y);
            finals[slot].Add(cardInfo);
            return true;
        }
    }
    return false;
}
```

The actual rules for picking up and setting down cards are almost as complex:

XNA Project: PhreeCell   File: Game1.Helpers.cs (excerpt)

```
bool TryPickUpCard(Vector2 position)
{
    for (int hold = 0; hold < 4; hold++)
    {
        if (holds[hold] != null && IsWithinRectangle(position, cardSpots[hold]))
```

```
        {
            Point pt = cardSpots[hold].Location;

            touchedCard = holds[hold];
            touchedCardOrigin = holds;
            touchedCardOriginIndex = hold;
            touchedCardPosition = new Vector2(pt.X, pt.Y);
            holds[hold] = null;
            return true;
        }
    }

    for (int pile = 0; pile < 8; pile++)
    {
        if (piles[pile].Count > 0)
        {
            Rectangle pileSpot = cardSpots[pile + 8];
            pileSpot.Offset(0, yOverlay * (piles[pile].Count - 1));

            if (IsWithinRectangle(position, pileSpot))
            {
                Point pt = pileSpot.Location;
                int pileIndex = piles[pile].Count - 1;

                touchedCard = piles[pile][pileIndex];
                touchedCardOrigin = piles;
                touchedCardOriginIndex = pile;
                touchedCardPosition = new Vector2(pt.X, pt.Y);
                piles[pile].RemoveAt(pileIndex);
                return true;
            }
        }
    }
    return false;
}

bool TryPutDownCard(CardInfo touchedCard)
{
    Vector2 cardCenter = new Vector2(touchedCardPosition.X + wCard / 2,
                                     touchedCardPosition.Y + hCard / 2);

    for (int cardSpot = 0; cardSpot < 16; cardSpot++)
    {
        Rectangle rect = cardSpots[cardSpot];

        // Greatly expand the card-spot rectangle for the piles
        if (cardSpot >= 8)
           rect.Inflate(0, hSurface - rect.Bottom);

        if (IsWithinRectangle(cardCenter, rect))
        {
            // Check if the hold is empty
            if (cardSpot < 4)
            {
                int hold = cardSpot;
```

```
            if (holds[hold] == null)
            {
                holds[hold] = touchedCard;
                return true;
            }
        }

        else if (cardSpot < 8)
        {
            int final = cardSpot - 4;

            if (TopCard(finals[final]) == null)
            {
                if (touchedCard.Rank == 0)  // ie, an ace
                {
                    finals[final].Add(touchedCard);
                    return true;
                }
            }
            else if (touchedCard.Suit == TopCard(finals[final]).Suit &&
                     touchedCard.Rank == TopCard(finals[final]).Rank + 1)
            {
                finals[final].Add(touchedCard);
                return true;
            }
        }
        else
        {
            int pile = cardSpot - 8;

            if (piles[pile].Count == 0)
            {
                piles[pile].Add(touchedCard);
                return true;
            }
            else
            {
                CardInfo topCard = TopCard(piles[pile]);

                if (touchedCard.Suit < 2 != topCard.Suit < 2 &&
                    touchedCard.Rank == topCard.Rank - 1)
                {
                    piles[pile].Add(touchedCard);
                    return true;
                }
            }
        }

        // The card was in a card-spot rectangle but wasn't a legal drop
        break;
    }
}

// Restore the card to its original place
```

```
    if (touchedCardOrigin is CardInfo[])
    {
        (touchedCardOrigin as CardInfo[])[touchedCardOriginIndex] = touchedCard;
    }
    else
    {
        ((touchedCardOrigin as
List<CardInfo>[])[touchedCardOriginIndex]).Add(touchedCard);
    }
    return false;
}
```

But all that work is justified by a return value of *true* from the following method:

XNA Project: PhreeCell   File: Game1.Helpers.cs (excerpt)

```
bool HasWon()
{
    bool hasWon = true;

    foreach (List<CardInfo> cardInfos in finals)
        hasWon &= cardInfos.Count > 0 && TopCard(cardInfos).Rank == 12;

    return hasWon;
}
```

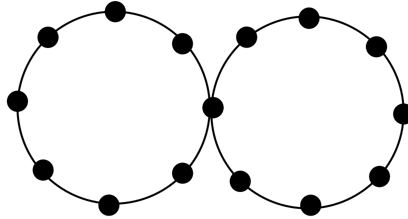
The *Update* method uses that to trigger the *CongratulationsComponent*, shown here in its entirety:

XNA Project: PhreeCell   File: CongratulationsComponent.cs

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace PhreeCell
{
    public class CongratulationsComponent : DrawableGameComponent
    {
        const float SCALE_SPEED = 0.5f;                     // half-size per second
        const float ROTATE_SPEED = 3 * MathHelper.TwoPi;    // 3 revolutions per
second

        SpriteBatch spriteBatch;
        SpriteFont pericles108;
        string congratulationsText = "You Won!";
        float textScale;
        float textAngle;
        Vector2 textPosition;
        Vector2 textOrigin;
```

```
        public CongratulationsComponent(Game game) : base(game)
        {
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(this.GraphicsDevice);
            pericles108 = this.Game.Content.Load<SpriteFont>("Pericles108");
            textOrigin = pericles108.MeasureString(congratulationsText) / 2;
            Viewport viewport = this.GraphicsDevice.Viewport;
            textPosition = new Vector2(Math.Max(viewport.Width, viewport.Height) /
2,
                                       Math.Min(viewport.Width, viewport.Height) /
2);
            base.LoadContent();
        }

        protected override void OnEnabledChanged(object sender, EventArgs args)
        {
            Visible = Enabled;

            if (Enabled)
            {
                textScale = 0;
                textAngle = 0;
            }
        }

        public override void Update(GameTime gameTime)
        {
            if (textScale < 1)
            {
                textScale +=
                    SCALE_SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
                textAngle +=
                    ROTATE_SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
            }
            else if (textAngle != 0)
            {
                textScale = 1;
                textAngle = 0;
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            spriteBatch.Begin(SpriteSortMode.Immediate, null, null, null, null,
null,
                              (this.Game as Game1).DisplayMatrix);
            spriteBatch.DrawString(pericles108, congratulationsText, textPosition,
                                   Color.White, textAngle, textOrigin, textScale,
                                   SpriteEffects.None, 0);
```

```
            spriteBatch.End();
            base.Draw(gameTime);
        }
    }
}
```



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)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.1f;           // laps per second
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Viewport viewport;
    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);
    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 *LoadContent* method needs to load the care and get its center point:

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

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;
    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.TotalSeconds) % 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 = viewport.Width > viewport.Height;

    if (isX == isLandscape)
        return xCurve.Evaluate(t) * (viewport.Width - 2 * car.Width) / 4 + car.Width;

    return yCurve.Evaluate(t) * (viewport.Height - 2 * car.Width) / 2 + 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.