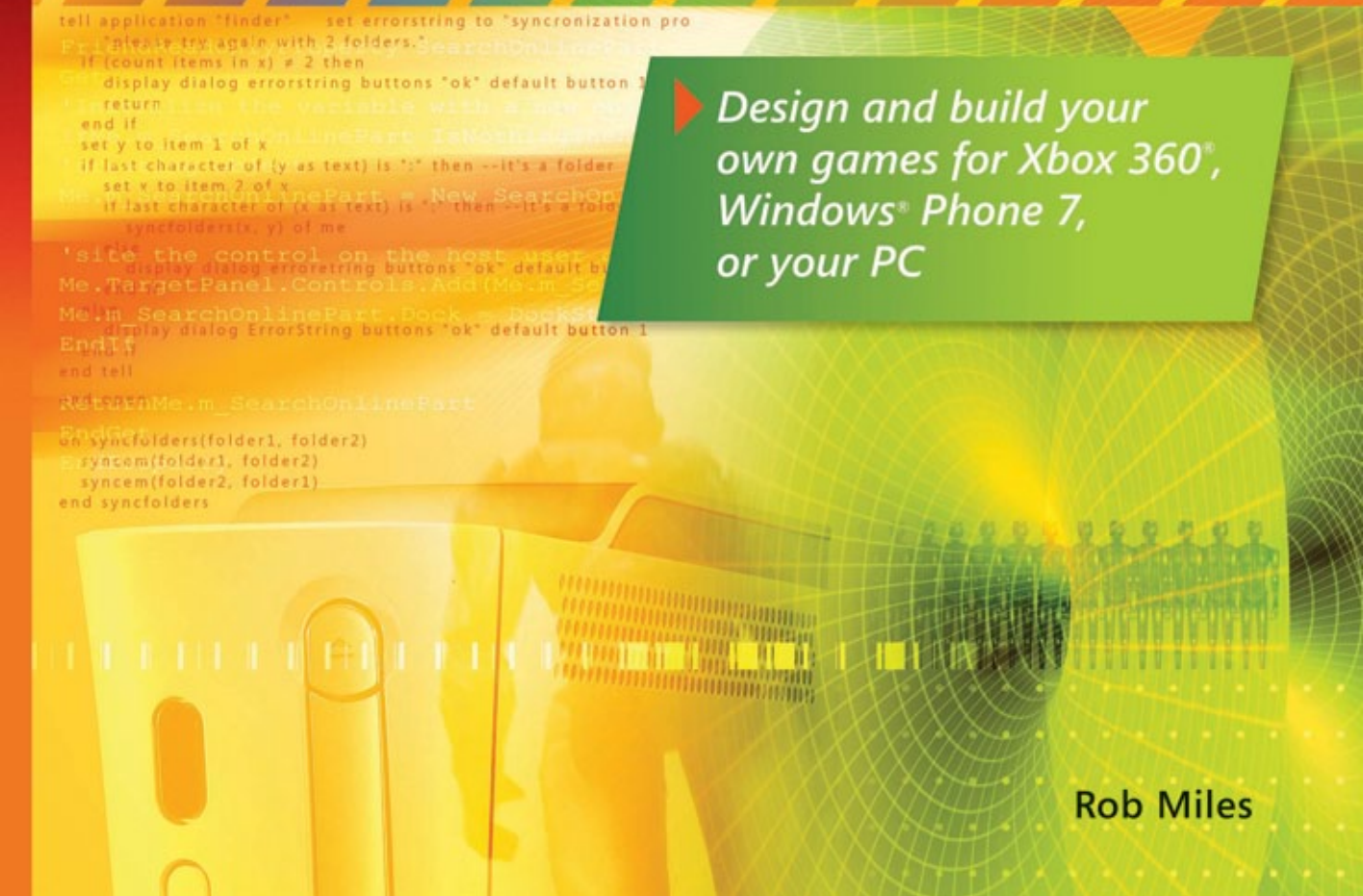
Microsoft

Learn Programming Now!

Microsoft® XNA® Game Studio 4.0

Design and build your own games for Xbox 360®, Windows® Phone 7, or your PC



Rob Miles

Microsoft®

Microsoft® XNA® Game Studio 4.0: Learn Programming Now!

Rob Miles

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2001012345
ISBN: 978-0-7356-5157-9

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Valerie Woolley
Editorial and Production: Waypoint Press
Technical Reviewers: Nick Gravelyn, Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Girvin

Body Part No. X17-37448

To Jake, a great dog who is much missed.

Table of Contents

Acknowledgments

I'm not sure if you are meant to have fun writing books, but I do. Thanks to Devon Musgrave, Ben Ryan, Valerie Woolley, and Steve Sagman for making everything fit so well together and to Kurt Meyer and Nick Gravely for making sure it all makes sense. I must also mention the XNA team who keep making a great thing better, year on year, and the Windows Phone team who have made something amazing.

Introduction

With Microsoft XNA, Microsoft is doing something really special. It is providing an accessible means for people to create programs for the Windows PC, Xbox 360, and Windows Phone. Now pretty much anyone can take a game idea, run it on a genuine console, and even send it to market in Xbox Live or the Windows Phone Marketplace.

This book shows you how to make game programs and run them on an Xbox 360, a Microsoft Windows PC, or a Windows Phone device. It also gives you an insight into how software is created and what being a programmer is really like.

Who This Book Is For

If you have always fancied writing software but have no idea how to start, then this book is for you. If you have ever played a computer game and thought, "I wonder how they do that?" or, better yet, "I want to make something like that," then this book will get you started with some very silly games that you and all your friends can have a go at playing and modifying. Along the way, you'll also get a decent understanding of C#, which is a massively popular programming language used by many thousands of software developers all over the world. The C# skills that you pick up in this book can also be used as the basis of a career in programming, should you find that you really enjoy writing programs. And because the design of the C# language is very similar to C, C++, and Java, you will find that your skills can be used with them too.

The book is structured into 19 chapters, starting with the simplest possible XNA program and moving on to show you how to use the Xbox gamepad, the keyboard, sounds, graphics, and network in your games. In the course of learning how to use C# and XNA, you create some very silly games, including Color Nerve, Mind Reader, Gamepad Racer, Bread and Cheese, and Button Bash. You can even download the full versions of these games from *http://www.verysillygames.com* and use them at your next party. The final section shows you how to take your programming skills and use them to create games for the Windows Phone device.

With this book, I show you that programming is a fun, creative activity that lets you bring your ideas to life.

System Requirements

You need the following hardware and software to build and run the code samples for this book. Chapter 1, "Computers, C#, XNA, and You," explains how to set up your environment.

- A Windows PC with 3-D graphics acceleration if you want to run your XNA games on your PC.
- Microsoft Windows Vista or Windows 7.
- Microsoft Visual Studio 2010 C# Express Edition for Windows Phone, Visual Studio 2010 Standard Edition, Visual Studio 2010 Professional Edition, or Visual Studio 2010 Team Suite.
- To test your games on a console, you need an Xbox 360 fitted with a hard disk. Your Xbox 360 must be connected to Xbox Live, and you need to join the App Hub. You will find out how to do this in Chapter 1.
- If you have a Windows Phone you can run XNA games on that as well. Any Windows Phone device can be connected to your PC so you can load your XNA games into it.

Code Samples

All the code samples discussed in this book can be downloaded from the book's detail page, located at:

http://oreilly.com/catalog/9780735651579

Display the detail page in your Web browser, and follow the instructions for downloading the files.

There are also code samples and games at *http://www.verysillygames.com*.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site at Oreilly.com:

1. Go to *http://microsoftpress.oreilly.com*.
2. In the **Search** box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, under the cover image, you'll see a list of links.
5. Click **View/Submit Errata.**

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey.

The survey is short, and we read *every one* of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

Chapter 3

Getting Player Input

In this chapter, you will

- Find out how Microsoft XNA represents the gamepads and keyboards.
- Discover the C# language structures that let us get player input.
- Write some really silly games and scare people with them.

Introduction

You now know the basics of computer game programming. You know that a program is actually a sequence of statements, each of which performs a single action. You have seen that statements are held inside methods, each of which performs a particular task, and that methods are held in classes along with data. The program itself works on data values, which are held in variables of a particular type, and the program can make decisions based on the values that the variables have. (If none of this makes much sense, reread Chapter 2, "Programs, Data, and Pretty Colors," until it does.)

Now you are going to expand your understanding to include how to receive input from the outside world so that games can actually react to what the player does. You shall see that once we have done this, a number of possibilities open up, and you can create some truly silly games, including "Color Nerve," "Mind Reader," "The Thing That Goes Bump in the Night," and "Gamepad Racer."

Program Project: A Mood Light Controller

In Chapter 2, you created a light that changes color over time. I also mentioned that this is the kind of thing that will be used in the starships of the future. A color-changing light is not all that useful for reading books, but it's great for setting moods; what our starship captain really needs is a light that she can set to any color. So now you are going to make a lamp that can be controlled by an Xbox gamepad. The user presses the red, blue, green, and yellow buttons on the gamepad to increase the amount of that color in the light. To make this work, you have to discover how to read the gamepad.

Before you start looking at gamepads, though, you need to decide how the program will actually work. Consider the following statement of C# from the previous mood-light program, which is part of the `Update` method:

```
if (redCountingUp) redIntensity++;
```

This is one of the tests that controls the intensity of the red part of the color. What it is saying is "If the Boolean value `redCountingUp` is `True`, increase the value of `redIntensity` by 1." The statement is processed each time `Update` is called (at the moment that is 60 times a second), so this means that if `redCountingUp` is `True`, the red intensity of the screen gets progressively brighter over time.

You want to write some code that says, "If the red button on Gamepad 1 is being pressed, increase the value of `redIntensity` by 1." Then, if the player holds down the button, the screen gets redder. So all you have to do is change this test to read the button on the gamepad, and you can create a user-controlled light easily.

Reading a Gamepad

The gamepads are actually very complex devices. They are connected to the host device either by a universal serial bus (USB) cable or by a wireless connection. As far as you are concerned, the way that programs work with gamepads does not depend on how they are connected. The connection to a gamepad can be used to read the buttons and joysticks and can also be used to send commands to the gamepad—for example, to turn the vibration effect on and off. The Xbox and XNA provide support for up to four gamepads connected simultaneously.

Gamepads and Classes

The gamepad information is represented in XNA by means of a class called `GamePadState`. The job of this class is to provide the connection between the program and the physical gamepad that the player is holding. To understand how you are going to use this, you have to learn a bit more about how classes work.

You have already seen what a class is in the section "Games and Classes" in Chapter 2. A class contains data (variables that can hold stuff) and methods (code that can do stuff). You can think of a class as an office, with a desk holding the variables and people acting as the methods. Figure 3-1 shows the office plan for the class `Game1`, which you have seen is the basis of an XNA game.

This class contains some variables on the desk (in this case, the background color intensities) and two methods, which we have called Mr. Draw and Mrs. Update. Each method has a corresponding telephone. Programs can place calls to the telephones to request that the method perform the required task.

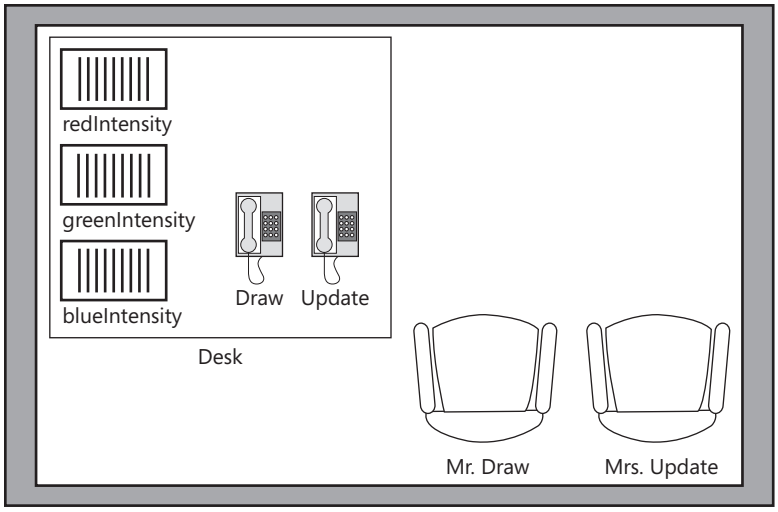


FIGURE 3-1 The `Game1` class as an office plan.

The Great Programmer Speaks: Classes Are Not Really Offices Our Great Programmer has been reading these notes and finds them quite amusing. She says that classes are not exactly like offices, but she thinks that for the purpose of getting an understanding of how programs are constructed, it is okay to regard them as such.

When an XNA game starts, the XNA system makes an *instance* of the `Game1` class that it then can ask to `Draw` and `Update`. When an instance of a class is created, the instructions for the methods that it contains are loaded into memory and space is set aside for the data variables that the instance holds.

The class files that you write give the plans for the class so when the program runs, instances of each class can be created. In real life, you would make a game office by building a room, putting a desk and some telephones in the room, and then hiring a Mr. Draw and a Mrs. Update. The process of making an instance of a class is similar. However, to save memory, the running program uses only one copy of the method code, which is shared among all the instances of a class.

Note It is important to remember that this happens when a program runs. The process of creating instances of classes is not performed by the compiler. The job of the compiler is to convert your C# source code into instructions that the target device runs. By the time that your program has control, the compiler has done its job, and the computer is just running the machine language output that the compiler produced.

Finding a Gamepad

XNA also looks after a lot of other things when a game is running, one of which is the `GamePad` class connected to all the gamepads. You don't have to know how the gamepad is actually connected; for all you know, it might use tiny pixies traveling up and down the wires carrying pixie notes written on pixie paper saying, "Master has pressed the Red Button," but then again it might not. Figure 3-2 shows how the `GamePad` class would look if it were an office.

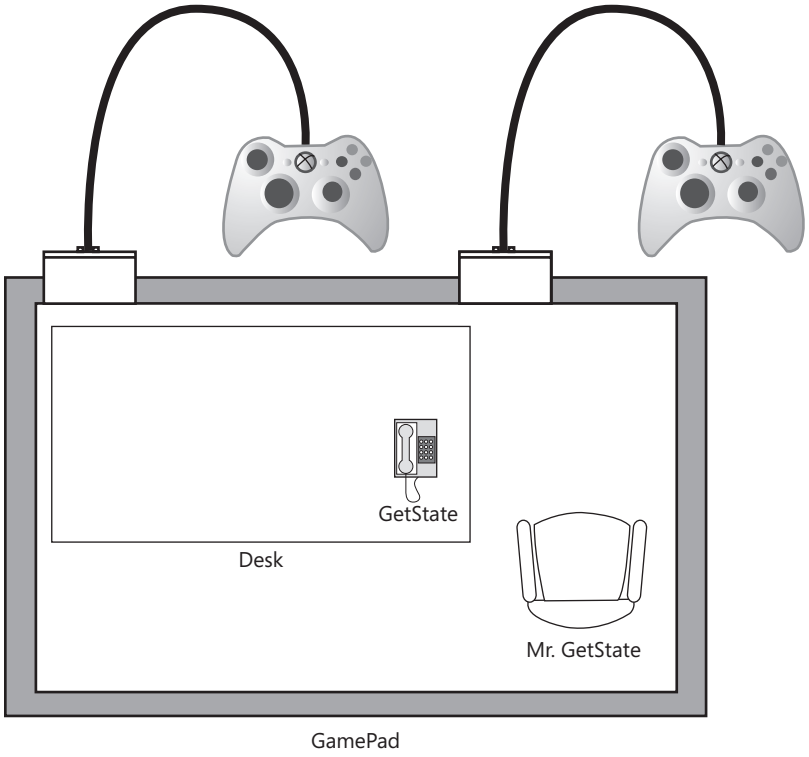


FIGURE 3-2 The `GamePad` class as an office.

The `GamePad` class contains a method called `GetState`, which gets the state of one of the gamepads. When `GetState` is called, it looks at one of the gamepads, reads its settings, and then sends information back for use in the statement it was called from.

The `GetState` method is supplied with a parameter that identifies the gamepad to be read. A *parameter* is a way that a call can give information to a method. You have seen these before; in your very first programs, you were passing `Color` parameters into the `Clear` method to select the color of the screen that you wanted.

In the case of the `GetState` method, the parameter identifies the gamepad that you want to read. If you are thinking in terms of offices, you can think of a parameter as part of the instructions that come down the telephone. When the phone rings and Mr. GetState answers it,

he is asked, "Get me the state of Gamepad 1." The information about the state of the gamepad is sent back in a `GamePadState` structure, which is shown in Figure 3-3.

GamePadState	
Buttons	
Green A	ButtonState.Pressed
Red B	ButtonState.Released
Blue X	ButtonState.Released
Yellow Y	ButtonState.Released
Start	ButtonState.Released
Back	ButtonState.Released

FIGURE 3-3 `GamePadState` structure with the green A button pressed.

You can think of this as a set of items filled in on a form if you wish, but actually it is a C# structure that contains the data members shown in Figure 3-3, as well as some other data.

So, if Mrs. Update wants to know the state of one of the gamepads on the Xbox, she calls the `GetState` method in the `GamePad` class and asks, "Can you give me the state of the gamepad for Player 1, please?" Mr. GetState jumps up, fills in a "GamePadState" form, and sends it back to her. Figure 3-4 gives the breakdown of the C# statement that gets the state of a gamepad into a variable of type `GamePadState`.

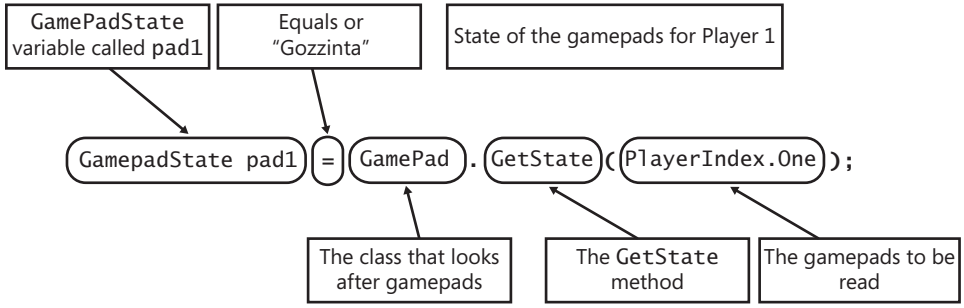


FIGURE 3-4 Getting the status of a gamepad.

Testing the Gamepad Status

Now that you have the status, you can use it in the program to see if a button has been pressed. Figure 3-5 shows the breakdown of the C# statement that will perform the test.

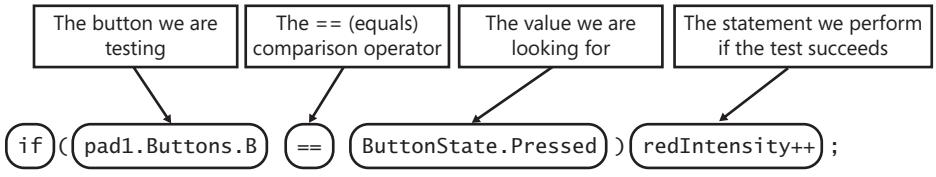


FIGURE 3-5 Testing a button on a gamepad.

This compares the state of the red button B with the value `ButtonState.Pressed`. If the two are equal, this means that the button is down, and the `Update` method must make the red intensity bigger. You can then use the same principle to manage the blue and green values, which means that you now have an `Update` method that looks like the following:

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

  GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

  if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
  if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
  if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;

  base.Update(gameTime);
}
```

The only problem with the `Update` method described here is that the program doesn't handle the yellow button yet. When the yellow button is pressed, the program needs to increase the green and the red intensities; that is, it must perform two statements if the condition is true. It turns out that doing so is very easy; you can just put the two statements into a block that is controlled by the condition, as shown here:

```
if (pad1.Buttons.Y == ButtonState.Pressed)
{
    redIntensity++;
    greenIntensity++;
}
```

You have seen blocks before; the body of a method (the bit that does the work) is a block. In C# terms, a *block* is a number of statements that are enclosed in curly braces. The code shown here performs both statements if the condition is true because they are in a block controlled by the condition.

> **The Great Programmer Speaks: Blocks Rock** Our Great Programmer tends to use blocks after `if` conditions even when she doesn't actually need to. She says that it makes the program text clearer, and that it is much easier to add extra statements later if you need to.

If you put the preceding statements into the `Update` method of one of your earlier Mood Light programs, you get compiler warning messages because the new version of `Update` doesn't use all the variables that were created for previous versions of the program. To get

rid of these warnings, you must delete the statements that create the unused variables. The Great Programmer doesn't like it when programs have variables in them that are not used. She says this looks unprofessional, and I agree with her.

Sample Code: Manual MoodLight All the sample projects can be obtained from the Web resources for this text, which can be found at *http://oreilly.com/catalog/9780735651579*. The sample project in the directory "01 Manual MoodLight" in the resources for this chapter implements the `Update` method, as shown in this section. You can increase the brightness of the colors on the screen by pressing the buttons on the gamepad.

Game Idea: Color Nerve

Every now and then, we are going to try out a game idea. These start out very simply and then build up to more complicated and interesting games. You can use the Manual MoodLight code to create your first game. The game uses something we saw in Chapter 2. You noticed that if you keep making a value bigger, there comes a point where it won't fit in the memory store allocated for it, and then it overflows. This is what caused the screen to go from bright white to black. However, you can use this to create our first "Very Silly Game."

Color Nerve is a game for two or more players. The players take turns pressing one or more buttons on the gamepad. (The other players must watch carefully to make sure that they actually do press at least one button.) Each player can press as many buttons as he wants for as long as he wants during his turn, but if the screen changes suddenly (because one of the color values has gone from 255 to 0), he is out, and the game continues. The last player left in the game is the winner.

This game can be very tactical. Players can press the buttons for very short times, or at the start of the game, they can show their nerve by holding the buttons down for longer periods, trying to cause problems for the next player. They can also try to work out which color has wrapped around so that they can press that button when it is their turn. The game works very well at parties, any number of people can take part, and the rules are very easy to understand. In Chapter 4, "Displaying Images," you will improve the game to add pictures as well as a plain screen.

Using the Keyboard

XNA works with keyboards as well as with gamepads. You might be surprised to learn that you can plug a USB keyboard into an Xbox 360 and use it just as you'd use the keyboard on the PC. If you want the program to work with the keyboard, you can add code that does this, as shown here:

```
KeyboardState keys = Keyboard.GetState();

if (keys.IsKeyDown(Keys.R)) redIntensity++;
if (keys.IsKeyDown(Keys.B)) blueIntensity++;
if (keys.IsKeyDown(Keys.G)) greenIntensity++;
if (keys.IsKeyDown(Keys.Y))
{
    redIntensity++;
    greenIntensity++;
}
```

Note that the process is very similar to how the gamepad works, but there are slight differences. You don't need to tell the `GetState` method on the `Keyboard` which keyboard to read because XNA supports only a single keyboard. The `KeyboardState` item that is returned from the call is not actually a piece of paper; instead, it is an object that provides methods that the program can use to discover whether a particular key is pressed. Rather than seeing if the state of a button is set to the value `ButtonState.Pressed`, the program can call the method `IsKeyDown`. You supply the `IsKeyDown` method with a parameter that identifies the key you are interested in, as follows:

```
if (keys.IsKeyDown(Keys.R)) redIntensity++;
```

This code is a conditional statement that increases the value of `redIntensity` if the R key is pressed. The method `IsKeyDown` returns `true` if the key is down and `false` if not. You can, therefore, use it to control the update of the `redIntensity` value.

Stopping the Game with the Escape Key

The `Update` method that is created when you make a new XNA game contains a test that checks for the Back button on gamepad 1 and calls the `Exit` method to stop the game when the Back button is pressed. If you are using a keyboard instead of a gamepad you will not be able to press this button to stop the game. You can add a test for the Escape key on the keyboard. This key is a "control" key, in that it does not actually relate to a printable character, but is designed to signal an action you want the program to take. Other control keys include the Enter key and the Backspace key. You can use the same `IsKeyDown` method to test for the Escape key.

```
if (keys.IsKeyDown(Keys.Escape)) Exit();
```

This code stops the game when the Escape key is pressed.

Using a Gamepad and a Keyboard at the Same Time

If you want to use a gamepad and a keyboard simultaneously, you have to test for both. This means that the `Update` method now looks like this:

```
protected override void Update(GameTime gameTime)
{
  GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

  if (pad1.Buttons.Back == ButtonState.Pressed) Exit();
  if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
  if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
  if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;
  if (pad1.Buttons.Y == ButtonState.Pressed)
  if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
  {
    redIntensity++;
    greenIntensity++;
  }

  KeyboardState keys = Keyboard.GetState();

  if (keys.IsKeyDown(Keys.Escape)) Exit();

  if (keys.IsKeyDown(Keys.R)) redIntensity++;
  if (keys.IsKeyDown(Keys.B)) blueIntensity++;
  if (keys.IsKeyDown(Keys.G)) greenIntensity++;
  if (keys.IsKeyDown(Keys.Y))
  {
    redIntensity++;
    greenIntensity++;
  }
  base.Update(gameTime);
}
```

This code is not good because you are doing the same thing twice, just triggered in a different way. The Great Programmer, if she ever saw this, would not be impressed. Fortunately C# provides a way that a program can combine two conditions and then perform some code if either condition is true. This way of combining conditions is called the *OR* logical operator because it is true if one thing or the other is true, and it is written in the program as two vertical bars (||):

```
GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
KeyboardState keys = Keyboard.GetState();

if (pad1.Buttons.B == ButtonState.Pressed ||
    keys.IsKeyDown(Keys.R)) redIntensity++;
```

The *OR* logical operator is placed between two Boolean expressions that can be either true or false. If one or the other expression is true, the combined logical condition works out to be true.

In this code, if the red button is pressed on the gamepad *or* the R key is pressed on the keyboard (or both), the `redIntensity` value increases. This is exactly what you want, and it means that Color Nerve can now be played with the gamepad or the keyboard (or both at the same time). Logical operators are so called because they produce logical rather than numerical results. There are other logical operators that you will use as you create more complex programs.

Note If you find this logical operator stuff hard to understand, just go back to the problem that you are trying to solve. You want the program to perform a statement (`redIntensity++`) if the red key is pressed on the gamepad *or* if the R key is pressed on the keyboard. So you use the *OR* operator (||) to combine the two tests and make a condition that triggers if one or the other condition is true.

Sample Code: Color Nerve The sample project in the directory "02 Color Nerve" in the resources for this chapter implements the game. You can adjust the colors of the screen by pressing the gamepad buttons or a key on the keyboard.

Adding Vibration

The communication between the gamepad and the game works in both directions. Not only can you read buttons on the gamepad, but also you can send commands to the gamepad to turn on the vibration motors. Again, you don't have to know exactly how these messages are delivered; all you need to know is the features of XNA that are used to control this vibration effect.

This means you can make your Color Nerve game even more exciting by making the gamepad vibrate when the intensity values are getting close to their limits. It is interesting how features like this can enhance even a simple game. You will be using the vibration effect on the gamepads quite a lot in the next few games.

Controlling the Vibration of a Gamepad

The `GamePad` class provides a method called `SetVibration` that lets a program control the vibration motors:

```
GamePad.SetVibration(PlayerIndex.One, 0, 1);
```

The `SetVibration` method uses three parameters. The first one identifies which gamepad you want to vibrate. The second parameter is a value between 0.0 and 1 that controls the vibration of the left motor. The bigger the number, the more the gamepad vibrates. The third parameter controls the right motor in the same way as the left one. The statement shown

here would set the right motor of Gamepad 1 vibrating at full speed. The left motor is the low-frequency vibration, and the right motor is the high-frequency vibration.

If you think of the `GamePad` class/office having a man called Mr. SetVibration, this means that he would be told which gamepad to vibrate and the settings for the left and right motors. Once the method has been called, the gamepad starts to vibrate, and it keeps vibrating until you call the method again to change its setting. In other words, you can think of the `SetVibration` method as a switch that can be set to a number of different positions. Initially, both of the gamepad motors are set at 0, which means no vibration.

Testing Intensity Values

The game needs to decide when to turn on the vibration. To do this, it must test the intensity values and turn on the vibration motor if any of them is getting too large. The program can decide to turn on the motors if any of the red, green, or blue intensity values is greater than 220. To do this, the program must test the intensity values as follows:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
```

This code shows another form of condition. In the previous examples, the conditions have been checking to see if two values are equal. This code tests if one value is greater than another. The greater-than sign (>) is another logical operator. Placed between two values, it returns `true` if the value on the left is greater than the value on the right and `false` if not. That is exactly what you want.

Using the preceding code, the gamepad starts to vibrate using the right motor when the red intensity value goes above 220. If you add this code to the `Update` method in the Color Nerve game, you find that if you increase the red value, the gamepad starts to vibrate. Unfortunately, our program has a bug. When the red intensity value returns to 0, the vibration does not stop. You need to add some code that turns off the motor when the intensity value is less than 220. It turns out that this is very easy to do—you can add an `else` part to the condition:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

The statement after the `else` is performed if the condition is found to be `false`. (You can add an `else` part to any `if` condition that you create.) This means that when the red intensity value returns to 0, the vibration stops. You can extend the tests using *OR* so that the program tests all the intensity values:

```
if ( redIntensity > 220 ||
     greenIntensity > 220 ||
     blueIntensity > 220 )
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

Now the vibration is controlled by all the intensity values. As an improvement to the game, you might want to experiment with different kinds of vibration for different colors, perhaps by using the low-frequency motor as well. This is controlled by the other value in the call of `SetVibration`:

```
GamePad.SetVibration(PlayerIndex.One, 1, 0);
```

The line of code shown here turns on the low-frequency vibration. You might also want to experiment with the thresholds at which the vibration starts.

The program still has one more problem. If you run it and make the gamepad vibrate, when the program finishes, the gamepad doesn't always stop vibrating. You need to add code that stops the vibration when the game ends. The game stops when the player presses the Back button on the gamepad. The test for this is in the `Update` method. If the Back button is pressed, the `Exit` method is called to stop the game:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
```

The `Exit` method removes the game display and shuts the game down in a tidy fashion. What the program must do is turn off the gamepad motors before `Exit` is called. To do this, the program needs to perform more than one statement if the Back button is pressed, so we need another block:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
    this.Exit();
}
```

Now, when the player presses the Back button to end the program, the vibration motors are turned off.

The Great Programmer Speaks: When in Doubt, Make Sure Yourself The Great Programmer says that if you are in a situation where you are not sure whether something is always the case, you should add code to remove all possible doubt. Testing the vibration behavior described in this section, I discovered that the gamepad is left vibrating on earlier versions of XNA, but not on some newer ones. To make absolutely sure that the vibration stops regardless of the version of XNA under which your game runs, you should include the code to stop the vibration yourself.

Sample Code: Vibration Color Nerve Game The sample project in the "03 Color Nerve with Vibes" directory in the source code resources for this chapter holds a version of Color Nerve that has the vibration effect enabled.

Game Idea: Secret Vibration Messages

Once you see that it is easy to read gamepad buttons and drive the motors, you can start to have more fun with XNA, particularly with wireless gamepads. You can create mind-reading games where your assistant seems to know exactly what you are thinking. What the audience doesn't know is that both of you are holding Xbox gamepads in your jacket pockets and using them to send signals back and forth using the vibration feature. The code to do this is actually very simple, and you should be able to understand what it does:

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        this.Exit();
    }

    GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
    GamePadState pad2 = GamePad.GetState(PlayerIndex.Two);

    if (pad1.Buttons.A == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.Two, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
    }

    if (pad2.Buttons.A == ButtonState.Pressed)
```

```
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
    }

    base.Update(gameTime);
}
```

The `Update` method reads the A button on the gamepad for Player 1. If this is pressed, it turns on the fast vibration motor in the gamepad for Player 2. It then repeats the process the other way, sending signals from Gamepad 2 to Gamepad 1. This gives you a way in which you can send wireless signals from one gamepad to another. Note that both conditions have `else` parts so that if the button is not pressed, the vibration is turned off.

You could also use this for practical jokes; for example, just leave a gamepad underneath your victim's bed and then wait until he turns the light off and settles down. Then give the vibration a quick blast for the maximum scare factor. Just don't blame me if you never get the gamepad back!

Sample Code: Vibration Messages The sample project in the "04 Mind Reader" directory in the source code resources for this chapter holds a version of the vibration message program. Just remember to use it wisely. The program also turns the display screen black so that it is not obvious that there is a program running.

Game Idea: Gamepad Racer

The final game idea in this chapter is really silly, but it can be great fun. The first thing you need to do is find a large, smooth table. Put a couple of books under the legs at one end so that the table is sloping, not horizontal. If you put a wireless Xbox gamepad at the top of the table and make the gamepad vibrate, it slides down the table toward the other end. You may need to experiment with the angle, but I've found that with care, you can arrange things so that a gamepad takes around 30 seconds to slide all the way down the table with vibration at full power. If you line up four gamepads on the top of the table, players can pick the one they think will win, and then you can race them down the slope.

The code for this game is very simple indeed; the `Update` method just turns on all the vibration motors in the gamepads:

```
protected override void Update(GameTime gameTime)
```

```
{
    // Allows the game to exit
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        GamePad.SetVibration(PlayerIndex.Three, 0, 0);
        GamePad.SetVibration(PlayerIndex.Four, 0, 0);
        this.Exit();
    }

    GamePad.SetVibration(PlayerIndex.One, 1, 1);
    GamePad.SetVibration(PlayerIndex.Two, 1, 1);
    GamePad.SetVibration(PlayerIndex.Three, 1, 1);
    GamePad.SetVibration(PlayerIndex.Four, 1, 1);

    base.Update(gameTime);
}
```

The only complication is that when the game ends, you must turn off all the vibrations. Put all the gamepads at the top of the slope and then run the program. Press the Back button on Gamepad 1 to stop the game.

Sample Code: Gamepad Racer The sample project in the "05 GamepadRacer" directory in the source code resources for this chapter holds a version of the Gamepad Racer program.

Note By carefully tuning vibration values it is possible to "sabotage" gamepads so that the same one wins each time. Note that I do not condone such behavior.

Program Bugs

Your younger brother is still trying to learn to program, but he keeps having problems. He claims that this book is faulty because the programs don't work properly when he types them in. He is trying to get the Color Nerve game to work, but every time he runs the program, the yellow intensity gets brighter whether he presses the button or not. You take a look at his program and find the following code in the `Update` method:

```
if (pad1.Buttons.Y == ButtonState.Pressed ||
   keys.IsKeyDown(Keys.Y)) ;
{
   redIntensity++;
   greenIntensity++;
}
```

This is the only part of the program where the yellow intensity is being increased, and it seems that the condition is being ignored.

This looks perfectly okay, and it seems to compile and run correctly, but it seems to be making the yellow intensity brighter every time. At this point, it is a good idea to look at Microsoft Visual Studio and see if the compiler is trying to tell you anything about the code. Figure 3-6 shows your brother's code after he has compiled it.

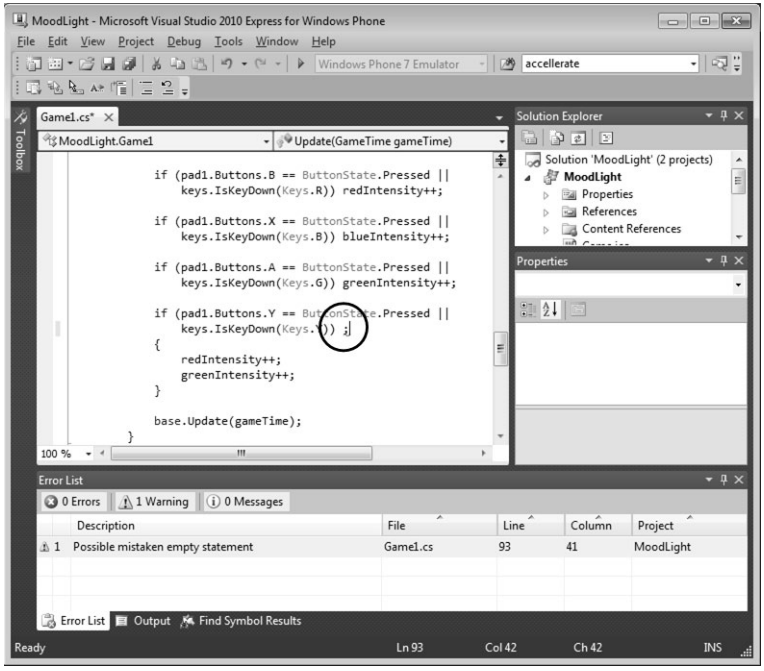


FIGURE 3-6 Visual Studio compiler warning display.

Your attention is drawn to the bottom left corner, where the message "Possible mistaken empty statement" appears. If you double-click this message, you find that the cursor moves to a point just after the `if` condition (I've drawn a circle around it in Figure 3-6).

The C# compiler is trying to tell us something about this statement. If we go back to the original listing, we find that your brother has added an extra semicolon at the end of the condition. The problem is that this ends the statement controlled by the condition. So if the R button or the R key is pressed or the Dpad is pressed down, the program decides to do nothing (an empty statement) and then goes on and performs the next statements no matter what, leading to the effect that we are seeing. Figure 3-7 shows how this happens.

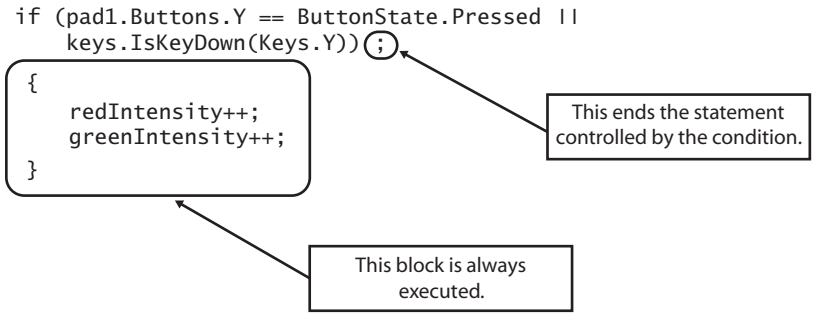


FIGURE 3-7 The effect of an extra semicolon.

You remove the semicolon, the warning goes away, and the program works fine. Your younger brother is now starting to revise his opinion of you and offers to take out the trash that night, even though it is your turn.

The Great Programmer Speaks: Helping Other People Is a Good Plan The Great Programmer has been watching all this with approval. She figures that it is always a good idea to try to help people who are stuck with a problem. Sometimes when a programmer working on uncovering a bug has the chance to explain what is going wrong with a piece of code to an innocent bystander, that can be enough to allow the programmer to work out what is broken. That means you can get a reputation as a fearsome bug fixer just by standing by. Furthermore, seeing what mistakes other people make can give you hints on things that you need to look out for when your programs go wrong. Oh, and sometimes you get your trash taken out for free.

Conclusion

You have learned a lot in this chapter, and you have finally managed to create some games that players can have fun with. You have seen how XNA allows programs to interact with physical devices by calling methods on classes, and we have seen how a program can make decisions on the information that it receives from the devices and use this to make simple (and silly) games.

Chapter Review Questions

No chapter would be complete without a review. So here it is. You should know the routine by now; just decide whether a statement is true or false and look the answers up in Appendix A at the back of the book to find out whether you are a winner or a loser.

1. If a class is an office, a method is a desk.
2. The compiler creates all the instances of classes in a program.
3. An `if` statement must have an `else` part.
4. A parameter is used to feed information into a class.
5. The `else` part of an `if` statement is always performed.
6. The state of a gamepad is represented in an XNA program by a byte value.
7. The `GamePad.GetState` method can be used to see if a button is pressed on a gamepad (this is a tough question; you are allowed to look at the chapter to work it out).
8. A block is a number of C# statements enclosed in curly brackets.
9. The C# condition `(true || false)` means "true or false" and would work out to `true`.
10. The C# condition `(redIntensity > 220)` evaluates to `true` if the value in `greenIntensity` is greater than 220.
11. The gamepad vibration always turns off automatically when an XNA game stops running.

Index

Numbers and Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z