XNA 3.0 Game Programming Recipes: A Problem-Solution Approach

ISBN-13 (pbk): 978-1-4302-1855-5

ISBN-13 (electronic): 978-1-4302-1856-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Development Editor: Joohn Choe
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editors: Heather Lang, Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Linda Weidemann, Wolf Creek Publishing Services
Proofreader: April Eddy
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`. You may need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Getting Started with XNA 3.0

The first part of this chapter will get you up and running with XNA 3.0 by guiding you through the installation process and helping you get your code running on a PC and on the Xbox 360 console. The second part of this chapter contains some more advanced topics for those interested in the inner workings of the XNA Framework.

Specifically, the recipes in this chapter cover the following:

- Installing XNA Game Studio 3.0 and starting your first XNA 3.0 project (recipes 1-1 and 1-2)
- Running your code on the PC, Zune, and the Xbox 360 console (recipes 1-3, 1-4, and 1-5)
- Learning more about the timing followed by the XNA Framework (recipe 1-6)
- Making your code plug-and-play using `GameComponent` classes and `GameServices` (recipes 1-7 and 1-8)
- Allowing users to save and load their games using XNA's storage capabilities (recipe 1-9)

1-1. Install XNA Game Studio 3.0

The Problem

You want to start coding your own games.

The Solution

Before you can start coding your own games, you should install your development environment. XNA Game Studio 3.0 allows you to create your whole game project using a single environment. Best of all, it's completely free to install.

First, you need a version of Visual Studio 2008 that allows you to develop C# programs. This is required because XNA uses C#.

On top of Visual Studio 2008, you will install XNA Game Studio 3.0.

How It Works

Installing Visual Studio 2008

XNA Game Studio 3.0 requires Visual Studio 2008 to be installed on your PC. If this is not yet the case, you can download the Visual C# 2008 Express Edition for free.

To do this, go to `www.microsoft.com/express/download/` and find the Visual C# 2008 Express Edition box (make sure you select the C# edition, indicated by the green color). Select the language of your choice, and hit the Download button. This will download a very small file, which you should run afterward.

During setup, use the default selections and hit Next until the program starts downloading and installing.

■ **Note** You can find updated links to these packages on the Download section of my site (`www.riemers.net`).

Installing XNA Game Studio 3.0

Go to `http://creators.xna.com/en-US/downloads`, and click the Download XNA Game Studio 3.0 link at the bottom-right corner of the top box. On the page that opens, click the Download button (just above the Quick Details section) to download XNA Game Studio 3.0.

Once you've downloaded and run the file, the installer will check whether you have installed Visual C# Express Edition 2008. If you have followed the instructions in the previous section, you shouldn't be getting any error messages.

During setup, you will be presented with the Firewall Setup page. Make sure you select the first option, and allow both suboptions. If you don't, you will run into trouble when connecting to your Xbox 360 or when testing multiplayer games between multiple PCs. Keep in mind that when you experience trouble when connecting your game over other PCs or consoles, the problem might be caused by incorrect or third-party firewall settings.

Finally, hit the Install button to install XNA Game Studio 3.0.

1-2. Start Your First XNA 3.0 Project

The Problem

You want to start coding a new XNA 3.0 game. In addition, the default startup code already contains a few methods, so you want to know what these are for and how they help make your life easier.

The Solution

Opening a new project is the same in most Windows programs. In XNA Game Studio 3.0, go to the File menu, and select New ➤ Project.

How It Works

Starting XNA Game Studio 3.0

Start XNA Game Studio 3.0 by clicking the Start button and selecting Programs. Find Microsoft XNA Game Studio 3.0, click it, and select Microsoft Visual Studio 2008 (or Microsoft Visual C# 2008 Express Edition if you installed the free version).

Starting a New XNA 3.0 Project

In XNA Game Studio 3.0, open the File menu, and select New ➤ Project. In the list on the left, XNA Game Studio 3.0 under Visual C# should be highlighted by default, as shown in Figure 1-1. On the right, highlight Windows Game (3.0). Give your new project a fancy name, and hit the OK button.

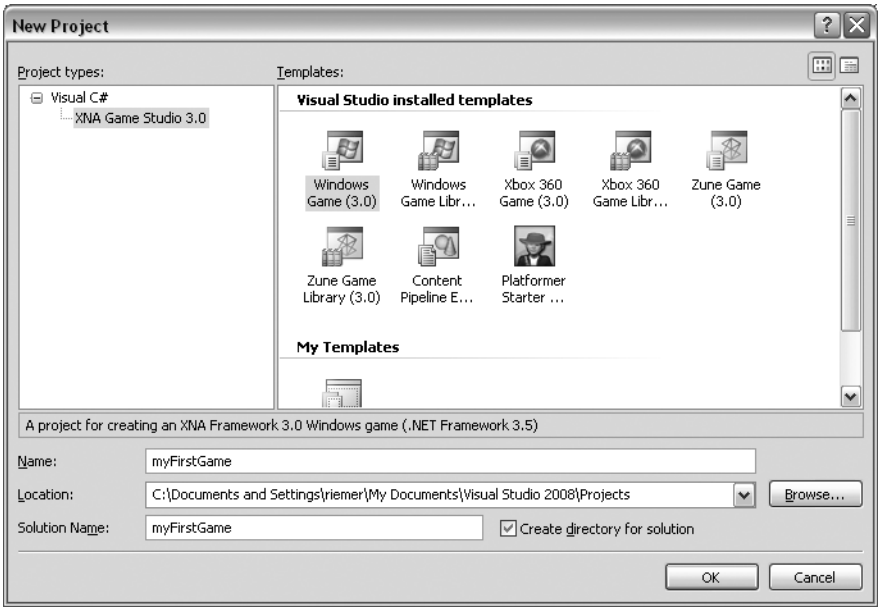

Figure 1-1. *Starting a new XNA 3.0 project (Visual Studio 2008 Express Edition)*

Examining the Predefined Methods

When you start a new XNA 3.0 project, you will get a code file already containing some code. Comments (shown in green) make up more than 50 percent of the code to help you get started.

In a few moments, you will find that the methods you're presented with really are quite useful, because they greatly reduce the time you would otherwise spend doing basic stuff. For example, when you run your project at this moment, you will already be presented with an empty window, meaning you don't have to waste your time coding a window or processing the window's message queue.

The predefined methods are discussed in the following sections.

Game1 Constructor

The `Game1` method is called once, at the very moment your project is run. This means none of the internal clockwork has been initialized the moment this method (the constructor) is called. The only code you should add here consists of the instantiations of `GameComponent` classes (see recipe 1-7), because you cannot access any resources (such as the `GraphicsDevice` class) since they haven't been initialized at this point.

Initialize Method

The `Initialize` method is also called once, after all the internal initialization has been done. This method is the ideal place to set the initial values of the objects in your game, such as the starting positions and starting speeds of the objects of your game. You have full access to all resources of your `Game` object.

Update Method

When running your game, XNA will make its best effort to call the `Update` method exactly 60 times per second (once every 0.0167 seconds). For more information on this timing, including how to change it, read recipe 1-6.

This makes the `Update` method an excellent place to put code that updates the logic of your game. This can include updating the positions of your objects, checking whether some objects collide, starting an explosion at that position, and increasing the score.

Also, processing user input and updating camera/model matrices should be done here.

Draw Method

In this method, you should put the code that renders your scene to the screen. It should render all 2D images, 3D objects, and explosions to the screen, as well as display the current score.

By default, the `Draw` method is called at the same frequency as the screen refresh rate, which depends on the screen or on the Zune device. See recipe 1-6 for more information on the `Draw` method.

LoadContent Method

Whenever you start a game, you will want to load art (such as images, models, and audio) from disk. To speed things up and allow a lot of flexibility, XNA manages this art through the content pipeline.

All art loading should be done in the `LoadContent` method. This method is called only once at the beginning of your project.

A detailed example on how to load a 2D image into your XNA project is given in recipe 2-1. The same approach can be used to load any kind of art.

UnloadContent Method

If some of the objects used in your game require specific disposing or unloading, the `UnloadContent` method is the ideal spot to do this. It is called once, before the game exits.

Adding an .fx HLSL File

If you want to go one step further and add an HLSL file to your project, simply find the Content entry in your Solution Explorer at the top-right of your screen. Right-click it, and select Add ➤ New Item. Select "Effect file," and give it a name of your choice.

You'll get some default code, which you'll want to extend or replace with code you find elsewhere in this book. After that, you need to import it like any other content object: by creating a suitable variable and linking this file to that variable.

Add this variable to the top of your main `Game` class:

```
Effect myEffect;
```

Then link it to your code file in the `LoadContent` method:

```
protected override void LoadContent()
{
    myEffect = Content.Load<Effect>("effectFile");
}
```

■ **Note** You'll have to change the name of the asset, `effectFile` in this case, to the name of your HLSL file.

1-3. Deploy Your XNA 3.0 Game on Xbox 360

The Problem

Once you have created and tested your code on the PC, you want to upload your game to and run it on your Xbox 360 console.

The Solution

One of the nicest features of XNA is that you can make your code run on both PCs and on Xbox 360, without having to change anything. There are a few prerequisites before you can upload your working code to Xbox 360 though.

First, you need to have an Xbox Live account, which can be created for free through the `http://creators.xna.com` site or on Xbox 360.

Next, you need a Creators Club license, which is free for most students or can be bought through the Xbox Live Marketplace. This license costs $49 USD for four months or $99 USD for one year. If you are a student, you might have access to a free license.

Next, you need to download and install XNA Game Studio Connect, the front-end program that listens for a connection from your PC.

Last but definitely not least, you need a LAN connection between your PC and Xbox 360, and the Xbox 360 should be connected to the Internet. The PC and Xbox 360 should also

be paired, because you would otherwise run into trouble when you have multiple Xbox 360 consoles in your network.

Once you have fulfilled these four prerequisites, you can upload and run your code on Xbox 360 from within XNA Game Studio 3.0 on your PC very easily.

How It Works

Setting Up the Xbox Live Account

Signing up for a Silver Xbox Live account is free and required if you want to run your own code on your Xbox 360 console. If you have already used your Xbox 360 console, you'll probably already have a Live account. If you haven't, start your Xbox 360 console, insert a game disc, and follow the instructions on your screen.

Obtaining the Creators Club License

If you are a student, chances are you can obtain a free license from the Microsoft DreamSpark program. You can access this from `http://downloads.channel8.msdn.com`. Log in with your student credentials to obtain a code, which you can enter by going to the Marketplace tab in your Xbox 360 dashboard and choosing "Redeem code."

Otherwise, you can simply log your Xbox 360 console on to the Xbox Live Marketplace and then navigate to Games ➤ All Game Downloads. In the list, find XNA Creators Club, and select it.

Then select Memberships, and you can buy a license for four months or for one year. Alternatively, you can also enter a code that you can find on a Creators Club voucher card.

Installing XNA Game Studio Connect on Your Xbox 360

This program makes your Xbox 360 listen for any incoming connections from your PC.

You can download this for free by going to the Xbox Live Marketplace and browsing to Game Store ➤ More ➤ Genres ➤ Other. Start the program after you've finished installing it.

Connecting Your Xbox 360 and PC

Before your PC can stream data to the Xbox 360, the two devices need to be connected by a LAN and to the Internet. If both your Xbox 360 and PC are attached to a router, switch, or hub, this should be OK.

Nowadays, more and more home networks are relying on a wireless network. This might be a problem, because the Xbox 360 doesn't ship with a wireless adapter by default. One solution is to have a PC with both a wireless and a wired (Ethernet) network, which is common for most new laptops. Connect the PC to your wireless network at home, and add a $5 patch cable between your Xbox 360 and PC. Finally, on your PC, click the Start button, and navigate to Settings ➤ Network Connections. Highlight both your wireless and Ethernet adapters, right-click one, and select Bridge Connections, as shown in Figure 1-2. Wait for a few minutes, and both machines should be connected to the Internet and to each other!

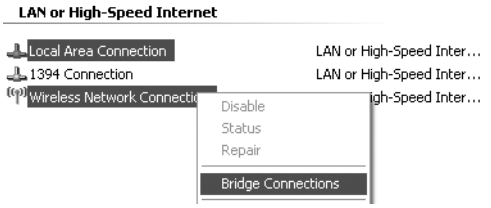

Figure 1-2. *Bridging two network adapters on one PC*

Pairing Your PC and Xbox 360

In case you have multiple Xbox 360 consoles in your network, you should specify which Xbox 360 you want to upload your code to. If you haven't done already, start XNA Game Studio Connect on your Xbox 360 by going to the Game tab and selecting Games Library ➤ My Games ➤ XNA Game Studio Connect. If this is the first time you've launched Connect, you will be presented with your Xbox 360's serial number, a series of five five-character strings.

On your PC, click the Start button, and navigate to Programs ➤ Microsoft XNA Game Studio 3.0 ➤ XNA Game Studio Device Center. Click the Add Device button, click the Xbox 360 icon, and give your Xbox 360 console a name of your choosing. Next, you are invited to enter the serial number shown by your Xbox 360. If both your Xbox 360 and PC are connected by the network, the pairing should succeed, and your console should appear in the device list. The green sign indicates your currently active Xbox 360, in case you have paired your PC to multiple Xbox 360 consoles.

Generating an Xbox 360 Project from an Existing XNA 3.0 Project

In XNA Game Studio 3.0, it's easy to convert your PC game to an Xbox 360 project. Simply open your project, and find the Solution Explorer at the top-right of your screen. Right-click your project's name, and select Create Copy of Project for Xbox 360, as shown in Figure 1-3.

This will result in a second project being created and added to your solution. All files of your original project will be referenced by the new project, not copied, so that any changes you make in a file in one project will be visible in the other project as well.

In some cases, you might need to add some references that the wizard has forgotten to copy, but all in all, the wizard will save you quite a bit of time.

From now on, you can select on which target you want to run your project at the top of your screen. If this is the first time you've run a project on Xbox 360, you need to add the Xbox 360 profile by selecting "Configuration Manager" at the top of your screen, as shown in Figure 1-4.

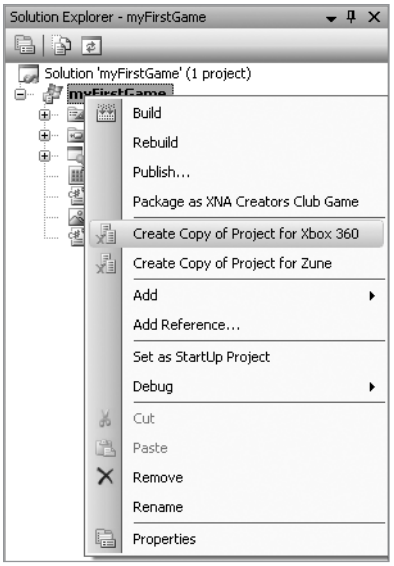

Figure 1-3. *Generating an Xbox 360 project*

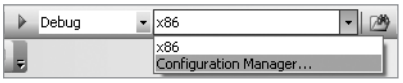

Figure 1-4. *Selecting the Configuration Manager*

In the dialog that appears, click the "Active solution platform" list and select <New...>. Next, select Xbox360, and close all dialogs.

From now on, the list shown in Figure 1-4 should contain an Xbox360 entry, which you should select whenever you want to deploy to your Xbox360 console. Make sure your Xbox is running XNA Game Studio Connect and is waiting for a connection. When you hit F5, your files will be uploaded to and executed on your Xbox 360 console!

1-4. Deploy Your XNA 3.0 Game on the Zune

The Problem

Once you have created and tested your 2D game on the PC, you want to upload your game to your Zune and run it.

The Solution

One of the main new features of XNA 3.0 is Zune support for 2D games. Since the Zune obviously can't compare to the power of the graphics card inside your computer, you cannot execute any XNA 3D application on the Zune. However, the complete `SpriteBatch` API is available on the Zune, allowing you to create mobile 2D games.

How It Works

As with the Xbox360 console, you don't have to make any adjustments to your code to make it run on the Zune. Just connect the Zune to your PC, set it as the active target, and deploy your game.

Connecting the Zune to Your PC

Connect the Zune to your PC using the USB cable that shipped with the Zune, and power it on. Next, open the XNA Game Studio Device Center by clicking the Start menu and selecting Programs ➤ Microsoft XNA Game Studio 3.0 ➤ XNA Game Studio Device Center. Click the Add Device button, and select the Zune. If you haven't done so yet, you might be asked to download an up-to-date version of the Zune software by following the provided link.

If you have installed the software, make sure the firmware on your Zune is up to date. You can do so by connecting your Zune and opening up the Zune software on your PC. If your PC is connected to the Internet and a newer version of the firmware is found, the Zune software will show the Device Update screen, allowing you to update your firmware by clicking the Install button (make sure you never power off any device while you are reprogramming its firmware).

When you have installed the Zune software, updated the Zune firmware, and connected the Zune, your Zune should be listed on your screen when you click the Zune button in the XNA Game Studio Device Center. Select your Zune, and click Next. After the connection has been verified, your Zune should be listed in the XNA Game Studio Device Center main screen.

Setting the Zune As a Target Platform in XNA 3.0 Game Studio

With the Zune connected, go to XNA 3.0 Game Studio to deploy your 2D game to the Zune. As with the Xbox360 console, the first time you deploy to the Zune, you need to set it as a deployment target. This can be done by clicking the Solution Platforms box in the top-center of the screen and selecting Configuration Manager, as shown in Figure 1-4.

In the upcoming dialog, select <New…> from the Active solution configuration list at the top-left of the screen, and select your Zune from the top list in the following screen. After clicking the OK button a couple of times, you should see your Zune as the target platform at the top-center of your window.

Deploying Your XNA 3.0 Game to the Zune

Finally, you need to duplicate your XNA PC project to an XNA Zune project. This is done very easily, by right-clicking your project in the Solution Explorer and selecting Create Copy of Project for Zune. This option is also shown in the list of Figure 1-3. XNA will add this project to your Solution Explorer and will sync both projects whenever you change either of them.

To upload your game to the Zune, make sure the Zune is selected as the active target platform at the top-center of your screen and that the Zune software is not running on your PC, and hit F5 to initiate the deployment.

Once your Zune screen indicates the deployment has been completed, restart your Zune, select Games in your Zune's main menu, and start your game!

1-5. Deploy Your XNA 3.0 Game on Another PC

The Problem

You have finished the first version of your game and want to show it off on a friend's PC. However, when you double-click the `.exe` file, you get some errors.

The Solution

Distributing XNA games to different PCs has become easier than it was in version 2.0. At the time of this writing, you still need to make sure two other packages are already installed on the other PC before you can feel safe running your game's executable.

The two packages that need to be installed first:

- The XNA Framework Redistributable 3.0
- The .NET 3.5 Framework

The good news is that XNA is now compatible with Visual Studio's ClickOnce technology, allowing you to create a single `.exe` file that first checks whether these packages are there and to install the packages before installing your game if they're absent. Nevertheless, if you want to use the networking functionality in XNA 3.0 (see Chapter 8), you still have to install the complete XNA Game Studio 3.0 package on the destination PC.

How It Works

Installing the XNA 3.0 Framework Files

The first solution is downloading and installing the two packages on the PC, together with your game's binary files. You can solve this by downloading and installing the XNA Framework Redistributable 3.0, which you can find by searching for it on the Microsoft site at `www.microsoft.com`. The package is very small and contains all the basic XNA 3.0 Framework files.

Installing the .NET 3.5 Framework Files

XNA is the new managed wrapper around DirectX. Because it uses a managed .NET language (C#), you'll also need to make sure the .NET Framework 3.5 files are present on the system. You can download this package from the Microsoft site by searching for *.NET 3.5*.

Copying Binary Files

After you've compiled and tested your game, go to the executable directory, which can be any combination between `bin\x86\Debug` and `bin\x64\Release`. Make sure you copy all files and sub-maps you find that map to the destination PC. If you installed both packages, you shouldn't receive any error messages when you double-click the `.exe` file to start your game.

Creating a Single ClickOnce Installation Package Containing All Prerequisites

New to XNA 3.0, ClickOnce installation lets you create an `.exe` install file that installs both prerequisite packages. To create the install file, follow these steps:

1. In the Solution Explorer, right-click your project, and select Properties.
2. In the window that opens, select Publish at the bottom-left.
3. Click the Prerequisites button on the next screen.
4. You'll see a list of all Microsoft packages that can be packed with your game. Make sure you select .NET Framework 3.5 (SP1) and Microsoft XNA Framework Redistributable 3.0.
5. Most importantly, select "Download prerequisites from the same location as my application." This will cause the packages to be copied to the map where your project is published.
6. Hit OK, and close the Properties window of your project.

With all of this done, whenever you feel the need to publish your project, just right-click your project, select Publish, and hit Finish in the dialog that appears.

■ **Note** Unfortunately, there currently is a bug in Visual Studio that causes problems when publishing your project this way, which will hopefully be fixed by a next update or service pack. Therefore, for now, you should deselect the .NET Framework 3.5 (SP1) prerequisite in step 4, and include this as an additional installation file.

Creating a Setup Project for Your Game

Using this slightly more advanced approach, you can create a more customizable setup application that automatically detects the presence of both prerequisite packages and, if necessary, installs them.

■ **Note** This approach doesn't suffer from the bug mentioned in the ClickOnce approach.

To create a setup project, follow these steps:

1. Open XNA Game Studio 3.0.
2. Open the File menu, and select New ➤ Project.
3. In the tree view on the left, select Other Project Types ➤ Setup and Deployment.
4. On the right side, make sure Setup Project is selected.

5. Give a name to this project, such as **[name]**. During setup execution, this will be used as in "Welcome to the [name] Setup Wizard."
6. Click OK to start the new setup project.
7. Find the Applications Folder entry on the left side of the screen.
8. In Windows Explorer, browse to the output directory of your XNA game, which contains the `.exe` file of your game. Select all of the files and submaps (such as Content), and drag them onto the Applications Folder of step 7.
9. Back in XNA Game Studio 3.0, open the Project menu, and select "[name] properties."
10. Click the Prerequisites button.
11. Make sure Windows Installer 3.1, .NET Framework 3.5, and Microsoft XNA Framework Redistributable 3.0 are selected.
12. At the bottom of the dialog, select "Download prerequisites" from the same location as "my application."
13. Click OK twice to return to the setup project.
14. Click Application Folders.
15. In the Properties box at the bottom-right of the screen, remove [Manufacturer] from the target directory.
16. Now, press F6 to build the setup project solution.

This procedure will generate a map containing all the prerequisites and a simple `setup.exe`, which installs whatever necessary to run your XNA game on another PC.

1-6. Customize Game Loop Timing

The Problem

You want to change the default timing intervals at which the `Update` and `Draw` methods are called.

The Solution

By default, the `Update` method is called exactly 60 times each second, while the `Draw` method is called as often as possible, with the refresh rate of your screen at maximum.

By changing the values of the `TargetElapsedTime` and `IsFixedTimeStep` static properties of the `Game` class and the `SynchronizeWithVerticalRetrace` property of the `GraphicsDevice` class, you can change this default behavior.

How It Works

Changing the Update Frequency

By default, the `Update` method of your `Game` is called exactly 60 times each second, or once every 16.667 milliseconds. You can change this by adjusting the value of the `TargetElapsedTime` variable:

```
this.TargetElapsedTime = TimeSpan.FromSeconds(1.0f / 100.0f);
```

When you call this line of code, XNA will make sure the `Update` method will be called 100 times per second.

You can also instruct XNA to call the `Update` method not at regular intervals but instead each time before the `Draw` method is called. You do this by setting the `IsFixedTimeStep` variable to `false`:

```
this.IsFixedTimeStep = false;
```

Using IsRunningSlowly

You can specify an `Update` frequency of your choice. However, when the number you specified is too high, XNA will not be able to call your `Update` method at that frequency. If this is the case, the `gameTime.IsRunningSlowly` variable will be set to `true` by XNA:

```
Window.Title = gameTime.IsRunningSlowly.ToString();
```

■ **Note** You should check the `gameTime` argument passed to the `Update` method, and not the `gameTime` argument passed to the `Draw` method, to verify this.

Changing the Draw Frequency

While running the game, XNA will call the `Draw` method as frequently as possible, limited by these two rules:

- There's no use in calling the `Draw` method more frequently than the screen refresh rate. If a screen refreshes only 100 times per second, rendering 110 frames per second won't be helpful. On the PC and Xbox 360 console, screen refresh rates are determined by the PC screen and its settings. The Zune 30 refreshes 60 times per second; the other Zune devices show only 30 frames per second.
- The `Update` method should be called 60 times per second. If your game is too calculation intensive, the `Draw` method will be called less frequently to ensure XNA can call the `Update` method 60 times per second.

In some cases, it can be useful to call your `Draw` method at maximum frequency, for example, to determine the maximum frame rate of your `Game`. You can do this by setting the `graphics.SynchronizeWithVerticalRetrace` variable to `false`:

```
graphics.SynchronizeWithVerticalRetrace = false;
```

■ **Note** You must put this line in the `Game1` constructor at the top of your code, because XNA needs to be aware of this before it creates `GraphicsDevice`.

Understanding the Importance of the Update and Draw Frequencies

Since you'll put your update login in the `Update` method, a (temporal) decrease of the `Update` frequency would make all objects in your game move more slowly, which is very annoying.

When the `Draw` frequency is called less than the screen refresh rate, only the visual appearance of your game will suffer for a moment. It's less annoying that your game temporally only renders 80 instead of 100 frames per second to the screen.

Therefore, and as mentioned previously, if required, XNA will decrease the `Draw` frequency to ensure the `Update` method is called exactly 60 times each second.

1-7. Make Your Code Plug-and-Play Using GameComponents

The Problem

You want to separate part of your application into a `GameComponent` class. This will ensure reusability of the component in other applications.

The Solution

Certain parts of your applications can be separated from the rest of your application. In an XNA application, most such parts need to be updated or drawn, such as a particle or billboard system (see recipes 3-11 and 3-12).

One step in the correct direction is to create a separate class for such a part. In your main XNA `Game` class, you will then need to create an instance of this class, initialize it, update it from within the `Update` method, and, if applicable, render it to the screen from within the `Draw` method. Therefore, you will want your new class to have its own `Initialize`, `(Un)LoadContent`, `Update`, and `Draw` methods so you can easily call them from within your main XNA `Game` class.

If you find yourself defining these methods in your new class, it might be a nice idea to make your new class inherit from the `GameComponent` class. If you do this, you can add it to the `Components` list of your `Game`. This will cause the `Initialize` method of your newly defined class to be called after the `Initialize` class of your main `Game` class finishes. Furthermore, each time the `Update` method of your main `Game` class finishes, the `Update` method of your newly defined `GameComponent` class will be called automatically.

If your component should also render something, you should inherit from the `DrawableGameComponent` class instead of from the `GameComponent` class. This will expect your component to also contain a `Draw` method, which will be called after the `Draw` method of your main `Game` class finishes.

■ **Note** At the end of the `Initialize` method in your main `Game` class, you'll notice the call to `base.Initialize`. It is this line that starts calling the `Initialize` methods of all the `GameComponent` classes of your `Game` class. You can find the same kind of call at the end of the other methods in your main `Game` class, which will call the corresponding methods of all `GameComponent` classes currently registered in your `Game`.

How It Works

As an example, the billboarding code of recipe 3-11 will be separated into a `GameComponent` class. Even better, because this code also needs to render something to the screen, you will make it a `DrawableGameComponent` class.

Creating a New (Drawable)GameComponent

Add a new class file to your project by right-clicking your project and selecting Add ➤ New File. Select Class in the upcoming dialog box; I called my new class `BillboardGC`. In the new file that is presented to you, you'll want to add the XNA `using` lines so you have access to all XNA functionality in your new class, which can be done very easily by copying the `using` block of your main `Game` class into the new class.

Next, make sure you make your new class inherit from the `GameComponent` class or the `DrawableGameComponent` class, as shown in the first line of the following code snippet. Add all the code of the component, and separate it nicely between the `Initialize`, `(Un)LoadContent`, `Update`, and `Draw` methods of your new class.

The following example shows how this can be done for the billboarding code of recipe 3-11. Some methods such as `CreateBBVertices` have not been fully listed, because in this recipe you should focus on the `Initialize`, `LoadContent`, `Update`, and `Draw` methods.

```
class BillboardGC : DrawableGameComponent
{
    private GraphicsDevice device;

    private BasicEffect basicEffect;
    private Texture2D myTexture;
    private VertexPositionTexture[] billboardVertices;
    private VertexDeclaration myVertexDeclaration;
    private List<Vector4> billboardList = new List<Vector4>();

    public Vector3 camPosition;
    public Vector3 camForward;
    public Matrix viewMatrix;
    public Matrix projectionMatrix;
```

```
    public BillboardGC(Game game) : base(game)
    {
    }

    public override void Initialize()
    {
        device = Game.GraphicsDevice;
        base.Initialize();
    }

    protected override void LoadContent()
    {
        basicEffect = new BasicEffect(device, null);
        myTexture = Game.Content.Load<Texture2D>("billboardtexture");
        AddBillboards();
        myVertexDeclaration = new VertexDeclaration(device, ➥
        VertexPositionTexture.VertexElements);
    }

    public override void Update(GameTime gameTime)
    {
        CreateBBVertices();
        base.Update(gameTime);
    }

    .
    .
    .

    public override void Draw(GameTime gameTime)
    {
        //draw billboards
        .
        .
        .
    }
}
```

■ **Note** As you can see in the `Initialize` method, your component can access the main `Game` class. This allows your component to access the public fields of the main `Game` class, such as `Game.GraphicsDevice` and `Game.Content`.

Using Your New GameComponent

Now that you have defined your GameComponent, you should add it to the list of GameComponent classes of your main Game class. Once added, its main methods will automatically be called.

The easiest way to do this is to create a new instance of your GameComponent and add it immediately to the Components list. An ideal place to do this is in the constructor of your main Game class:

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

    Components.Add(new BillboardGC(this));
}
```

This will cause the Initialize and LoadContent methods to be called at startup and the Update and Draw methods of the new class to be called each time the Update and Draw methods of your main Game class have finished.

In some cases, you will need to update some public variables of the component. In the case of the billboarding component, you'll need to update the camPosition and camForward variables so the component can adjust its billboards and the View and Projection matrices so they can be rendered correctly to the screen. Therefore, you'll want to keep a link to your component by adding this variable to your main Game class:

```
BillboardGC billboardGC;
```

Then store a link to your component before storing it in the Components list:

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

    billboardGC = new BillboardGC(this);
    Components.Add(billboardGC);
}
```

Now in the Update method of your main Game class, you can update these four variables inside your component. At the end of the Update method of your main Game class, the Update method of all components is called, allowing the billboarding component to update its billboards:

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

```
    billboardGC.camForward = quatCam.Forward;
    billboardGC.camPosition = quatCam.Position;
    billboardGC.viewMatrix = quatCam.ViewMatrix;
    billboardGC.projectionMatrix = quatCam.ProjectionMatrix;

    base.Update(gameTime);
}
```

The `Draw` method of your main `Game` class is even simpler: just clear the screen before calling the `Draw` method of all the components:

```
protected override void Draw(GameTime gameTime)
{
    device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, ➥
    Color.CornflowerBlue, 1, 0);
    base.Draw(gameTime);
}
```

The last line will cause the `Draw` method of the billboarding component to be called, rendering the billboards to the screen.

■ **Tip** The next recipe shows a much cleaner approach to update the variables of your `GameComponents`.

The Code

All the code is available for download at `www.apress.com`.

The `Initialize`, `LoadContent`, `Update`, and `Draw` methods of the `GameComponent` and main `Game` class were listed earlier in the text.

1-8. Allow Your GameComponents to Communicate with Each Other by Implementing GameServices

The Problem

As explained in recipe 1-6, you can separate parts of your code into reusable `GameComponent` classes. Examples of such components can be a camera, particle system, user input processing, billboard engine, and more.

One of the main benefits of using `GameComponent` classes is that you can easily switch between game functionalities, for example, camera modes. Changing from a first-person camera to a quaternion camera (see recipe 2-4) involves changing just one line of code in the `Initialize` method of your main `Game` class.

Using GameComponent classes to achieve this is one thing, but you need to make sure you don't have to change the rest of your code (which uses the camera) when you switch from one component to another.

This recipe uses some more advanced .NET and object-oriented functionality.

The Solution

You will make both your camera components subscribe to the same interface, such as the (self-defined) ICameraInterface interface. When you initialize your camera component, you let your Game class know that from now on Game contains a component that implements the ICameraInterface interface. In XNA words, the component registers itself as the GameService of the ICameraInterface type.

Once this has been done, the rest of your code can simply ask the Game class for the current ICameraInterface service. The Game class will return the camera that is currently providing the ICameraInterface service. This means your calling code never needs to know whether it is actually a first-person camera or a quaternion camera.

How It Works

An *interface* is some kind of contract that you make your class (your GameComponent, in this case) sign. An interface contains a list of functionality (nonconcrete methods and/or properties, actually) that the class should minimally support. When your class is subscribed to an interface, it promises that it implements the methods listed in the definition of the interface.

This is how you define the ICameraInterface interface used in this recipe:

```
interface ICameraInterface
{
    Vector3 Position { get;}
    Vector3 Forward { get;}
    Vector3 UpVector { get;}

    Matrix ViewMatrix { get;}
    Matrix ProjectionMatrix { get;}
}
```

■ **Note** The methods used here are getter methods, introduced in C# 2.0. They are methods, but act to the outside world as read-only public properties. For example, the Position method should return a Vector3 object.

Any class wanting to subscribe to ICameraInterface should implement these five getter methods.

For the rest of your code, it isn't of any importance to know whether the current camera is a first-person or quaternion camera. The only thing that matters is that the camera can

produce valid `View` and `Projection` matrices and maybe some other directional vectors. So, it suffices for your main `Game` class to have a camera class that subscribes to `ICameraInterface`.

Making Your GameComponent Subscribe to an Interface

In this example, you will have two camera components. Because they don't need to draw anything to the screen, a `DrawableGameComponent` is not necessary, so inherit from the `GameComponent` class. Also, make your component subscribe to `ICameraInterface`:

```
class QuakeCameraGC : GameComponent, ICameraInterface
{
    .
    .
    .
}
```

■ **Note** Although a class can inherit from only one parental class (the `GameComponent` class, in this case), it can subscribe to multiple interfaces.

Next, you need to make sure your class actually lives up to its contract by implementing the methods described in the interface. In the case of the `QuakeCamera` and `Quaternion` classes described in recipe 2-3 and recipe 2-4, this is already the case. See the accompanying code for this recipe for the minor changes in turning them into two `GameComponent` classes.

Subscribing to the ICameraInterface Service

In your `Game` class, you should have one and only one camera component that provides the `ICameraInterface` service at a time. When you activate a camera component, it should let your main `Game` class know it is the current implementation of `ICameraInterface`, so your main `Game` class knows it should pass this camera to the rest of the code in case it is asked for the current provider of `ICameraInterface`.

You do this by registering it as a `GameService` in the `Services` collection of the main `Game` class:

```
public QuakeCameraGC(Game game) : base(game)
{
    game.Services.AddService(typeof(ICameraInterface), this);
}
```

You add the `this` object (the newly created first-person camera component) to the list of interfaces, and you indicate it provides the `ICameraInterface` service.

Usage

Whenever your main `Game` code needs the current camera (for example, to retrieve the `View` and `Projection` matrices), you should ask the main `Game` class to give you the current implementation of `ICameraInterface`. On the object that is returned, you can access all the fields defined in the `ICameraInterface` definition.

```
protected override void Draw(GameTime gameTime)
{
    device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, ➥
    Color.CornflowerBlue, 1, 0);

    ICameraInterface camera;
    camera = (ICameraInterface)Services.GetService(typeof(ICameraInterface));

    cCross.Draw(camera.ViewMatrix, camera.ProjectionMatrix);

    base.Draw(gameTime);
}
```

You ask your main `Game` class to return the class that provides the `ICameraInterface` interface. Since the `GetService` method is capable of returning any type of object, you need to help the compiler a bit by specifying the type of interface the object should be cast to.

Note that the code inside this `Draw` method never knows whether it is talking to a first-person camera or a quaternion camera. All that matters is that it supports the functionality listed in the `ICameraInterface` definition. The beauty of this approach is that you can easily swap between camera modes—somewhere in the `Update` method, for example—with the rest of your code not even noticing this change.

■ **Note** You can also define a camera as a global variable and store a link to the currently active implementation of `ICameraInterface` in the `Initialize` method of the main `Game` class.

Using Multiple GameComponents

`GameServices` are especially useful to ensure interoperability between multiple `GameComponent` classes. In case you have a camera `GameComponent` providing the `ICameraInterface` service, other `GameComponent` classes can access this by querying the `ICameraInterface` service from the main `Game` class.

This means you don't have to provide any hard links between the different components, such as the ugly main `Update` method of the previous recipe. In case you have created a camera `GameComponent` that supplies the `ICameraInterface` service, you can query this service from any other `GameComponent`, such as from the `Initialize` method of the billboard `GameComponent` created in recipe 1-6:

```
public override void Initialize()
{
    device = Game.GraphicsDevice;
    camera = (ICameraInterface)Game.Services.GetService(typeof(ICameraInterface));

    base.Initialize();
}
```

Next, the `Update` and `Draw` methods of the billboarding component can access the required fields from the camera. All your main `Game` needs to do is instantiate the camera and billboarding components. The camera will subscribe itself to the `ICameraInterface` service, allowing the billboarding component to retrieve the camera.

The camera will automatically be called to update itself, after which the billboarding component will be asked to do the same. The billboarding component is able to access the camera through the `ICameraInterface` service, and finally the billboarding component is kindly asked to draw itself. This whole process requires two lines of code in your main `Game` class, plus you can easily swap between camera modes!

Changing the Updating Order of Your GameComponents

If you have a component that requires the output of another component, such as in this case where the billboarding component needs your camera component, you may want to be able to specify that the camera component should be updated first. You can do this before adding the component to the `Components` list of the main `Game` class:

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

    //GameComponent camComponent = new QuakeCameraGC(this);
    GameComponent camComponent = new QuatCamGC(this);
    GameComponent billComponent = new BillboardGC(this);

    camComponent.UpdateOrder = 0;
    billComponent.UpdateOrder = 1;

    Components.Add(camComponent);
    Components.Add(billComponent);
}
```

First, you create your camera and billboarding components. Before adding them to the `Components` list, you set their updating order, with a lower number indicating the component that should be updated first.

Using `GameComponent` classes and `GameServices`, in order to switch between two camera modes, you need to change only the line that adds the camera component. In the previous code, the quaternion mode is activated. If you want to switch to `Quake` mode, simply uncomment the line that adds the `QuakeCamera GameComponent` class, and comment the

line that adds the QuatCam GameComponent class. The remainder of the code doesn't need to be changed, because all it needs is the ICameraInterface service, provided by one of the GameComponent classes.

The Code

All the code is available for download at www.apress.com.

The Game1 constructor displayed previously is all the code needed to get your camera and billboarding working. The Initialize, (Un)LoadContent, Update, and Draw methods of your main Game class are empty.

When your camera component is created, it subscribes itself to the ICameraInterface service:

```
public QuakeCameraGC(Game game) : base(game)
{
    game.Services.AddService(typeof(ICameraInterface), this);
}
```

Whenever any code of your project requires the camera, you can ask your main Game to return the service that implements ICameraInterface:

```
ICameraInterface camera;
camera = (ICameraInterface)Services.GetService(typeof(ICameraInterface));
cCross.Draw(camera.ViewMatrix, camera.ProjectionMatrix);
```

1-9. Save and Load Data to or from a File

The Problem

You need some kind of saving mechanism for your game.

The Solution

Although saving or loading data is usually one of the last steps involved in creating a game, you definitely want to have some sort of saving mechanism for your game. Basically, XNA uses the default .NET file I/O mechanisms, meaning you can create/open/delete files quite easily. Furthermore, using the XmlSerializer class, it is incredibly easy to save your data to disk and to reload it afterward.

The only problem that needs to be tackled is finding a location to save the data to disk that is valid for PCs and for the Xbox 360 and Zune. You can solve this problem by using a StorageDevice, which needs to be created first.

■ **Note** The concept of creating a StorageDevice might seem complex. Don't let that turn you away, though, because the rest of the recipe (starting from "Saving Data to Disk") is very simple and powerful and is not restricted only to XNA, because it is default .NET functionality.

How It Works

Before you can save data to disk, you'll need a valid location to which you have access to write. The solution is provided by creating a `StorageDevice`, which asks the user on the Xbox 360 console where to store the data. However, you need to make sure you're not making multiple save/load calls to the `StorageDevice` at the same time, so you'll need to keep track of its activity:

```
bool operationPending = false;
```

Creating a StorageDevice Asynchronously

This process involves opening the Xbox `Guide`, which would block your whole program until the user closes the `Guide`. To solve this, this process has been made asynchronous. This concept is explained in recipe 8-5.

The `Guide` requires the `GamerServicesComponent` to be added to your game (see recipe 8-1), so add the following line to your `Game1` constructor:

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

    Components.Add(new GamerServicesComponent(this));
}
```

Next, each time the `Update` method is called, you will check whether the user wants to save data. If this is the case, you will call `Guide.BeginShowStorageDeviceSelector`, which will open a dialog box so the user can select where the data should be saved. The program doesn't halt at this line, however, because this would stall your whole program until the user closes the dialog box. Instead, this method expects you to pass as the first argument another method, which should be called once the user exits the dialog box:

```
KeyboardState keyState = Keyboard.GetState();
if (!Guide.IsVisible && !operationPending)
    if (keyState.IsKeyDown(Keys.S))
    {
        operationPending = true;
        Guide.BeginShowStorageDeviceSelector(FindStorageDevice, "saveRequest");
    }
```

If the user presses the S button, the dialog box will open, but the code will continue immediately without waiting for the user's response. Notice that you've set the `operationPending` variable to `true`, so the first `if` check of the preceding code will be evaluated negatively until you've set it to false when the operation eventually completes (see the next code snippet).

Your program will continue to run while the dialog box is being displayed on the screen. Once the user has made a selection and closed the dialog box, the `FindStorageDevice` method you specified as the first argument will be called. This means you'll have to define this method, or your compiler will complain:

```
private void FindStorageDevice(IAsyncResult result)
{
    StorageDevice storageDevice = Guide.EndShowStorageDeviceSelector(result);
    if (storageDevice != null)
        SaveGame(storageDevice);
}
```

The result of `BeginShowStorageDeviceSelector` is received as the argument by this method. If you pass this result to the `Guide.EndShowStorageDeviceSelector` method, you will get the selected data storage. However, if the user canceled the operation, the result will be null, so you have to check for this. If the resulting `StorageDevice` is valid, you pass this to the `SaveGame` method, which you'll define in a moment.

But first, imagine what would happen if you also allowed the user to perform a second operation for which the user needs to specify a data location, such as when loading data. This would require you to define a second method, `FindStorageDeviceForLoading`, for example. A cleaner way would be to specify an identifier to your asynchronous call, which you can check for in the `FindStorageDevice` method. Your `Update` method would contain this block of code:

```
KeyboardState keyState = Keyboard.GetState();
if (!Guide.IsVisible && !operationPending)
{
    if (keyState.IsKeyDown(Keys.S))
    {
        operationPending = true;
        Guide.BeginShowStorageDeviceSelector(FindStorageDevice, "saveRequest");
    }
    if (keyState.IsKeyDown(Keys.L))
    {
        operationPending = true;
        Guide.BeginShowStorageDeviceSelector(FindStorageDevice, "loadRequest");
    }
}
```

As you can see, in both cases, a dialog box will be displayed, which will call the `FindStorageDevice` method after it closes. The difference is that this time you're specifying an identifier, which you can check for in the `FindStorageDevice` method:

```
private void FindStorageDevice(IAsyncResult result)
{
    StorageDevice storageDevice = Guide.EndShowStorageDeviceSelector(result);
    if (storageDevice != null)
    {
        if (result.AsyncState == "saveRequest")
            SaveGame(storageDevice);
        else if (result.AsyncState == "loadRequest")
            LoadGame(storageDevice);
    }
}
```

Depending on the identity of the call, you will call the `SaveGame` or `LoadGame` method.

Saving Data to Disk

Once you have a valid StorageDevice, you can easily specify a name for the data file that will be written to disk:

```
private void SaveGame(StorageDevice storageDevice)
{
    StorageContainer container = storageDevice.OpenContainer("BookCodeWin");
    string fileName = Path.Combine(container.Path, "save0001.sav");

    FileStream saveFile = File.Open(fileName, FileMode.Create);
}
```

This will create a file called save0001.sav. If it exists, it will be overwritten.

■ **Note** On a PC, this file will be created in a map located in the My Documents\SavedGames folder.

Once you have a valid file name and have opened the file, you can save your file using default .NET functionality. Imagine the data you would want to save looks like this:

```
public struct GameData
{
    public int ActivePlayers;
    public float Time;
}
```

All you need to do is create an XmlSerializer, which is capable of converting your data into XML and saving this to disk:

```
XmlSerializer xmlSerializer = new XmlSerializer(typeof(GameData));
xmlSerializer.Serialize(saveFile, gameData);
saveFile.Close();
container.Dispose();
operationPending = false;
```

You indicate the XmlSerializer should be capable of serializing GameData objects, after which you stream your GameData object to file with a single command! Don't forget to close the file stream and container, or your program will keep them locked. Also reset the operationPending variable to false, so your Update method will listen for any new save or load requests from the user.

For this to work, you need to link to the System.IO and System.Xml.Serialization namespaces, which can be done easily by adding these lines to your using block at the very top of your code:

```
using System.IO;
using System.Xml.Serialization;
```

The last line requires you to add a reference to `System.Xml`, which can be done by opening the Project menu and selecting Add Reference. Highlight System.Xml, as shown in Figure 1-5, and hit OK.

Figure 1-5. *Adding a reference to System.Xml*

Loading Data from Disk

You load data from disk in the same way as you save data to disk, just a bit opposite. You check whether the file exists, and if it does, you open it. Once again, you create an `XmlSerializer`, but this time around you use it to deserialize a `GameData` object from the file stream. This single line loads all data from the file and transforms this data into a valid `GameData` object!

```
private void LoadGame(StorageDevice storageDevice)
{
    StorageContainer container = storageDevice.OpenContainer("BookCodeWin");
    string fileName = Path.Combine(container.Path, "save0001.sav");
    if (File.Exists(fileName))
    {
        FileStream saveFile = File.Open(fileName, FileMode.Open);
        XmlSerializer xmlSerializer = new XmlSerializer(typeof(GameData));

        gameData = (GameData)xmlSerializer.Deserialize(saveFile);
        saveFile.Close();
        operationPending = false;
    }
    container.Dispose();
}
```

The Code

All the code is available for download at `www.apress.com`.

The `Update` method simply checks whether the user wants to save or load a file and opens the `Guide` dialog box:

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

```
    KeyboardState keyState = Keyboard.GetState();
    if (!Guide.IsVisible && !operationPending)
    {
        if (keyState.IsKeyDown(Keys.S))
        {
            operationPending = true;
            Guide.BeginShowStorageDeviceSelector(FindStorageDevice, "saveRequest");
        }
        if (keyState.IsKeyDown(Keys.L))
        {
            operationPending = true;
            Guide.BeginShowStorageDeviceSelector(FindStorageDevice, "loadRequest");
        }
    }

    gameData.Time += (float)gameTime.ElapsedGameTime.TotalSeconds;

    base.Update(gameTime);
}
```

Once the `Guide` has been closed by the user, the `FindStorageDevice` method is called. This method in return calls the `SaveData` or `LoadData` method, depending on the identity of the asynchronous call. You can find this `FindStorageDevice` method entirely in the previous code; only the `SaveGame` method has not yet been listed in full:

```
private void SaveGame(StorageDevice storageDevice)
{
    StorageContainer container = storageDevice.OpenContainer("BookCodeWin");
    string fileName = Path.Combine(container.Path, "save0001.sav");

    FileStream saveFile = File.Open(fileName, FileMode.Create);
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(GameData));

    xmlSerializer.Serialize(saveFile, gameData);
    saveFile.Close();
    container.Dispose();
    operationPending = false;

    log.Add("Game data saved!");
}
```