Beginning XNA 2.0 Game Programming: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-924-2

ISBN-10 (pbk): 1-59059-924-1

ISBN-13 (electronic): 978-1-4302-0512-8

ISBN-10 (electronic): 1-4302-0512-1

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

Lead Editor: Ewan Buckingham
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Kylie Johnston
Copy Editor: Susannah Davidson Pfalzer
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Becky Hornyak
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.

CHAPTER 1

Game Planning and Programming Basics

In this chapter we present some fundamental concepts of planning and programming games that you must keep in mind when creating games. We won't discuss everything about planning a game, or present all general concepts regarding game programming. However, after reading this chapter you'll understand the basic ideas behind creating a game and how XNA makes game development easy for you.

Planning the Game

The effort involved in creating a good game starts way before the coding phase. Somewhat undervalued by nonprofessional game programmers, the planning is the most important phase of your game development project. That's because in this phase you define the guidelines for all the next stages.

Before thinking about which game you'll create, you have to choose your *target market*, because this choice will define the direction for your entire game development effort.

NPD Group, in its research, divides the market into six categories: heavy gamers (who constantly play games and are responsible for most of the market sales), avid console gamers (who buy mainly console games, and might play console games many hours a day), mass market gamers (who usually only buy "blockbuster" games), prefer portable gamers (as the category name says, they prefer playing games using portable devices), secondary gamers (who usually don't buy games, and play games bought by other people), and infrequent gamers (who play games every so often).

We won't provide an extensive study of these segments, but let's highlight some significant points about the two "edge" categories:

- *Infrequent gamers*: These gamers are also called "casual players." Games for this player category must be easy to play, with no complex storyline, and must provide challenging but brief levels, to give the player a feeling of accomplishment in short matches. Games for such a market usually don't rely on highly detailed 3-D graphics

or extraordinary sound effects, and include card games (poker, hearts, solitaire, and so on), puzzles (Tetris, Sudoku, crosswords, and so on), board games (mah-jongg, chess, checkers, and so on), and similar. Don't be fooled about such games: although they might be easier to develop, they rely on balanced levels and game play to sustain the appeal for the players, which can be hard to achieve.

- *Heavy gamers*: This group, also called "hardcore gamers," takes playing games seriously. They are usually moved by difficult challenges and a good storyline that helps the players immerse themselves in the game world. Games for such players usually include extremely detailed 3-D environments, engaging background music and sound effects, and a long game play with many challenges.

Once you choose the target market, the next logical step is to define the game genre. There are many divisions of game genres, but sticking with NPD Group's research approach, the best-selling game genres are presented in Table 1-1.

Table 1-1. *Best-Selling Computer and Video Games by Units Sold*

Computer Game	Console Game	Genre
30.8%		Strategy
19.8%	9.3%	Children & Family Entertainment
14.4%	8.7%	Shooter
12.4%	7.8%	Role-Playing
5.8%		Adventure
4.7%	30.1%	Action
3.7%	17.3%	Sports
	11.1%	Racing
	4.7%	Fighting
8.4%	11%	Others

Copyright NPD Group, 2005

■ **Note** A detail worth noting in Table 1-1 is that the best-selling game genres are fairly different from computer and console games. For instance, the best-selling genre in computer games—strategy—is accountable for almost one-third of sales in this segment, but it's not even among the seven top-selling genres in console games. Fortunately, with XNA you aren't forced to choose one platform: you can create a game that will run with minimal adjustments on both Xbox 360 and computers.

Choosing the target market and the game genre for your game will help you to narrow down your choices about which game to develop. And, if you already have a game in mind, thinking about these points will help you to refine your ideas to the next step: defining the team involved in the game development project, and choosing your place in such a team. This leads us to a second important game development concept: *the game team*.

Smaller teams, or even a single multiskilled person, might create games for casual players, while creating games for hardcore players might involve a team with dozens of people skilled in different areas.

Although you might be able to develop games on your own, developing a game is always more than simply coding. You'll need nice graphics and sound effects, and you'll need to design the game levels, just to name a few different activities in the game project. In a big game development project, you'll need skills such as the following:

- *Project management*: Someone must be in charge of controlling time, scope, resources needed, communications, coordination between team members, and so on. Even if you're developing a game with a few friends of yours, it's crucial to define "who's in charge" to solve problems and define the project direction.

- *Script writers*: The script writers are responsible for writing the game storyline, ultimately defining the challenges to face and the mysteries to solve. They usually help define the whole game background, such as the game characters, the dialogue, and the level division.

- *Level designers*: Level designers usually create and use tools to define each of the game levels, according to the programming premises given by the coding team and the story written by the script writers.

- *Artists*: "Artists" is a broad category, encompassing concept art creators, computer art creators, the people responsible for texturing (creating textures for the 3-D models), computer colorists, and so on. These folks create the "splash" or opening game screen and the game menus and static images, and might also create the art for the marketing team.

- *Modelers*: These people are responsible for creating the 3-D models for the game, following the concept and computer art.

- *Animators*: Creating a 3-D model is not the same thing as animating it, so some teams include specialists in creating the model animations for the game. This team also creates the *cut-scenes*—the video sequences presented in the beginning of the game and at special points in the game, such as when a player wins a challenge, or at the beginning or ending of each level.

- *Musicians*: This is also a broad category, which ranges from the responsibility for writing (and playing) the game background and ambience music to the people who create voices and sound effects for the game.

- *Programmers*: Someone must be in charge of writing the game code, including all math and physics calculations needed to meet the desired game effects. This book is intended as the first step into this category.

- *Testers*: It's not advisable that the same person who writes the code be accountable for testing it. The goal for the testers is to find as many bugs as they can, trying to do unexpected things inside the game so the bugs surface in the game development process instead of during the player's game.

This list goes on, including people who are responsible for preparing and conducting the marketing efforts for the game, people who deal with publishing channels, and people who take care of the needed hardware and software infrastructure for the game development and, sometimes, for the game publishing (if the project includes Internet game servers, for example).

Enhancing Your Plan for a Great Game

Choosing the game's target market and genre and selecting the right people for the game project aren't the only key points you need to think about when planning your game.

A lot of information is available in books and on the Internet about planning games. Here we'll provide you with an overview of some points you simply can't afford to live without when planning your game.

- *Game goal*: Everything starts with a clearly defined game goal: to win the World Cup, to defeat the evil mage and avoid the world's destruction, to save as many lemmings as you can in each level. This goal ultimately guides the creation of the game storyline and defines whether it's an innovative game or just another clone of a best-selling title.

- *Ending criteria*: Besides the game goal, it's also important to define the game-end criteria: when to end the game, which includes the player's winning criteria (usually the game goal or some goal related to it) and the nonwinning criteria (when the number of lives reaches zero, when the time is up, and so on). When defining the nonwinning game-end criteria, it's also important to define how the player will return to a new game. Providing a saving or autosaving feature is crucial for long games, but might diminish the challenge for a short game such as chess or solitaire.

- *Storyline*: Closely related to the game goal, the storyline provides a background that explains and justifies the game goal, and is crucial to keep the player immersed in the game. When there's a storyline to be followed (not all games have one), everything in the game must contribute to it. The wrong music or a small out-of-place detail in a game would break the illusion as much as seeing someone using a wristwatch in a movie such as *Gladiator*. Creating nonlinear storylines makes the players feel like their decisions make a difference in the game flow, which, although hard to achieve, greatly improves the gaming experience.

- *Playability*: The playability refers to how easy and fun the game is to play. The first 15 playing minutes are vital for players to decide if they'll keep playing, so the game should provide a balance of easy-to-control movements for beginners, and complex (and harder to use) movements for advanced players.

- *Replayability*: This term refers to the desire players have, after finishing a game, to go back and play again. For simple games such as Tetris, the appeal of playing again is obvious, but for more complex games you must plan this appeal in the form of built-in features (such as extra levels unlocked every time the player finishes the game), or as game extensions the player can download or buy.

- *Forgiveness*: Entering in the details of game play, this concept refers to the programmer's ability to provide the correct balance between mathematical accuracy and playability. For example, in a shooter game, if the player shoots a bullet that passes close to an enemy without touching the enemy, it's better to count it as an accurate shot. On the other hand, the programmer might choose to decrement the player's energy only for enemy shots that hit the player's character's torso, ignoring bullets on head, arms, and legs, to make the game easier.

- *Challenge*: You might say that challenge is the opposite of forgiveness: it's the game's ability to provide difficult but not impossible challenges to beat. If the game is too easy—or too hard—the player will simply exchange it for a better-balanced one. The game can provide different skill levels to choose from, and must offer levels with increasingly difficult challenges to keep the player interested.

- *Reward*: Rewarding players when they win is as important as offering good challenges for them to beat. These rewards—which might be special items, money, energy, lives, unlocking new levels, and so on—include prize in-level challenges (such as an amount of gold and extra experience gained for every monster defeated), end-of-level awards (such as presenting a cut-scene and giving bonus points), and a big show at the game ending. Remember: nothing is more frustrating for a player than spending dozens of hours to win a game only to see a puny "congratulations" screen in the end!

- *Saving and registering*: How the game will save the players' evolution and the means it provides to the players to register their experience are important parts of the game playability and reward system. In long games, providing a way for players to start easily from where they left off, a way to register their high scores and compare to other people, and even the ability to "take pictures" from the game to present later to their friends might make the difference needed to provide the right appeal.

- *Game "ecosystem"*: Nowadays, the game team must remember that a game isn't only a game. It includes communities of players on the Internet, homemade extensions created by fans, and so on. These thoughts must guide all game development, from planning a long-term game franchise, coding a game that allows expansions, and establishing marketing approaches to increment the participation of fans in online communities, among other initiatives.

- *Polishing*: A great game is only great if every detail is planned and developed to contribute to player immersion, and especially if such details are tested to work as planned. If a game appears to offer some freedom of choice to the player, but presents a "you can't do this" message—or, even worse, an error message—every time the player tries something imaginative, it's halfway to a total failure. Always remember to include test time in every game project, even for the simpler ones!

Enough planning for now. In the next section, you'll get to create your first XNA program and explore the game programming concepts behind it.

XNA Game Programming Concepts

In this section you'll create an empty XNA game solution, and dig into the solution details to understand the basic concepts behind the program.

If you haven't done so, be sure to download and install the latest version of XNA Game Studio and Visual C# Express Edition from `http://www.microsoft.com/XNA`. If you already have Visual Studio 2005, XNA Game Studio will work just fine. The samples in this book work in either programming environment.

Once everything is in place, run Visual C# now and choose File ➤ New Project. You'll see the dialog box in Figure 1-1.

In this dialog, click the Windows Game (2.0) project type and click OK to create a new game project named `WindowsGame1`. Notice that version 2.0 includes extra project types in the New Project dialog, so be sure you always select the 2.0 version of the project templates.

You'll probably want to pay attention to the Location field in this dialog, because it shows the location in which your project will be created. You're free to change this location to your directory of choice.

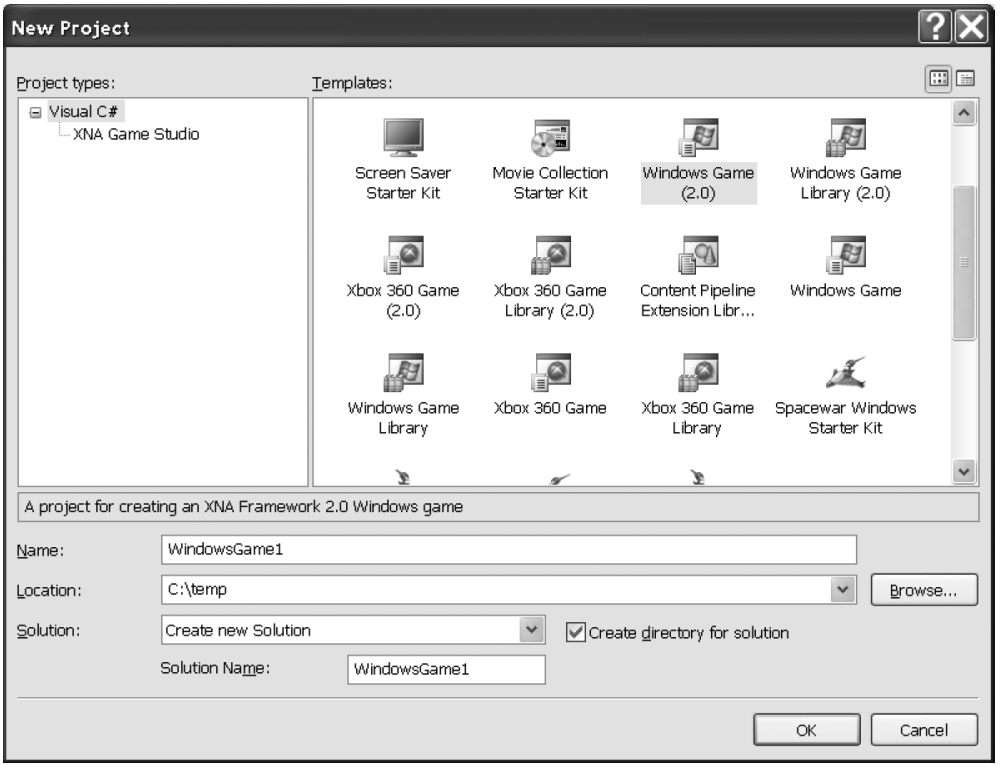

Figure 1-1. *Creating a new Windows Game (2.0) project in Visual C# Express Edition*

Once the project is created, choose the Start Debug icon in the toolbar, or press the F5 key to run the newly created game. Although it's not impressive right now—just a blue screen—as you'll see, this program has all the basics needed to start coding a game.

Close the game window, and analyze the files that were created for you. The Solution Explorer shows the newly created files (see Figure 1-2).

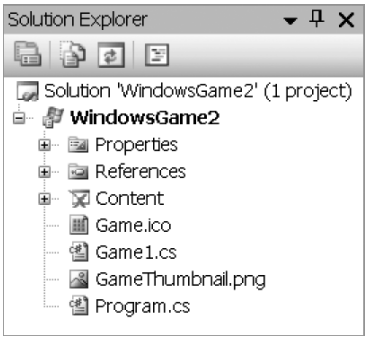

Figure 1-2. *The Solution Explorer for a Windows Game project*

Along with an icon and a thumbnail file, in Figure 1-2 you can see that two code files were created for you: `Program.cs` and `Game1.cs`. To better understand what these files mean, you have to learn a basic game programming concept: the game loop.

General Game Structure

The central logic for every game includes preparing the environment where the game will run, running the game in a loop until the game ending criteria is met, and cleaning up the environment.

The idea of having the main program logic running in a loop is crucial for a game, because the game needs to keep running whether or not it has user interaction. This doesn't happen with some commercial applications, which only do something in response to user input.

The following pseudocode presents a simplified game structure, including the game loop:

```
Initialize graphics, input and sound
Load resources
Start game loop.  In every step:
    Gather user input
    Perform needed calculations  (AI, movements, collision detection, etc.)
    Test for game ending criteria – if met, stop looping
    Draw (render) screen, generate sounds and game controller feedback
Finalize graphics, input, and sound
Free resources
```

It's a simplified view—for instance, you can load resources inside the game loop when beginning each game level—but it still provides a good idea about a game's internal details.

Before XNA, this game structure had to be coded from scratch, so you had to contend with many details that weren't directly related to your game. XNA hides most of this complexity from you. When you create a new Windows Game project, the two files created encompass creating an object of the `Microsoft.Xna.Framework.Game` class (`Game1` object), presenting the code with the meaningful methods of this class you need to override, and calling the `Run` method, which starts the game loop.

The next pseudocode fragment presents `Game1` methods organized as the generic game loop presented before, so you can understand the general structure of the code before entering its details.

```
Game1() – General initialization (Game1.cs)
Initialize() – Game initialization (Game1.cs)
LoadContent() – Load Graphics resources (Game1.cs)
Run() - Start game loop (Program.cs).  In every step:
```

```
   Update() - Read user input, do calculations, and test for game ending (Game1.cs)
   Draw() - Renderization code (Game1.cs)
UnloadContent() - Free graphics resources (Game1.cs)
```

Comparing the two preceding pseudocode excerpts, you can see that the Windows Game project type provides you with a ready-made basic game structure, so you can start by including your game-specific code.

Let's see the details for each of the project files.

Opening the `Program.cs` file, you can see that there are only ten code lines (not counting the `using` statements), as presented in the following code snippet:

```
static class Program
{
    static void Main(string[] args)
    {
        using (Game1 game = new Game1())
        {
            game.Run();
        }
    }
}
```

This code fragment includes the `Program` class, where you have the XNA application entry point—the `Main` function. This function has only two lines: one for creating the `game` object from the `Game1` class, and another for calling the `Run` method of this object, which, as you already know, starts the game loop.

Note that by creating the object in a `using` statement, it is automatically freed when the statement ends. Another point to remember is that the `args` argument on the `Main` function receives the command-line parameters used when calling the game. If you wish to include command-line arguments in your game—such as special cheat codes for helping you test the game—this is where you need to deal with them.

The `Game1` class is implemented in the `Game1.cs` file. A quick look at the `Game1` class in this file shows you that it's derived from the `Microsoft.Xna.Framework.Game` class, the base class offered by XNA that encapsulates window creation, graphics, audio and input initialization, and the basic game logic we already talked about.

Let's open it to explore its details in the next sections.

Game Initialization

The `Game1` class starts by defining and creating objects that will reference the graphics device manager, most commonly referred to in the gaming world as *device*, and a `SpriteBatch` object, used to draw text and 2-D images. The `Game1` class constructor also

configures the root directory for the content manager, which is the entry point for the XNA Content Pipeline, so the XNA Framework is informed of where to find the game content (graphics, sounds, 3-D models, fonts, and so on). The following code bit presents the device and content manager initialization:

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


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

You don't need to change these first lines, nor include code here—you can simply use them as is. In the next sections we'll see some details about the device and the Content Pipeline, so you can get an overall idea of what's happening behind the scenes.

The Graphics Device Manager

The graphics device manager, or simply device, is your entry point to the graphics handling layer, and includes methods, properties, and events that allow you to query and change this layer. In other words, the device represents the way to manage the access to the graphic card feature.

For now, all you need to know is that by creating the `graphics` object of the `GraphicsDeviceManager` class, a game window is created for you, and you'll use the `graphics` object when performing any graphics operation. All the complexities about querying the features and initializing the 3-D graphics layer are hidden from you.

The Content Pipeline Manager

The Content Pipeline is one of the most interesting features XNA brings you, because it simplifies how your game deals with content generated by different content generation tools.

In a non-XNA game, you have to worry about how to load game content as audio, graphics, and 3-D models: Where is the content located? How will your program read this content? Do you have the right libraries to read the content in the format it was created in, by the commercial 3-D tool you're using to create it?

The Content Pipeline streamlines the processing of all game content so you can deal with it easily. It comprises a number of steps, which include importers to read the content and generate a well-known format, a processor that reads this format, a content compiler that generates the ready-to-use content, and finally the content manager. Figure 1-3 presents a high-level view of the Content Pipeline.

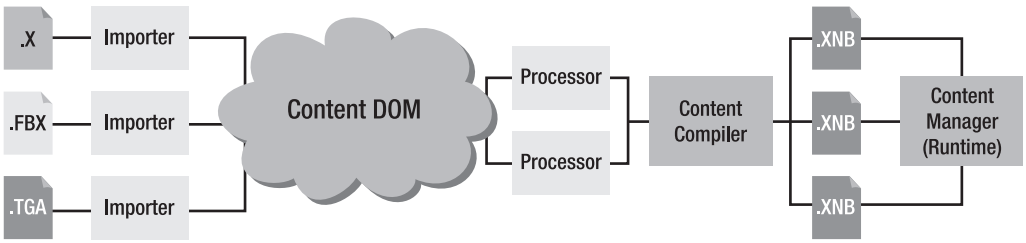

Figure 1-3. *The XNA Content Pipeline*

One interesting thing about the Content Pipeline is that it is based on content you effectively include in your C# project. That means that when the project is built, the content is transformed into a recogizable format and moved to a known directory, so the program will always know where to find the content and how to read it.

When including content in your XNA program, you use one of the content importers provided as part of the framework. These importers normalize the content data, putting it in a format that can be easily processed later. The importers support the following file formats:

- *3-D file formats*: X (used by DirectX), FBX (transport file format, originally created by Autodesk and supported by most commercial and many freeware tools).
- *Material file formats*: FX (effect files, which can be used to describe 3-D model rendering details or add effects to the 3-D scene).
- *2-D file formats*: BMP, DDS, DIB, HDR, JPG, PFM, PNG, PPM, and TGA (the most commonly used image file formats).
- *Font description*: SPRITEFONT (XML files used by XNA, which describe how to generate a texture map from a specific font type size. The game then uses the images on the texture map to write text onscreen).
- *Audio file formats*: .XAP (generated by the XACT tool, which imports most of the audio file formats).

After the importers process the content, the processors will read this content and generate an object the game can handle at runtime.

Finally, the game uses the content manager to read such objects so they can be easily used.

You can extend the content compiler to include new processors, and you can also extend the Content Pipeline with new importers, so you don't have to stick to the predefined formats.

Game Initialization Methods in an XNA Game

Looking back at the game logic pseudocode, you can see that before entering the game loop you have to do the needed initialization and load the game resources. Besides the class constructor, seen in the previous sections, in XNA such initialization is done in the `Initialize` and `LoadContent` methods.

For now, all you need to know is why there are two initialization routines; in later chapters you'll see details and examples for each of these methods.

The `Initialize` method is called once when you execute the `Run` method (described in the beginning of this section), just before the game loop starts. This is the right place to include any nongraphical initialization routines, such as preparing the audio content.

This method also includes a call to its base method, which iterates through a `GameComponents` collection and calls the `Initialize` method for each of them. That means that you can create game components that the `Game` class will also call, when you're creating more sophisticated games. But don't worry about this detail right now: we'll get back to it when creating our games.

The graphics are loaded in a separate method because sometimes the game needs to reload the graphics. The graphics are loaded according to the current device settings to provide maximum performance. So, when these settings change (such as when you change the game resolution or when you go from windowed to full screen mode), you need to reload the graphics. The `LoadContent` method is called every time the game needs to load or reload the graphics.

Game Finalization

Before presenting the game loop–related methods, we'll give a quick overview of the game finalization routines.

Because XNA's internal closing routines and XNA's garbage collector do most of the finalization routines for you, the finalization is simplified.

The basic game project you created includes an overload for the `UnloadContent` method. Like its peer used to load graphics, this method is called every time the game needs to free any graphics resources you have loaded.

Advanced games might include specific routines in each game class to load and unload graphic resources, not requiring the use of these load and unload methods.

Game Loop

Most of the game processing occurs inside the game loop. It's here where the game checks if there is player input to process, the game characters' artificial intelligence is calculated, the game components' movements are executed, the collisions between them are considered, the game ending criteria is checked, and finally, where the controller vibration is activated, the sound is played, and the screen is drawn.

The `Microsoft.Xna.Framework.Game` class provides two overridable methods that the internal game loop calls: `Update`, where you must include the game calculations, and `Draw`, where you draw the game components. Let's take a closer look at these methods, presented in the next code snippet, to highlight some relevant details:

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

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

The first important point to discuss is the `gameTime` parameter received by both methods. This parameter is crucial to all the game logic, because the game must know how much time has passed since the last step on the game loop to do the right calculations—for example, to calculate the correct position for the game components according to their speeds in the game. Let's take a closer look at the `GameTime` class properties:

- `ElapsedGameTime`: This property represents the amount of game time since the last time the game loop was called. Dealing with game time means that the game loop is called a fixed number of times per second, so the game logic can use game time as a basic unit of time to perform calculations. Creating games based on game time instead of real time is easier, because the game can define movements expressed in units per game update, simply incrementing the game components by the calculated rate in every update. When the `IsFixedTimeStep` property of the `Game` class is `true`, this class ensures that `Update` will be called the right number of times per second, dropping frames in a game slowdown if necessary.

- `ElapsedRealTime`: This property represents the amount of real time since the last time the game loop was called. By setting the `IsFixedTimeStep` property of the `Game` class to `false`, the game loop will run at maximum speed, being called as many times as possible per second. This might increase the code complexity, but also might allow for greater speed in the game.

- `TotalGameTime` *and* `TotalRealTime`: These properties represent the total amount of time since the game started, counted in game time (fixed units per second) or real time.

- `IsRunningSlowly`: If the `Game` class is calling the `Update` method less than defined in the `Game.TargetElapsedTime` property, this property is set to `true`, so the game has the information to do any needed adjustments.

Another detail worth mentioning about the `Update` method is that it comes with a predefined code for ending the game when the Back button is pressed in the Xbox 360 controller:

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

The `GamePad` class allows access to the current state of the controller and enables the game to fire the controller vibration. The class doesn't buffer user input, so the information you gather is exactly synchronized with current user interaction. As you can infer by the previous code, you can check for buttons, triggers, thumbsticks, or directional pad status.

We'll talk about dealing with player input in the next chapter, including gamepad, mouse, and keyboard input.

The `Draw` method includes a line to clear the graphics device, filling the game window with a single color—`CornflowerBlue`:

```
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
```

As we stated before, the device (represented here by the `graphics` variable) is your interface to the graphics layer and will be used in every graphics operation. In this case, the code shows a use of the `GraphicsDevice` property, which exposes properties and methods that allow reading and configuring many details about game rendering. We won't get into further details about this class now; you'll learn more about it in the next chapters, when seeing the basics of 2-D and 3-D game programming and when creating your games.

Summary

This chapter showed basic game programming concepts presented in a Windows Game XNA project type. These general concepts are present in any game, so make sure you understand the idea behind the general game structure, especially the idea of the game loop:

```
Initialize graphics, input and sound
Load resources
Start game loop.  In every step:
    Gather user input
    Perform needed calculations  (AI, movements, collision detection, etc.)
    Test for game ending criteria – if met, stop looping
    Draw (render) screen, generate sounds and game controller feedback
Finalize graphics, input, and sound
Free resources
```

It's also important to review the mapping of this general structure for games to the XNA `Game` class overridable methods:

```
Game1() – General initialization (already written for us)
Initialize() – Include nongraphics initialization here
LoadContent() – Include graphics initialization here
Run() - Start game loop.  In every step:
    Update() - Include code here to read and process user input, do calculations
                for AI, movements, and collisions, and test for game ending
    Draw() – Include the drawing (renderization) code here
UnloadContent() – Free graphics resources
```

In the next chapter, you'll write some simple examples that explore 2-D game programming concepts, so you'll be ready to start creating 2-D games with XNA.