



Tom Miller
Dean Johnson

Foreword by Shawn Hargreaves,
Principal Software Design Engineer,
XNA Game Studio, Microsoft

XNA Game Studio 4.0 Programming

Developing for Windows® Phone 7
and Xbox 360®

Developer's Library



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data is on file

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-672-33345-3

ISBN-10: 0-672-33345-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana
First printing, December 2010

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Deadline Driven
Publishing

Indexer
Erika Millen

Proofreader
Jennifer Gallant

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Composition
Nonie Ratcliff

Contents at a Glance

Introduction	1
1 Getting Started	5
2 Sprites and 2D Graphics	13
3 The Game Object and the Default Game Loop	29
4 Introduction to 3D Graphics	41
5 Lights, Camera, Action!	85
6 Built-In Shader Effects	105
7 States, Blending, and Textures	141
8 Introduction to Custom Effects	171
9 Using the Content Pipeline	215
10 Having Fun with Avatars	239
11 Understanding Performance	287
12 Adding Interactivity with User Input	311
13 Turn Up the Volume	353
14 Storage	375
15 Gamer Services	391
16 Multiplayer Networking	409
17 Using Media in XNA Game Studio	441
A Reach vs. HiDef Chart	455
B Using the Windows Phone FMRadio	459
C Windows Phone 7 Launchers and Choosers	463
D Dealing with Tombstoning	479
Index	487

Contents

Foreword xiv

Acknowledgments xv

About the Authors xvi

Introduction 1

So You Want to be a Game Developer? 1

A Brief History of XNA Game Studio 1

What Is Available in Game Studio 4.0? 3

Why This Book? 4

1 Getting Started 5

Installing XNA Game Studio 4.0 5

 Downloading the Tools 6

 App Hub Membership 6

 XNA Game Studio Connect 9

Writing Your First Game 11

 Your First XNA Game Studio Windows Game 11

 Your First XNA Game Studio XNA Xbox 360 Game 11

 Your First XNA Game Studio Windows Phone 7

 Game 12

Download Samples 12

Summary 12

2 Sprites and 2D Graphics 13

What Does 2D Mean? 13

Show Me Something on Screen 14

Spritebatch 16

 Drawing 16

 Moving Things Around 19

 Animation 20

 Controlling State 21

Rendering Text 25

Summary 27

3 The Game Object and the Default Game Loop 29

What Is in a New Project?	29
The Game Class	32
Virtual Methods	32
Methods	33
Properties	34
GameTime	34
Game Loop	36
Update and Draw	36
Components	38
GameComponents	38
Summary	40

4 Introduction to 3D Graphics 41

3D Graphics in XNA Game Studio	41
What Are 3D Graphics?	42
Makeup of a 3D Image	42
3D Math Basics	43
Coordinate Systems	44
Vectors in 3D Graphics	46
Matrix	53
Graphics Pipeline	61
Graphics Card	62
Vertex Shader	62
Backface Culling	63
Viewport Clipping	63
Rasterization	64
Pixel Shader	64
Pixel Tests	64
Blending	65
Final Output	65
Reach and HiDef Graphics Profiles	65
Graphics Profiles Define Platform Capabilities	66
The Reach Profile	66
The HiDef Profile	66

Let the 3D Rendering Start	67
GraphicsAdapter	67
GraphicsDevice	69
Drawing with Primitives	71
Summary	83

5 Lights, Camera, Action! 85

Why Do I See What I See?	85
View Matrix	87
Projection Matrix	88
Perspective	89
Orthographic	93
Camera Types	93
Static Cameras	94
Models	95
What Is a Model?	95
Rendering Models	99
Summary	103

6 Built-In Shader Effects 105

Using BasicEffect	106
Basic Lighting	108
Textures, Vertex Colors, and Fog	114
Using the Effect Interfaces	121
Using DualTextureEffect	122
Using AlphaTestEffect	124
Using EnvironmentMapEffect	124
Using SkinnedEffect	127
Summary	140

7 States, Blending, and Textures 141

Device States	141
BlendState	142
DepthStencilState	149
Render Targets	155
Faking a Shadow with a Depth Buffer and Render Targets	158

Back to Device States	161
The Stencil Buffer	161
RasterizerState	164
SamplerStates	166
Other Texture Types	169
Summary	170

8 Introduction to Custom Effects 171

What Is a Custom Effect?	171
High Level Shading Language	172
Creating Your First Custom Effect	172
Parts of an Effect File	173
Global Variables	174
Vertex Structures	174
Drawing with a Custom Effect	177
Vertex Color	179
Texturing	180
Setting Sampler States in Effect File	183
Textures Repeating	184
Lighting	186
Ambient Lighting	186
Triangle Normals	190
Diffuse Lighting	192
Emissive Lighting	198
Specular Lighting	199
Fog	202
Point Lights	206
Effect States	209
Alpha Blending Using Effect States	211
Summary	213

9 Using the Content Pipeline 215

Tracing Content Through the Build System	215
Content Processors	216
Content Importers	223
Combining It All and Building Assets	226
Combining What You Learned So Far	235
Summary	238

10 Having Fun with Avatars 239

- Introduction to Avatars 239
 - Accessing Avatar Information Using AvatarDescription 240
 - Loading Avatar Animations with AvatarAnimation 243
 - Drawing the Avatar Using AvatarRenderer 246
- Modifying Avatar Lighting 248
- Playing Multiple Animations 249
- Blending Between Animations 253
- Interacting with Objects 260
- 2D Avatars Using Render Targets 263
- Custom Avatar Animations 265
 - Creating the Custom Animation 266
 - Building the Custom Animation Type 267
 - Creating the Content Processor 273
 - Adding the Custom Animation to Your Game 283
 - Updating Your Game to Use the Custom Animation 284
- Summary 285

11 Understanding Performance 287

- General Performance 287
 - Who Takes Out the Garbage? 289
 - Multithreading 292
- Graphics Performance 293
- Measuring Performance 295
 - Performance Measurement Tools 306
- Cost of Built-In Shaders 307
- Summary 309

12 Adding Interactivity with User Input 311

- Using Input in XNA Game Studio 311
- Polling versus Event-Based Input 312
- The Many Keys Of A Keyboard 312
 - Reading Keyboard State 313
 - Moving Sprite Based on Keyboard Input 315
 - Onscreen Keyboard 316

Precision Control of a Mouse	320
Reading Mouse State	320
Moving Sprite Based on Mouse Input	322
Setting the Mouse Position	324
Xbox 360 Gamepad	324
Reading Gamepad State	325
Moving Sprites Based on Gamepad Input	329
Thumb Stick Dead Zones	332
Other Types of Controllers	332
Is the Gamepad Connected?	333
Multitouch Input For Windows Phones	334
Reading the TouchPanel Device State	334
Determine Number of Touch Points	336
TouchPanel Width, Height, and Orientation	337
Moving Sprite Based on Multitouch Input	337
Reading Gestures from the TouchPanel	339
Displaying GestureSample Data	341
Windows Phone Sensors and Feedback	342
Acceleration Data using the Accelerometer	344
Locating a Windows Phone with the Location Service	348
Providing User Feedback using Vibration	351
Summary	351

13 Turn Up the Volume 353

Playing Sound Effects	353
Using SoundEffect for Audio Playback	354
Microsoft Cross-Platform Audio Creations Tool (XACT)	360
Dynamic Sound Effects	368
Recording Audio with a Microphone	368
Generating Dynamic Sound Effects	371
Summary	374

14 Storage 375

What Is Storage?	375
Isolated Storage	375
Saving and Loading Data	377
The IsolatedStorageFile Object	379

XNA Game Studio Storage	380
Recreating the Project on Xbox	380
Devices and Containers	382
Getting a Device	383
Looking at the API	387
Loading Loose Files from Your Project	388
Summary	390

15 Gamer Services 391

GamerServicesComponent	391
Guide Class	392
Trial Mode	392
Now the Bad News	397
Platform-Specific Guide Functionality	397
Gamers and Profiles	402
GameDefaults	405
Presence	406
Privileges	406
With Friends Like This...	407
Summary	408

16 Multiplayer Networking 409

Multiplayer Games	409
Getting Ready for Networking Development	410
Main Menu and State Management	412
Creating a Network Session	416
Building a Game Lobby	423
Playing the Game	425
Searching for an Available Network Session	430
Joining an Available Network Session	435
Sending Player Invites	438
Simulating Real World Network Conditions	439
Summary	440

17 Using Media in XNA Game Studio	441
What Is Media?	441
Playing a Song	441
MediaPlayer	442
Songs and Metadata	443
Media Enumeration	444
Media Library	444
Video	448
Rendering a Video	448
Visualizations	451
Rendering Visualization Data	451
Summary	453
A Reach vs. HiDef Chart	455
B Using the Windows Phone FMRadio	459
C Windows Phone 7 Launchers and Choosers	463
D Dealing with Tombstoning	479
Index	487

Foreword

I got my first computer in 1989, when I was 13. It was an Oric-1 with a 1-MHz CPU and 48k RAM. It didn't come with any games, but when you switched it on, up came a screen that said:

Ready



It was ready to be programmed, and the manual dived straight into teaching me how to do this:

First the bad news. ORIC doesn't understand English. But now the good news. You don't have to learn a complicated electronic language, because ORIC speaks a language called BASIC. If your machine is switched on, we'll see how easy this is. Type:

```
PRINT "HELLO"
```

and then press the [RETURN] key.

Wow! I just made my first program, and the computer did exactly what I told it to do. What a thrill! I was hooked.

A few years later, we upgraded to an Atari ST. This was better than the Oric in all ways but one: bigger, faster, higher resolution. When I switched it on, excited to start programming, I saw a desktop waiting for me to launch an application. Where was the programming language? I was horrified to learn I could not program this machine without first locating and buying an expensive third-party interpreter or compiler. If I had not already learned to program on the Oric, this hurdle would have been too steep, so I would never have bothered to program the Atari, never gotten a job in the games industry, never joined the XNA team, and would not be writing this foreword today.

Learning to program is important for many reasons. As a society, we need skilled programmers to create and maintain the programs that make the modern world work. As a democracy, we need people who understand computers well enough to make sure we control these programs, and not the other way around. And as individuals, programming can be great fun.

I worry that as computers have become more powerful, they also became more intimidating. I think the best thing about XNA Game Studio is how it restores the immediacy and fun I experienced with my Oric. To lower the barriers to entry, we have a free and easy-to-learn programming language called C#. To provide that magical thrill of making the machine do your bidding, we have powerful yet simple APIs, and the ability to run your creations not just on PC, but also Xbox 360 and Windows Phone. And last but not least, to learn how to put all these pieces together, we have books like this one. Nice work Dean and Tom!

I hope you have as much fun making games with XNA as I did with my Oric.

—Shawn Hargreaves, Principal Software Design Engineer,
XNA Game Studio, Microsoft

Introduction

So You Want to be a Game Developer?

We've worked in what you would call the "game industry" for years, and during our time, we've met many people and developers. They almost universally share a similar trait in that they either have been or at some time wanted to be a game developer. Games are everywhere—in movies, television, and the Internet. The audience of people playing games has expanded, too, with the popularity of online social media sites such as Facebook.

Much like a book or a movie, games can be a journey. They can tell a story and put the person experiencing it into a whole new world. Unlike a book or a movie, though, games are interactive. The player actually has an effect on the world. People can get immersed in these worlds, and once they are, a natural extension of that is the desire to create other (even better) worlds for others to get immersed in. With this desire to create other worlds, we have a budding game developer.

You may find this surprising, but writing a game is hard work, or, writing a good game is hard work. First, you need to have a good idea, and hopefully, that idea is fun. After you have the idea, though (even if it is the best idea ever), you still have to write the actual core game play and mechanics. After that, there is a ton of work to get something that looks polished. Finishing developing a game is probably one of the most difficult things for someone to do, particularly if he or she isn't aware of how much work is required.

It isn't easy to become a professional game developer. For established publishers and studios, it is difficult to come in off the street and get a job writing games. Publishers and studios want people with experience who have shipped games. Times are changing, however, with platforms that allow self-publishing (such as XNA Game Studio on Xbox or for Windows Phone 7). Anyone can publish a game and actually charge for it to make money.

A Brief History of XNA Game Studio

This book covers XNA Game Studio 4.0, and it has been quite a journey to get to this fourth release. XNA Game Studio 4.0 naturally builds on previous versions of XNA Game Studio, which build on a combination of technologies that go way back.

The technologies go all the way back to Windows 95 as a matter of fact! When Windows 95 was released, Microsoft released something called the Windows Game SDK,

which would later be renamed to something you might be more familiar with—DirectX. It is somewhat humorous that the X that is everywhere started off as a joke. The original APIs released in DirectX 1.0 were DirectDraw, DirectInput, and DirectSound. The X was used as shorthand to replace each of the actual API component names to talk about the whole group, and that X soon became the official name. It migrated all the way through to the original Xbox to XNA.

Before DirectX, making games was much more difficult than it is today. There wasn't a standard way of talking to the various pieces of hardware that existed on all the computers, so if you wanted to write a game that worked for everything, you had to write special code for each piece of hardware you wanted to support. With DirectX, there was a standard way of accessing the hardware, and game developers and hardware manufacturers all over rejoiced!

DirectX has gone through quite a few versions, adding new functionality as it developed (such as 3D graphics, networking, and music) and is now on version 11 that shipped with Windows 7. When people talk about DirectX 11, though, they are almost always talking about Direct3D 11, as no other components have changed since DirectX9.

I got ahead of myself, though. Let's backtrack a little to DirectX 7.0. This was the first version of DirectX that included functionality for a language other than C, as it included DirectX for Visual Basic. This was actually when I joined the DirectX team, specifically to work on that portion of the product. I continued to work on it through DirectX 8.0.

DirectX 8.0 was the first version to include programmable shaders, something you read more about in Chapter 8. It's actually hard to believe how far we've come since then, as there isn't any way to write graphics code without shaders! DirectX 8.0 is also the time I began looking at this funny thing called .NET.

DirectX 9.0 was the first release of DirectX that included a component specifically designed for the Common Language Runtime (CLR). This component is Managed DirectX. A lot of work went into that project and although it looked only vaguely familiar to people using DirectX, it fit right in for people using C# and the other managed languages.

The response Managed DirectX received was surprising and a bit overwhelming. Although DirectX for Visual Basic had expanded the development audience, Managed DirectX did so even more. The API was cleaner, easier to use, and felt like all of the other managed components that were out there. The biggest worry then (and one you still hear about today) was related to performance. No one could believe that a managed API (particularly one with a garbage collector) could run fast.

After spending a few years working on Managed DirectX, I left the DirectX team in January of 2006 to join a new group that wanted to develop this thing called XNA, which was eventually released late in 2006 as XNA Game Studio Express.

Game Studio changed all the rules. It took the ease of use and power that Managed DirectX had, made it even easier and more powerful, and added the capability to run games on an Xbox 360. Historically, game consoles have always been closed systems, including the Xbox 360. Before Game Studio, the only way to run code on an Xbox 360

was to be a registered native developer, which required a contract with Microsoft and an approved game!

Much like DirectX, Game Studio kept evolving. With 2.0, support for networking via Xbox LIVE was added. Any version of Visual Studio 2005 was allowed to be used (rather than the C# Express that was required in the first version). At the launch of 3.0, new features arrived with the inclusion of support for Zune and the capability to publish and sell games on Xbox LIVE via the Xbox LIVE Community Games (now called Xbox LIVE Indie Games). Version 3.1 included support for the Zune HD, Video, Xbox LIVE Parties, and Avatars.

Game Studio 4.0 is the latest version where the biggest feature is the addition of the Windows Phone 7 development. There are, of course, other updates, too, including a much simpler graphics API and features such as microphone support and dynamic audio. This version is what this book covers. It has been a wild ride getting here, and I can't wait to see where we go next.

What Is Available in Game Studio 4.0?

Game Studio 4.0 has everything you need to make great and compelling games for Windows Phone 7, Xbox 360, and Windows. The Game Studio 4.0 release is broken into two different profiles: One is called Reach, which encompasses features that exist on all platforms, and the other is called HiDef, which includes extra features that exist only on Xbox 360 and Windows (depending on the graphics card). Each of these areas is discussed in depth later in the book. Table 1 shows the major namespaces and feature areas contained in the framework.

Table 1 Namespaces included in XNA Game Studio 4.0

Namespace	Features
Microsoft.Xna.Framework	General framework features, math, and game objects
Microsoft.Xna.Framework.Graphics	All graphics features, including 2D and 3D
Microsoft.Xna.Framework.Audio	All audio features
Microsoft.Xna.Framework.Input	All input features, including game pads, keyboard, and mouse
Microsoft.Xna.Framework.GamerServices	Functionality for accessing portions of the Xbox LIVE services
Microsoft.Xna.Framework.Media	Media features for pictures, music, and so on
Microsoft.Xna.Framework.Content	Content pipeline features
Microsoft.Xna.Framework.Net	All synchronous networking features
Microsoft.Xna.Framework.Storage	Storage features for HiDef

Why This Book?

This book is not only the best reference source for Game Studio 4.0, it also has the added benefit of being authored by two of the people responsible for bringing this product to you. It would be difficult to find two people more knowledgeable on this subject than us, and we want to transfer that knowledge to you.

We cover every aspect of game development using Game Studio 4.0 for every platform that it supports. This includes specifics for APIs that exist on Windows Phone 7 (such as accelerometer and other sensors) that are not part of the Game Studio API, but are important for games.

This page intentionally left blank

The Game Object and the Default Game Loop

When you create a new project, many things happen behind the scenes, and many features to help drive the game available to you. In this chapter, you learn about these features, including:

- The game class
- The standard game loop
- Game components

Up until now, you've created new projects and added some code to do other fancy things, but you haven't taken a step back to look at the default game template. Now is a good time to take a look at what is provided for you automatically and the other features available.

What Is in a New Project?

Open Visual Studio and create a new Game Studio 4.0 Windows Game project. Notice that your main project includes two code files (`program.cs` and `game1.cs`), and you have a content project you previously used. You can safely ignore everything in `program.cs` because it is simply the stub that launches the game. As a matter of fact, this isn't even used on Windows Phone 7.

The interesting things that are discussed in this chapter are in `game1.cs`. Notice first that the `Game1` class that is created comes from the `Game` object provided by Game Studio. The initial starting project gives you everything you need to start creating a game. It has a spot for initialization, a spot to load the content your game needs, a spot to update the game state, and a spot to render everything.

More things happen behind the scenes than you are probably aware of, however. Start with the first thing you see in the constructor, creating the `GraphicsDeviceManager`.

```
graphics = new GraphicsDeviceManager(this);
```

This one line of code starts a chain reaction of operations. It naturally creates a new `GraphicsDeviceManager` (which is discussed in just a moment), but it does more than that. This object implements `IGraphicsDeviceService` and `IGraphicsDeviceManager`. When you create the object, it takes the game parameter you've passed in (that is, the `this` parameter) and adds itself to the `Services` property of the game object.

Note

You can create and add your own services to this property (it maintains a list of services), and it is a convenient way to get game-specific services directly from the game rather than passing them around everywhere.

After the graphics device manager has been added to the services list, the actual graphics device is created when the constructor has finished executing. The default options work just fine, but you actually do have some control over the settings the device has.

Notice that quite a few different properties on this object can be used to control how the device is created or to get information about it. The first one is the `GraphicsDevice`. Right now, it hasn't been created yet, but after it has been, it can be accessed here. You most likely never need it, though, because the `GraphicsDevice` is a property of the `Game` itself.

The `GraphicsProfile` is another property you can access. Profiles are discussed in Chapter 4, "Introduction to 3D Graphics". Next is the `IsFullScreen` property that behaves differently depending on the platform you run. The default value here is `false`, although on Xbox 360, it doesn't matter what this is set as because you are always full screen on that platform. On Windows, setting this to `true` causes the device to be created in what is called full screen exclusive mode, and your rendering encompasses the entire screen. On Windows Phone 7, this controls whether the system tray bar is visible or not visible.

Note

Taking over the full screen in exclusive mode on Windows is not a nice thing to do without the user asking you to do so. In modern operating systems, the graphics hardware is shared nicely between games and the operating system, and forcing the operating system to yield to your game can cause behavior that some of your players may very well find annoying (the authors here included).

The next set of properties is the most commonly changed, and it includes the preferences. Because they are preferences and not requirements, the runtime attempts to use these settings, and if it cannot use them, it falls back to what it feels is the closest to what you requested. These properties are `PreferMultisampling`, `PreferredBackBufferWidth`, `PreferBackBufferHeight`, `PreferBackBufferFormat`, and `PreferDepthStencilFormat`.

The back buffer is where your content is rendered, and the sizes in these preferences (width and height), control how large that area is. On Windows, in nonfull screen mode, this also controls the size of the window. In full screen mode, it controls the resolution of the monitor when it takes exclusive control of it. On Xbox 360 and Windows Phone 7, the devices have a built-in native resolution. For Windows Phone 7, the device has a resolution of 480×800 (in portrait mode), whereas the Xbox is configurable. On each of these platforms, if you ask for a different back buffer resolution, it is scaled to the native device resolution.

Multisampling is the process used to remove what are called the “jaggies” from rendered images. These jagged edges are formed normally on the edges of objects or on lines that are not on pixel boundaries (for example, nonhorizontal or vertical lines). Multisampling blends each pixel with other pixels around it to help soften these jagged edges. It does this by rendering the image larger and blending multiple pixels down to a single pixel. Although it doesn’t necessarily remove the jagged edges, it certainly can help. There is a performance cost for doing this, so this defaults to false.

The last two preferences are the formats for the back buffer and the depth stencil. Formats are used to describe how data is laid out for the final rendered image. For the back buffer, this is how the color is laid out, and the default for this is actually `SurfaceFormat.Color`. This is a 32-bit format that has 8 bits for red, green, blue, and alpha. The depth stencil buffer formats control how many bits are used for the depth buffer and stencil buffer.

Note

Depth buffers are discussed in more detail in Chapter 7, “States, Blending, and Textures”.

The last two properties are `SupportedOrientations`, which is mainly used for Windows Phone 7, and `SynchronizeWithVerticalRetrace`. Synchronizing with the vertical retrace is a way to prevent tearing by pausing until the device is ready to render the entire screen at once. These are discussed in depth in Chapter 11, “Understanding Performance” when performance and measurement are discussed.

There are also six different events you can hook off of the `graphics` object, most of which are self-explanatory based on the names. The one interesting one is `PreparingDeviceSettings`. This event is triggered right before the device is created, and it gives you the opportunity to override any of the settings before the device is actually created. Use this only if you know the device supports the settings you request.

There are also two methods on the object, `ApplyChanges` which attempts to instantly update the device to the current settings (or create a new device if required), and `ToggleFullscreen`, which makes a windowed game full screen and a full screen game windowed during runtime. Using either of these is rarely required.

The last thing the constructor does is set the root directory of the automatically created content manager to “Content,” which is where your content project places the content you add to your game. The content manager is created for you when the game is created,

so you can begin using it immediately. The content manager is discussed more in depth in Chapter 9, “Using the Content Pipeline.”

```
Content.RootDirectory = "Content";
```

The default template has overrides for five common methods: `Initialize`, `LoadContent`, `UnloadContent`, `Update`, and `Draw`. Although nothing happens in `Initialize` and `UnloadContent`, the other three have basic stub code. The `LoadContent` method creates the sprite batch object you almost certainly need. The `Draw` method clears the screen to the infamous `CornflowerBlue` color. Finally, the `Update` method adds a quick check to see if you’re pressing the Back button on your controller to see if it should exit the game. We get into the flow of these methods and how they’re used in just a moment, but first, let’s take a look at the `Game` class itself.

Note

For Windows Phone 7 projects, there is another very important aspect to the game lifetime you need to understand called “Tombstoning”. It is discussed in depth in Appendix D, “Dealing with Tombstoning”.

The Game Class

The `Game` class is where all of the magic in your game takes place. Almost everything in your game is driven in some part by this class, and it is the root of your project. Let’s dive in and see what it is made of.

Virtual Methods

The `Game` object your class derives from has many knobs and buttons (figuratively) to help control your game and its flow. It has several virtual methods you can override to help control different features, some of which we’ve seen already.

`Initialize`, as you would expect, is called once just after the object is created. It is where you would do most of your initialization (aside from loading content) for your game. This is an ideal place to add new game components for example (which are discussed later in this chapter). `LoadContent` and `UnloadContent` are two other areas where you should load (or unload) your content. Content is loosely defined and can be any external data your game needs, whether it’s graphics, audio, levels, XML files, and so on.

The `Update` method is where you handle updating anything your game requires, and in many games, you can do things such as handle user input. Because most games are essentially a simulation with external forces, you need a central spot where you can perform the operations to advance that simulation. Common things you’d do include moving objects around the world, physics calculations, and other simulation updates.

`Draw` probably needs no further explanation. All drawing code for each scene occurs here. There are two other drawing methods you can override: `BeginDraw` and `EndDraw`. These are called directly before and after `Draw` is called, respectively. If you override the `EndDraw` call, you need to ensure you call the `base.EndDraw` or manually call `GraphicsDevice.Present`; otherwise, you never see your scenes drawn on screen.

Note

`Present` is the last call made at the end of drawing and tells the graphics device, “Ok, I’m done drawing now; show it all on screen.”

Much like the pre- and post-drawing methods, there are also `BeginRun` and `EndRun` methods you can override that are called before the game begins and just after the game run ends. In most cases, you do not override these, unless you are doing something such as running multiple simulations as individual game objects.

You can override the method `ShowMissingRequirementMessage`. Most people probably don’t even realize it is there. By default, this does nothing on non-Windows platforms, and on Windows, it shows a message box giving you the exception detail. This enables customization if the platform you run on doesn’t meet the requirements of your game, which is normally only an issue on a platform such as Windows where you can’t guarantee which features it supports.

The last three methods you can override are mirrors of events you can hook. `OnActivated` is called at the same time the `Activated` event is fired, and it happens when your game becomes activated. Your game is activated once at startup, and then anytime it regains focus after it has lost focus. To mirror that, you use the `OnDeactivated` method, which is called when the `Deactivated` event is fired, and that happens when your game becomes deactivated, such as it exits or it has lost focus. On Windows and Windows Phone 7, your game can lose focus for any number of reasons (switching to a new app, for example), whereas on Xbox 360, you see this only if the guide screen displays.

Finally, the `OnExiting` method is called along with the `Exiting` event. As you can probably guess, this happens just before the game exits.

Methods

Most of the methods on the `Game` class are virtual so there aren’t many here to discuss, and they’re almost all named well, so you can guess what they do. The `Exit` method, which causes the game to start shutting down. The `ResetElapsedTime` method resets the current elapsed time, and you’ll learn more about it later in this chapter. The `Run` method is what starts the game running, and this method does not return until the game exits. On Xbox 360 and Windows, this method is called by the autogenerated main method in `program.cs` at startup, and on Windows Phone 7, this method throws an exception. Due to platform rules, you can’t have a blocking method happen during startup on Windows Phone 7. A timer starts and periodically calls `RunOneFrame`, which does the work of a single frame. You can use this method on Xbox 360 and Windows, but you shouldn’t have to use it since the game object is doing that for you.

The `SuppressDraw` method stops calling `Draw` until the next time `Update` is called. Finally, the `Tick` method advances one frame; namely, it calls `Update` and `Draw`.

Properties

After covering all of the methods, properties are naturally the next item on the list. You've already seen the `Services` property, which enables you to add new services to the game and query for existing ones. You've also already seen the `Content` and `GraphicsDevice` properties, which store the default content manager and graphics device.

A property called `InactiveSleepTime` gives you some control of how your game handles itself when it is not the foreground window. This value tells the system how long to “sleep” when it is not the active process before checking to see if it is active again. The default value of this is 20 milliseconds. This is important on Windows where you can have many processes run at once. You don't want your game to run at full speed when it isn't even active.

Speaking of being active, the `IsActive` property tells you the current state of the game. This maps to the `Activated` and `Deactivated` events, too, as it turns true during `Activated` and false during `Deactivated`. Chapter 12, “Adding Interactivity with User Input” discusses the `IsMouseVisible` property, even though you can probably guess what it does.

The `LaunchParameters` property is used for Windows Phone 7 to get information about parameters required for launching, but this can be used for any platform and translates the command-line parameters on Windows into this object. It is a dictionary of key value pairs. On Windows, if your command line is as follows, the object would have a dictionary with three members:

```
game.exe /p /x:abc "/w:hello world"
```

The first member would have a key of “p” with a value of an empty string. The second member would have a key of “x” with a value of “abc.” The third member has a key of “w” with a value of “hello world.”

On Windows Phone 7, applications are launched with a URI that includes these parameters; for example, if your launch command is as follows, the object would have a dictionary with two members:

```
app://{appguid}/_default#/Main.xaml?myparam1=one&myparam2=two
```

The first member would have a key of “myparam1” and a value of “one.” The second member would have a key of “myparam2” and a value of “two.”

The last two properties are `IsFixedTimeStep` and `TargetElapsedTime`. Timing is so important to game development there is a whole section on that! Because anticipation is probably overwhelming, that section is next.

GameTime

You may not realize it, but a lot of things in a game depend on time. If you create a race game and your cars are going 60 miles per hour, you need to know how much to move them based on a given time. The framework tries to do a lot of the work of handling time for you.

There are two major ways to run a game, and in the framework, they are referred to as “fixed time step,” and “variable time step.” The two properties mentioned in the previous section—`IsFixedTimeStep` and `TargetElapsedTime`—control how time is handled. `IsFixedTimeStep` being true naturally puts the game into fixed time step mode, whereas false puts the game into variable time step mode. If you are in fixed time step mode, `TargetElapsedTime` is the target time for each frame. The defaults for projects are true for `IsFixedTimeStep` and 60 frames per second for `TargetElapsedTime` (which is measured in time, so approximately 16.6667 milliseconds).

What do these time steps actually mean? Variable time step means that the amount of time between frames is not constant. Your game gets one `Update`, then one `Draw`, and then it repeats until the game exits. If you noticed, the parameter to the `Update` method is the `GameTime`.

The `GameTime` object has three properties that you can use. First, it has the `ElapsedGameTime`, which is the amount of time that has passed since the last call to `Update`. It also includes `TotalGameTime`, which is the amount of time that has passed since the game has started. Finally, it includes `IsRunningSlowly`, which is only important in fixed time step mode.

During variable time step mode, the amount of time recorded in `ElapsedGameTime` passed to `update` can change depending on how long the frame actually takes (hence, the name “variable” time step).

Fixed time step is different. Every call to `Update` has the same elapsed time (hence, it is “fixed”). It is also different from variable time step in the potential order of `Update` and `Draw` calls. While in variable time step, you get one `update` for every `draw` call; in fixed time step, you potentially get numerous `Update` calls in between each `Draw`.

The logic used for fixed time step is as follows (assuming you’ve asked for a `TargetElapsedTime` of 60 frames per second).

`Update` is called as many times as necessary to catch up to the current time. For example, if your `TargetElapsedTime` is 16.667 milliseconds, and it has been 33 milliseconds since your last `Update` call, `Update` is called, and then immediately it is called a second time. `Draw` is then called. At the end of any `Draw`, if it is not time for an `Update` to occur, the framework waits until it is time for the next `Update` before continuing.

If at any time, the runtime detects things are going too slow (for example, you need to call `Update` multiple times to catch up), it sets the `IsRunningSlowly` property to true. This gives the game the opportunity to do things to run faster (such as rendering less or doing fewer calculations).

If the game gets extremely far behind, though, as would happen if you paused the debugger inside the `Update` call if your computer just isn’t fast enough, or if your `Update` method takes longer than the `TargetElapsedTime`, the runtime eventually decides it cannot catch up. When this happens, it assumes it cannot catch up, resets the elapsed time, and starts executing as normal again. If you paused in the debugger, things should just start working normally. If your computer isn’t good enough to run your game well, you should notice things running slowly instead.

You can also reset the elapsed time yourself if you know you are going to run a long operation, such as loading a level or what have you. At the end of any long operation such as this, you can call `ResetElapsedTime` on the game object to signify that this operation takes a while, don't try to catch up, and just start updating from now.

Notice that in Windows Phone 7 projects, the new project instead sets the `TargetElapsedTime` to 30 frames per second, rather than the 60 used on Windows and Xbox 360. This is done to save battery power, among other reasons. Running at half the speed can be a significant savings of battery life.

Note

Which time step mode you actually use is a matter of personal preference. We personally choose fixed time step mode, but either can work for any type of game. During performance measurement, though, you should use variable time step mode. This is discussed in Chapter 11, "Understanding Performance."

Game Loop

This chapter has been nothing but text so far. Words, words, and rambling—that just isn't exciting. Let's get something on the screen!

Update and Draw

The basic flow of your game is to initialize everything, and then call `Update` and `Draw` continually until you exit the game. This is what is called the game loop. You've already seen it in action in Chapter 2, "Sprites and 2D Graphics," without even realizing it most likely. To ensure you do realize it, let's do a slightly more complex example.

First, you need to add the `XnaLogo.png` file to your content project from the accompanying CD. Because you are drawing this image much like before, you need to declare a texture variable to hold it. This time, though, also declare a position variable, as follows:

```
Texture2D texture;  
Vector2 position;
```

Of course, you need to load that texture, so in your `LoadContent` method add this line:

```
texture = Content.Load<Texture2D>("XnaLogo");
```

You should probably also initialize that position to something! Add the following to the `Initialize` method:

```
position = Vector2.Zero;
```

Finally, because you need to actually draw the image, replace your `Draw` method with the following:

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

```

    spriteBatch.Draw(texture, position, Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}

```

Running the project now shows you (as you might have guessed) the image in the upper left corner of the window. Because of the position variable in the Draw call, you are set up to get that thing moving! Add the following to the Update method:

```

position = new Vector2(position.X + (float)gameTime.ElapsedGameTime.TotalSeconds,
    position.Y + (float)gameTime.ElapsedGameTime.TotalSeconds);

```

Running the project now has the image slowly moving down and to the right, and if you leave it running long enough, it eventually goes off screen! If you want the image to bounce around the screen, though, it would be complicated code. Do you add to the position or subtract? Instead, store your direction in a new variable:

```

Vector2 velocity;

```

Then, update the Initialize method to include:

```

velocity = new Vector2(30, 30);

```

Finally, change your update method to:

```

position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);

```

These updates provide faster movement and easier code, although the image still goes off screen if you let it. You need to detect if the image has gone off screen and make changes to ensure it “bounces” off the edges. This is the largest section of code you’ve seen so far, but it makes the image bounce around, so modify your Update call to this:

```

position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
if (!GraphicsDevice.Viewport.Bounds.Contains(new Rectangle(
    (int)position.X, (int)position.Y, texture.Width, texture.Height)))
{
    bool negateX = false;
    bool negateY = false;
    // Update velocity based on where you crossed the border
    if ((position.X < 0) || ((position.X + texture.Width) >
        GraphicsDevice.Viewport.Width))
    {
        negateX = true;
    }
    if ((position.Y < 0) || ((position.Y + texture.Height) >
        GraphicsDevice.Viewport.Height))
    {
        negateY = true;
    }
    // Move back to where you were before
    position -= (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
}

```

```

    if (negateX) velocity.X *= -1;
    if (negateY) velocity.Y *= -1;
    // Finally do the correct update
    position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
}

```

Run the project now and you can see the image bounce around the screen! You simply check to see if the image is currently in the bounds of the viewport, and if it is not, check to see which edge it is currently over. Then, move it back to where it was before the update, swap the velocity axis of the sides you've crossed, and update the position again.

Components

As you can probably imagine, if you had to draw everything inside a single class, your code would quickly become a mess of things to keep track of!

GameComponents

Luckily, the framework provides an easy way to encapsulate objects called game components. There are three types of game components you can use for your games: `GameComponent`, `DrawableGameComponent`, and `GameServicesComponent`. The latter is discussed later, but let's dive into the others now.

First, you want to take the image-bouncing code you wrote and move it into a component, so in your main game project, right-click the project and select Add -> New Item. Choose Game Component from the list of templates (it might be easier to find if you choose the XNA Game Studio 4.0 section in the list on the left), name it `BouncingImage.cs`, and then click the Add button.

This adds a new file to your project with a new class deriving from `GameComponent`, which is close to what you want but not quite. Open up `BouncingImage.cs` (it should have opened when you added the component), and change it to derive from `DrawableGameComponent` instead:

```
public class BouncingImage : Microsoft.Xna.Framework.DrawableGameComponent
```

Now you can begin moving the code you used in your `Game` class to render your bouncing image to this component. Start by moving the three variables you added to the new `BouncingImage` class (texture, position, and velocity). Move the code initializing your two vectors into the new class's `Initialize` method and move the code where you modify the vectors in `update` to the new class's `Update` method. You need to do just a few things to complete your bouncing image component.

You need a way to load the texture, and `DrawableGameComponent` has the same virtual `LoadContent` method that the game has, so you can simply override it in your `BouncingImage` class now:

```
protected override void LoadContent()
{
    texture = Game.Content.Load<Texture2D>("XnaLogo");
    base.LoadContent();
}
```

Finally, all you need now is to draw the image. Just like `LoadContent`, `DrawableGameComponent` also has a `Draw` virtual method you can override:

```
public override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(texture, position, Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

As you might have seen already, this won't compile. The `spriteBatch` variable is declared in the game, and it is a private member variable. You can create a new `sprite batch` for this component, but it isn't necessary. If you remember back to earlier in this chapter, we talked about the `Services` property on the `Game` class.

Go back to the `game1.cs` code file and to the `LoadContent` method. Replace the loading of the texture (which you just did in the `BouncingImage` class) with the following line (directly after the `sprite batch` has been created):

```
Services.AddService(typeof(SpriteBatch), spriteBatch);
```

This adds a "service" of type `SpriteBatch` to the game class and uses the just created `sprite batch` as the provider of that service. This enables you to use the "service" from anything that has access to the game. Back in the `BouncingImage.Draw` method, before the `Begin` call, add the following:

```
SpriteBatch spriteBatch = Game.Services.GetService(
    typeof(SpriteBatch)) as SpriteBatch;
```

Now that you have your drawing code in your component compiling, you can remove it from the `Draw` call in the game. It should have nothing but the `Clear` call and the `base.Draw` call now. With everything compiling, you can run your project and you will see absolutely nothing except a blank cornflower blue screen. This is because your component was never actually used! Back in the `Initialize` method in the game, add the following:

```
Components.Add(new BouncingImage(this));
```

That's it. Running the code gets you back to where you were before, but with everything much more encapsulated. It's also much easier to add new instances of the bouncing image; for example, add this to your game's `Update` method:

```
if ( ((int)gameTime.TotalGameTime.TotalSeconds % 5) == 4 )
    Components.Add(new BouncingImage(this));
```

Running it now adds a new set of bouncing images every 5 seconds (actually it adds quite a few because it adds one for every update that happens during that second). You can go ahead and remove that code; it is just an example of how easy it is to include more.

You might have noticed that you didn't actually have to do anything outside of add your component to the `Components` collection for it to start working magically. Your `Initialize` method is called for you, as is your `LoadContent` method, and the `Update` and `Draw` methods were called for each frame.

By default, components are updated and drawn in the order they were added, and they are always updated and drawn. However, these are all changeable, too. If you set the `DrawOrder` property, when the components are being drawn, the components with the lower `DrawOrder` values are drawn first. Similarly, if you use the `UpdateOrder` property, the components with the lower `UpdateOrder` value are updated first. The higher value these properties have, the later they happen in the list of components. If you want something drawn as the last possible component, you set the `DrawOrder` to `int.MaxValue`, for example. Of course, if you have more than one component with the same `UpdateOrder` or `DrawOrder`, they are called in the order they were added.

Of course, there might be times when you don't want your component to be drawn at all! If this is the case, you can simply set the `Visible` property to `false`, and your `Draw` override is no longer called until that property switches back to `true`. If you need to temporarily suspend updating for a while, you can just change the `Enabled` property to `false`!

There are also events (and virtual methods) to notify you when any of these properties change if you need to know about the changes.

Note

The `GameComponent` class has the same behavior as the `DrawableGameComponent` class without the extra methods and properties used for Drawing, such as `LoadContent` and `DrawOrder`.

Summary

Now you have a basic understanding of how game flow is executed and the order of operations of a normal game. You also understand components and how they interact. It is time to move into some 3D graphics before getting back to some of the more advanced 2D operations.

Index

Numbers

2D avatars, render targets, 263-265

2D graphics

- coordinate system, 13
- drawing to screen, 14-16
 - animation, 20-21
 - controlling state, 21-25
 - Draw method, 16-19
 - moving things around, 19-20
- explained, 13-14
- GraphicsDeviceManager, 14
- SpriteBatch object, 16
 - animation, 20-21
 - controlling state, 21-25
 - drawing, 16-19
 - moving things around, 19-20
- text rendering, 25-27
- Texture2D class, 14

2D sprites, 43

3D audio positioning, 356-357

3D graphics, 41-43

- coordinate systems, 44-45
- defined, 42
- matrix, 53-54
 - combining matrix transforms, 58
 - identity, 54-55
 - manipulating vectors, 59, 61
 - rotation, 56-57

- scale, 57
- translation, 55
- vectors, 46
 - addition, 47-48
 - cross product, 50
 - dot product, 49
 - negation, 49
 - point versus direction and magnitude, 46
 - scalar multiplication, 48
 - subtraction, 48
 - Vector4, 46
 - XNA Game Studio, 51, 53

3D math, 43**3D rendering**

- drawing primitives, 71-72
 - DrawIndexedPrimitives, 80, 82
 - DrawPrimitives, 78-80
 - DrawUserIndexedPrimitives, 75-78
 - DrawUserPrimitives, 72-75
 - primitive types, 71
 - vertex types, 71
- GraphicsAdapter class, 67-68
- GraphicsDevice, 69-70
 - creating, 70
 - reference devices, 71

A
acceleration data

- accelerometer, 344-346
- autorotation and, 348
- reading, 344

accelerometer, 344-346

- moving sprites, 346-347

AccelerometerSensor, 344**accessing AvatarDescription, 240-241****adding**

- custom animation to games, 283-284
- SoundEffectInstance, 357-360
- wave files, XACT, 361, 363-364

AllowOnlineCommunication, 407**AllowOnlineSessions, 407****AllowPremiumContent, 407****AllowPurchaseContent, 407****AllowTradeContent, 407****AllowUserGeneratedContent, 407****alpha blending, effect states, 211-213****AlphaTestEffect, 105, 124, 309****ambient light, 108, 186-190****analog input, 312****angles, converting to radians, 19****animated model properties, 138****animation**

- 2D graphics, 20-21
- avatars, 243-244
 - blending, 253-257, 259-260
 - functionality through interfaces, 246
 - playing multiple, 249-252
 - transforms and expressions, 245
 - updating, 244
- custom animations
 - avatars, 265-273
 - content processors, creating, 273-282

AnimationBlender, 254, 257, 259**API, storage, 387-388****App Hub membership, installing XNA Game Studio 4.0, 6-7****Apply3D, 360****ApplyChanges, 31****aspect ratio, 90****AspectRatio, 90**

assets, content pipeline, 226-234
AsyncCallback, 385
attenuation, 209
audio playback
 looping with SoundEffect, 356
 using cue, 364-366
AudioEmitter, 356
AudioEngine, 365
AudioListener, 357
autorotation, acceleration data and, 348
AvailableNetworkSession, 433
AvatarAnimation, 243-244, 250
 functionality through interfaces, 246
 transforms and expressions, 245
 updating, 244
AvatarAnimationPreset, 244
AvatarCustomAnimation, 270, 285
AvatarDescription
 accessing avatar information, 240-241
 Changed event, 243
 constructing from byte arrays, 242
 creating random, 242
AvatarExpression, 270
AvatarEye, 245
AvatarEyebrow, 245
AvatarMouth, 245
AvatarRenderer, 246-247
 detrerring current state of, 248
 drawing while loading, 247-248
avatars, 239-240
 2D avatars, render targets, 263-265
 accessing information using
 AvatarDescription, 240-241
 Changed event, 243
 constructing AvatarDescription, 242
 creating random, 242

 animations, 243-244
 blending, 253-257, 259-260
 functionality through
 interfaces, 246
 playing multiple, 249-252
 transforms and expressions, 245
 updating, 244
 custom animations, 265-273
 adding to games, 283-284
 content processors, 273-282
 updating games to use, 284-285
 drawing with AvatarRenderer,
 246-247
 detrerring current state of, 248
 while loading, 247-248
 interacting with objects, 260-263
 modifying lighting, 248-249

B

backbuffer, 65
backface culling, 63
base.Draw method, 303
BasicEffect, 105-107, 307
 effect interfaces, 121-122
 fog, 119-121
 lighting, 108-114
 textures, 114-115, 118-119
 vertex colors, 115-118
Begin method, 22
BeginMark, 299
BeginShowKeyboardInput, 316, 318-320
BeginShowMessageBox, 396
BeginShowSelector method, 387
BinaryWriter helper class, 379
BlendFactor, 147

blending, 65
 animations for avatars, 253-260
BlendState object, 142-148
 premultiplied alpha, 148-149
Blinn-Phong shading, specular lighting, 200-202
bones, 98-99
BouncingImage class, 38
build systems, tracing content, 215-216
BuildAndLoadAsset method, 232, 237
built-in shaders, cost of, 307-308
ButtonPressed method, 415
byte arrays, constructing AvatarDescription, 242

C

calculating lighting, 110
cameras, 85-86
 static cameras, 94-95
Changed event (AvatarDescription), 243
chatpad input, 313
chatpads, input, 313
choosers, Windows Phone 7, 472-476
choosing
 content processors, 220
 storage devices, 385
Clamp, 168
CleanSkeleton method, 280
ClearOptions, 153
CLR (Common Language Runtime), 2
CLR profiler, 306
color, vertex, 179-180
color value, 65
combining
 content, 226-234
 matrix transforms, 58
Common Language Runtime (CLR), 2

CompareFunction.LessEqual, 151
components, GameComponents, 38-40
connections, Xbox 360 gamepad, 333
constructing AvatarDescription from byte arrays, 242
containers, storage, 382-383
content, tracing through build system, 215-216
content importers, 223-226
Content object, 15
content pipeline, 216
 combining and building assets, 226-234
 combining what you have learned so far, 235-237
 content importers, 223-226
 content processors, 216-222
 extensions, debugging, 222
 tracing content, 215
content processors, 216-222
 choosing, 220
 creating, 273-276, 278-282
 properties, 221
ContentImporter, 230, 233
ContentManager, 237
controllers, Xbox 360 gamepad, 332-333
converting angles to radians, 19
coordinate systems, 13
 3D graphics, 44-45
CopyAbsoluteBoneTransformsTo, 99
CopyBoneTransformsFrom method, 99
CopyBoneTransformsTo method, 99
cost, 288
 of AlphaTestEffect, 309
 of built-in shaders, 307-308
 of DualTextureEffect, 308
 of EnvironmentMapEffect object, 308
 of SkinnedEffect object, 308

CPU bound, 294
CreateFile method, 378
CreateLookAt, 86, 88
CreateOrthographic, 93
CreateOrthographicOffCenter, 93
CreatePerspective method, 92
CreatePerspectiveFieldOfView, 86, 89
CreatePerspectiveOffCenter method, 92
CreateSessionDraw, 417
CreateSessionUpdate, 424
cross product, vectors, 50
cue, audio playback, 364-366
culling, backface, 63
CullMode, 166
CurrentPosition, 255, 272
custom animations
 adding to games, 283-284
 avatars, 265-273
 content processors, 273-282
 updating games to use, 284-285
custom effects, 171
 creating, 172-173
 drawing with, 177-178
 effect states, 209-210
 alpha blending, 211-213
 HLSL (High Level Shading Language), 172
 lighting, 186
 ambient lighting, 186-190
 diffuse lighting, 192-197
 emissive lighting, 198
 fog, 202-206
 point lights, 206-209
 specular lighting, 199-202
 triangle normals, 190-192

 parts of effect files, 173
 global variables, 174
 vertex structures, 174-177
 texturing, 180-183
 repeating textures, 184-186
 setting sampler states, 183-184
 vertex color, 179-180
cutouts, depth, 153

D

DancePad, 332
data, isolated storage, saving and loading, 377-379
dead zones, thumb sticks (Xbox 360 gamepad), 332
DebugCommandUI, 306
debugging content pipeline extensions, 222
DebugManager, 306
DefaultProcessor, 231
depth, cutouts, 153
depth buffer, faking shadows, 158-161
depth test, 65
DepthBufferEnable property, 154
DepthBufferFunction, 154
DepthBufferWriteEnable, 154
DepthStencilState, 149-155
determining current state of AvatarRenderer, 248
device states, 141-142
 BlendState object, 142-148
 premultiplied alpha, 148-149
 DepthStencilState, 149-155
 RasterizerState, 164-166
 stencil buffer, 161, 163-164

devices

- getting, 383-386
- storage, 382-383
 - choosing, 385
- diffuse lighting, 192-196**
 - multiple lights, 196-197
 - oversaturation, 197
- digital input, 312**
- direction versus point (vectors), 46**
- directional diffuse lighting, 196**
- directional lighting, 109-111, 193**
- DirectX, 2**
- DiscardContents, 156**
- displaying GestureSample data, 341-342**
- dot product, 49**
- downloading tools, installing XNA Game Studio 4.0, 6**
- Draw method, 16-19**
 - game loop, 36-38
 - virtual methods, Game class, 32
- DrawIndexedPrimitives, 80, 82**
- drawing**
 - 2D objects to screen, 14-16
 - animation, 20-21
 - controlling state, 21-25
 - Draw method, 16-19
 - moving things around, 19-20
 - avatars with AvatarRenderer, 246-248
 - with custom effects, 177-178
 - primitives, 71-72
 - DrawIndexedPrimitives, 80, 82
 - DrawPrimitives, 78-80
 - DrawUserIndexedPrimitives, 75-78
 - DrawUserPrimitives, 72-75
 - primitive types, 71
 - vertex types, 71
- DrawModel call, 101-102**

- DrawModeViaMeshes, 101**
- DrawOrder** property, 40
- DrawPrimitives, 78-80**
- DrawString** method, 25
- DrawUserIndexedPrimitives, 75-78**
- DrawUserPrimitives, 72-75**
- DrumKit, 332**
- DualTextureEffect, 105, 122-124, 160, 308**
- dynamic sound effects, generating, 371-374**
- DynamicSoundEffectInstance, 370-371**

E

- effect files, 173**
 - global variables, 174
 - sampler states, setting, 183-184
 - vertex structures, 174-177
- effect interfaces, 121-122**
- effect states, 209-210**
 - alpha blending, 211-213
- Effect type, 177**
- effects. See custom effects**
- Effects collection, 96**
- ElapsedGameTime, 35**
- emissive lighting, 198**
- EmissiveColor, 114**
- EnableDefaultLighting, 111**
- EndMark, 299**
- enumerating microphones, 368**
- enumerations**
 - SpriteEffects, 19
 - SpriteSortMode, 22
- EnvironmentMapAmount parameter, 127**
- EnvironmentMapEffect, 105, 124-127, 308**
- EnvironmentMapSpecular parameter, 126**
- event-based input versus polling, 312**
- expressions, avatar animation, 245**
- extensions, debugging content pipeline, 222**

F

faking shadows with depth buffer and render targets, 158-161

feedback, Windows Phones, 342, 344
vibration, 351

fields of matrix, 59

FindSession method, 431-432

FindSessionDraw, 432

FindSkeleton method, 279

fire and forget audio playback, 354-355

FlattenSkeleton method, 279

FlattenTransforms helper method, 135

FlightStick, 332

floats, 90

FMRadio, 459-460

fog, 202-206
BasicEffect, 119-121

FontName node, 25

FresnelFactor parameter, 126

FriendRequestReceivedFrom, 407

FriendRequestSentTo, 408

G

Game class, 29, 32, 413
methods, 33
ResetElapsedTime, 33
Run, 33
RunOneFrame, 33
SuppressDraw, 33
properties, 34
InactiveSleepTime, 34
IsActive, 34
IsFixedTimeStep, 34
LaunchParameters, 34
TargetElapsedTime, 34

time, 34-36

virtual methods, 32
Draw, 32
Initialize, 32
OnActivated, 33
OnDeactivated, 33
ShowMissingRequirementMessage, 33
Update, 32

game loop
Draw, 36-38
Update, 36-38

Game Studio 2.0, 3

GameComponents, 38-40

GameDefaults, 405-406

GameLobbyUpdate method, 439

gamepad (Xbox 360), 324-325
connections, 333
controllers, 332-333
moving sprites, 329-331
reading gamepad state, 325-326
gamepad buttons, 326-328
gamepad direction pad, 328
gamepad thumb sticks, 329
gamepad triggers, 329
thumb stick dead zones, 332

gamepad state, reading, 325-326
gamepad buttons, 326-328
gamepad direction pad, 328
gamepad thumb sticks, 329
gamepad triggers, 329

GamePad.GetState, 327

GamePadButtons, 326-328

GamePadDPad, 328

GamePadThumbSticks, 329

GamePadTriggers, 329

GamePadType, 332**Gamer Services, 391**

GameDefaults, 405-406

gamers, 402-405

GamerServicesComponent, 391-392

Guide class, 392

- platform-specific guide functionality, 397-402

- trial mode, 392-396

IsFriend, 407-408

Presence property, 406

Privileges property, 406-407

profiles, 402-405

GamerObject, 427**GamerPresence object, 406****GamerProfile, 404****gamers, 402-405**

GameDefaults, 405-406

IsFriend, 407-408

Presence property, 406

Privileges property, 406-407

GamerServicesComponent, 391-392**Gamertag, 426****games**

- multiplayer. *See* multiplayer games

- writing first game, 11-12

GameServices, 409**GameState, 413****GameTime object, 35****garbage collectors, 289, 291-292****general performance, 287-289**

- garbage collectors, 289, 291-292

- multithreading, 292-293

generating

- sound effects, 371-374

- vectors, 301

GeoCoordinate, 350**geometry, 43****GestureSample, 340-341**

- displaying data, 341-342

GestureType, 339-340**GetFriends, 407****GetGamerPicture, 403****GetState method, 321****GetUserStoreForApplication method, 378****global variables, effect files, 174****GPU (graphics processing unit), 62****GPU bound, 294, 307****graphics. *See* 2D graphics; 3D graphics****graphics cards, 62****graphics performance, 293, 295****graphics pipeline, 61**

- backface culling, 63

- blending, 65

- color value, 65

- graphics cards, 62

- pixel shaders, 64

- pixel tests, 64

- depth test, 65

- scissor test, 64

- stencil test, 64

- rasterization, 64

- vertex shaders, 62

- projection space, 63

- view space, 62

- world space, 62

- viewport clipping, 63

graphics processing unit (GPU), 62**graphics profiles**

- HiDef profile, 66-67

- platform capabilities, 66

- Reach profile, 66

graphics resources, tombstoning, 484-485

GraphicsAdapter class, 67-68

GraphicsDevice, 17, 30, 69-70

creating, 70

reference devices, 71

GraphicsDeviceManager, 14, 30

GraphicsProfile, 30, 66, 455

HiDef, 455, 457

Reach, 455, 457

Guide class, 392

platform-specific guide
functionality, 397

messaging and signing in, 400-402

notifications, 397

players, 398-400

trial mode, 392-396

Windows Phone 7, 397

Guide.DelayNotifications method, 397

Guitar, 333

H

HiDef profile, 3

graphics profiles, 66-67

GraphicsProfile, 455, 457

history of XNA Game Studio, 1-3

HLSL (High Level Shading Language), 172

homogeneous divide, 63

I

IAvatarAnimation, 246

identity matrix, 3D graphics, 54-55

images. See 2D graphics; 3D graphics

InactiveSleepTime, 34

IndexBuffer, 82

Initialize virtual methods (Game class), 32

input, 311

analog, 312

chatpads, 313

digital, 312

event-based input versus polling, 312

keyboards, 312-313

moving sprites, 315-316

onscreen keyboards, 316, 318-320

reading keyboard state, 313-315

mouse, 320

moving sprites, 322-324

reading mouse state, 320-322

setting position, 324

multitouch. *See* multitouch input

polling versus event-based input, 312

Xbox 360 gamepad, reading gamepad
state, 325-326

gamepad buttons, 326-328

gamepad direction pad, 328

gamepad thumb sticks, 329

gamepad triggers, 329

input vertex structures, 174

installing XNA Game Studio 4.0, 5

App Hub membership, 6-7

downloading tools, 6

XNA Game Studio Connect, 9-10

XNA Windows Phone Developer
Registration tool, 11

integers, 90

interacting with objects (avatars), 260-263

interfaces, avatar animation, 246

InverseDestinationColor, 146

InverseSourceColor, 146

InviteAccepted, 408

InviteREjectedProperty, 408

IsActive, 34

IsFixedTimeStep, 34
IsFriend, 407-408
IsFullScreen, 30
isolated storage, 375-377
 IsolatedStorageFile object, 379-380
 saving and loading data, 377-379
IsolatedStorageFile object, 379-380
IsTrial, 394
IsVisualizationEnabled, 443

J–K

jaggies, 31
joining available network sessions, 435-436
JoinSession method, 435-436

key lights, 111
keyboard state, reading, 313-315
keyboards, input, 312-313
 moving sprites, 315-316
 onscreen keyboards, 316, 318-320
 reading keyboard state, 313-315

L

launchers, Windows Phone 7, 463-472
LaunchParameters, 34
Level object, 227-228
libraries, time ruler, 306
lightDirection, 193
lighting, 186
 ambient lighting, 108, 186-190
 BasicEffect, 108-114
 calculating, 110
 diffuse lighting, 192-196
 multiple lights, 196-197
 oversaturation, 197
 directional lights, 109-111
 emissive lighting, 198

 fog, 202-206
 key lights, 111
 modifying avatars, 248-249
 point lights, 206-209
 specular color, 113
 specular highlights, 112
 specular lighting, 199
 Blinn-Phong shading, 200-202
 Phong shading, 199
 triangle normals, 190-192

LinearClamp, 168

LinearWrap, 168

LoadContent method, 15, 38-39

loading

 avatars while drawing, 247-248
 data, isolated storage, 377-379
 loose files from projects, 388-390

location service, Windows Phones, 348

 reading location data, 348-351

looping audio playback (SoundEffect), 356

M

main menu, multiplayer games, 412-416

MainMenuDraw, 416

Managed DirectX, 2

managing performance, 295-303, 305

 performance measurement tools,
 306-307

manipulating vectors with matrices, 59, 61

matrix

 3D graphics, 53-54
 combining matrix transforms, 58
 identity, 54-55
 manipulating vectors, 59, 61
 rotation, 56-57
 scale, 57
 translation, 55

- fields of, 59
- methods, 60-61
- properties of, 60
- MeasureString method, 27**
- media, 441**
 - MediaPlayer, 442-443
 - songs
 - metadata and, 443-444
 - playing, 441-442
- media enumeration, 444-448**
- media libraries, 444-448**
- MediaPlayer, 442-443, 449**
- Meshes, 96-98**
- messaging, Guide class, 400-402**
- metadata, songs and, 443-444**
- methods, 86**
 - Begin, 22
 - Draw, 16-19
 - DrawString, 25
 - Game class, 33
 - ResetElapsedTime, 33
 - Run, 33
 - RunOneFrame, 33
 - SuppressDraw, 33
 - LoadContent, 15
 - MeasureString, 27
 - ToRadians, 19
- microphones**
 - enumerating, 368
 - reading data, 369
 - recording with, 368-371
- Microsoft Cross-Platform Audio Creation Tool (XACT), 353**
- Microsoft.Xna.Framework.Net, 410**
- mipmapping, 169**
- ModelBone object, 98**
- models, 95**
 - bones, 98-99
 - Meshes property, 96-98
 - rendering, 99-103
- modifying avatar lighting, 248-249**
- mouse, 320**
 - moving sprites, 322-324
 - reading mouse state, 320-322
 - setting position, 324
- mouse state, reading, 320-322**
- mouse window handle, setting, 324**
- MouseState structure, 322**
- moving sprites**
 - with accelerometer data, 346-347
 - with keyboard input, 315-316
 - with mouse, 322-324
 - multitouch input, 337-339
 - Xbox 360 gamepad, 329-331
- multiplayer networking, 409**
- multiplayer games, 409-410**
 - joining available network sessions, 435-436
 - main menu and state management, 412-416
 - network sessions, creating, 416-417, 424
 - networking development, 410, 412
 - playing, 427-430
 - searching for available network sessions, 430-434
 - sending player invites, 438-439
 - simulating real world network conditions, 439-440
- multiple lights, diffuse lighting, 196-197**
- multisampling, 31**
- multithreading, 292-293**

multitouch input for Windows Phones, 334

- displaying GestureSample data, 341-342
- moving sprites, 337-339
- number of touch points, 336
- reading gestures from TouchPanel, 339-341
- reading TouchPanel device state, 334-336
- TouchPanel width, height, orientation, 337

MyContentProcessor class, 236

N

namespaces, table of, 3

.NET runtime, 289

network sessions

- joining, 435-436
- multiplayer games, 416, 418, 424
- searching for available, 430-434

networking, multiplayer, 409

networking development, multiplayer games, 410, 412

networks, simulating real world network conditions, 439-440

NetworkSession, 424-425

NetworkSession.Create, 432

NetworkSession.Find, 432

NetworkSession.SimulatedLatency property, 439

NetworkSession.SimulatedPacketLoss property, 440

new projects, 29-32

nonuniform scale, 57

normals, triangle normals, 190-192

notifications, Guide class, 397

NumVertices, 97

O

objects

- Content, 15
- GraphicsDevice, 17
- interacting with (avatars), 260-263
- SpriteBatch, 16
 - animation, 20-21
 - controlling state, 21-22, 24-25
 - drawing, 16-19
 - moving things around, 19-20

OnActivated virtual methods (Game class), 33

OnDeactivated virtual methods (Game class), 33

onscreen keyboard, 316, 318-320

OpaqueDataDictionary, 237

opening XACT, 360

orthographic (projection matrix), 93

otherPlayer, 398

output vertex structures, 175

oversaturation, diffuse lighting, 197

P

packetReader, 429

Pan, 356

ParentBone, 98, 251

party invites, 439

party user interfaces, 400

passes, effect files, 176

performance

- general performance, 287-289
 - garbage collectors, 289, 291-292
 - multithreading, 292-293
- graphics performance, 293, 295
- managing, 295-303, 305
 - performance measurement tools, 306-307

- perspective, projection matrix, 89-92**
- phone-specific features, handling**
 - tombstoning, 482-484**
- Phong shading, specular lighting, 199**
- photo choosers, Windows Phone 7, 472**
- photos, rendering, 473**
- Pitch, 356**
- pixel shaders, 64, 176, 182**
- pixel tests, 64**
 - depth test, 65
 - scissor test, 64
 - stencil test, 64
- platform capabilities, graphics profiles, 66**
- PlatformContents, 156**
- playback, DynamicSoundEffectInstance, 370-371**
- player invites, sending, 438-439**
- players, Guide class, 398-400**
- playing**
 - multiplayer games, 427-430
 - multiple animations, 249-252
 - songs, 441-442
 - sound effects, 353
 - SoundEffect. *See* SoundEffect
 - XACT. *See* XACT
- PlayingGameUpdate, 427-428**
- point versus direction (vectors), 46**
- point lights, 206-209**
- polling versus event-based input, 312**
- PreparingDeviceSettings, 31**
- Presence property, 406**
- PresenceMode, 406**
- PresentInterval, 69**
- PreserveContents, 156**
- primitives, drawing, 71-72**
 - DrawIndexedPrimitives, 80, 82
 - DrawPrimitives, 78-80
 - DrawUserIndexedPrimitives, 75-78
 - DrawUserPrimitives, 72-75
 - primitive types, 71
- private void CreateSession(GameType gameType), 425**
- private void CreateSessionDraw(), 418**
- Privileges property, 406-407**
- ProcessAnimation, 282**
- processAnimations helper methods, 136**
- profilers, 307**
- profiles, 402-405**
 - GameDefaults, 405-406
 - IsFriend, 407-408
 - Presence property, 406
 - Privileges property, 406-407
- projection matrix, 88**
 - orthographic, 93
 - perspective, 89-92
- projection space, 63**
- projects**
 - loading loose files from, 388-390
 - new projects, 29-32
 - recreating on Xbox, 380-382
 - XACT, creating new, 360
- properties**
 - content processors, 221
 - Game class, 34
 - InactiveSleepTime, 34
 - IsActive, 34
 - IsFixedTimeStep, 34
 - LaunchParameters, 34
 - TargetElapsedTime, 34
 - matrix, 60

Q–R

radians, converting to angles, 19
rasterization, 64
RasterizerState, 164-166
Reach profile, 3

- graphics profiles, 66
- GraphicsProfile, 455, 457

reacting to tombstoning, 480-482
reading

- acceleration data, 344
- gamepad state, 325-326
 - gamepad buttons, 326-328
 - gamepad direction pad, 328
 - gamepad thumb sticks, 329
 - gamepad triggers, 329
- gestures from TouchPanel, multitouch input, 339-341
- keyboard state, 313-315
- location data, 348-351
- microphone data, 369
- mouse state, 320-322
- TouchPanel device state, 334-336

ReceiveData, 429
recording with microphones, 368-371
reference devices, GraphicsDevice, 71
reference types, 288
ReferenceStencil value, 163
render targets

- 2D avatars, 263-265
- faking shadows, 158-161

rendering

- 3D. *See* 3D rendering
- models, 99-103
- photos, 473
- targets, 155-158

- text, 25-27
- video, 448-450
- visualization data, 451-453

RenderScene helper, 160
RenderState object, 141
RenderTarget2D, 155, 158
RenderTargetUsage options, 156
repeating textures, 184-186
ResetElapsedTime, 33, 36
ReverseSubtract value, 145
rotation matrix, 3D graphics, 56-57
Run, 33
RunOneFrame, 33

S

Sampler, 182
sampler states, 166-169

- setting in effect files, 183-184
- texture types, 169

SampleRate, 372
SamplerState object, 141
saving data, isolated storage, 377-379
scale matrix, 3D graphics, 57
scissor test, 64
screens, drawing 2D objects to, 14-16

- animation, 20-21
- controlling state, 21-22, 24-25
- Draw method, 16-19
- moving things around, 19-20

searching for available network sessions, 430-434
SearchQuery, 471
SecularColorPower, 201
SendDataOptions, 429
sending player invites, 438-439

sensors, Windows Phones, 342, 344

sessions, creating, 423

shader models, 172

shaders

built-in, cost of, 307-308

pixel shaders, 176, 182

vertex, 182

vertex shaders, 175

shading

Blinn-Phong, 200-202

Phong, 199

shadows, faking with depth buffer and render targets, 158-161

ShowGameInvite, 400

ShowMarketplace, 394

ShowMissingRequirementMessage virtual methods (Game class), 33

SignedIn event, 403

SignedInGamers, 241, 403

signing in Guide class, 400-402

SimulateTrialMode, 393

simulating real world network conditions, 439-440

Size node, 25

SkinnedEffect, 105, 127-140, 308

songs

metadata and, 443-444

playing, 441-442

sound effects

generating, 371-374

playing, 353

SoundEffect. *See* SoundEffect

XACT. *See* XACT

recording audio with microphones, 368-371

SoundEffect, 354

3D audio positioning, 356-357

adding SoundEffectInstance to games, 357-360

adjusting Pitch, Pan, and Volume, 356

fire and forget, 354-355

loading from files, 354

looping audio playback, 356

SoundEffectInstance, 355

SoundEffectInstance, 355

adding to games, 357-360

Spacing node, 25

specular color, 112-113

specular highlights, 112

specular lighting, 199

Blinn-Phong shading, 200-202

Phong shading, 199

SpecularColor, 113

SpecularLightColor, 201

SpriteBatch object, 16

animation, 20-21

controlling state, 21-25

drawing, 16-19

moving things around, 19-20

SpriteEffects enumeration, 19

sprites, moving

with accelerometer data, 346-347

with keyboard input, 315-316

with mouse, 322-324

with multitouch input, 337-339

with Xbox 360 gamepad, 329-331

SpriteSortMode enumeration, 22

StartIndex, 97

state management, multiplayer games, 412-416

StateBlock object, 141

states

- controlling (2D graphics), 21-25
- device state
 - BlendState object. *See* BlendState object
 - DepthStencilState, 149-155
- device states, 141-142
 - RasterizerState, 164-166
 - stencil buffer, 161, 163-164
- sampler states, 166-169
 - texture types, 169
- static cameras, 94-95**
- stencil buffer, 161, 163-164**
- stencil test, 64**
- storage, 375**
 - API, 387-388
 - containers, 382-383
 - devices, 382-383
 - choosing, 385
 - getting, 383-386
 - isolated storage, 375-377
 - IsolatedStorageFile object, 379-380
 - saving and loading data, 377-379
 - loading loose files from your project, 388-390
 - recreating projects on Xbox, 380-382
- StorageContainer object, 388**
- streaming XACT, 366-368**
- StreamReader class, 226**
- Style node, 26**
- SupportedOrientations, 31**
- SuppressDraw, 33**
- SurfaceFormatColor, 31**
- SynchronizeWithVerticalRetrace, 31**
- synchronizing vertical retrace, 31**
- SystemLink, 410**

T

-
- Tag property (models), 96**
 - TargetElapsedTime, 34-35**
 - targets, rendering, 155-158**
 - techniques, effect files, 176**
 - tex2D function, 182**
 - text, rendering, 25-27**
 - Texture parameter, 126**
 - Texture2D, 14, 169**
 - TextureContent, 229**
 - TextureCoordinate values, 185**
 - TextureCube, 169**
 - textures**
 - AlphaTestEffect, 124
 - BasicEffect, 114-115, 118-119
 - DualTextureEffect, 122-124
 - repeating, 184-186
 - sampler states, 166
 - texturing custom effects, 180-183**
 - repeating textures, 184-186
 - setting sampler states, 183-184
 - thumb sticks, dead zones (Xbox 360 gamepad), 332**
 - time, game class, 34-36**
 - time ruler library, 306**
 - TimeRuler, 296**
 - TimeSpan, 351**
 - ToggleFullscreen, 31**
 - tombstoning, 476, 479**
 - graphics resources, 484-485
 - handling with phone-specific features, 482-484
 - reacting to, 480-482
 - tools**
 - downloading for installing XNA Game Studio 4.0, 6

Windows Phone Developer
Registration tool, 11

ToRadians method, 19

TotalGameTime, 35

touch points, multitouch input, 336

TouchCollection, 335

TouchLocation, 335, 337

TouchPanel

device state, reading, 334-336

reading gestures from, 339-341

width, height, and orientation, 337

TouchPanelCapabilities, 336

tracing content through build system,
215-216

transforms, avatar animation, 245

translation matrix, 3D graphics, 55

trial mode, Guide class, 392-396

triangle normals, 190-192

U

Update, 35

game loop, 36-38

virtual methods, Game class, 32

UpdateBoneTransforms, 273

UpdateOrder property, 40

updating

avatar animation, 244

games to use custom animation,
284-285

user input. *See* input

V

ValidateMesh call, 135

value types, 289

variables, global, 174

vector addition, 47-48

vector cross product, 50

vector dot product, 49

vector negation, 49

vector scalar multiplication, 48

vector subtraction, 48

Vector2 class, 18

Vector3, 46, 51-53

Vector4, 46

vectors, 46

3D graphics, 46

addition, 47-48

cross product, 50

dot product, 49

negation, 49

point versus direction and
magnitude, 46

scalar multiplication, 48

subtraction, 48

Vector4, 46

XNA Game Studio, 51, 53

generating, 301

manipulating with matrices, 59, 61

vertex buffer, 116-117

vertex color, 179-180

BasicEffect, 115-118

vertex shaders, 62, 175, 182

projection space, 63

view space, 62

world space, 62

vertex structures, effect files, 174-177

vertex types, drawing primitive types, 71

VertexBuffer, 78-79

VertexDeclaration, 116

VertexElement, 116

VertexOffset, 97
 vertical retrace, synchronizing, 31
VibrateController, 351
 vibration, Windows Phones feedback, 351
 video, rendering, 448-450
VideoPlayer object, 449
 view matrix, 87-88
 view space, 62
 viewport clipping, 63
 virtual methods, Game class, 32
 Draw, 32
 Initialize, 32
 OnActivated, 33
 OnDeactivated, 33
 ShowMissingRequirementMessage, 33
 Update, 32
 visualization data, rendering, 451-453
 visualizations, 451-453
 visualizers, 442
 Volume, 356

W

wave files, adding to XACT projects, 361, 363-364
 Wheel, 332
 Windows desktop runtime, 291
 Windows Game SDK, 1
 Windows Phone 7
 choosers, 472-476
 FMRadio, 459-460
 Guide class, 397
 launchers, 463-472
 writing first game, 12
 Windows Phone Developer Registration tool, installing XNA Game Studio 4.0, 11

Windows Phones

acceleration data using accelerometer, 344-346
 feedback, 342, 344
 vibration, 351
 location service, 348
 reading location data, 348-351
 multitouch input, 334
 displaying GestureSample data, 341-342
 moving sprites, 337-339
 number of touch points, 336
 reading gestures from TouchPanel, 339-341
 reading TouchPanel device state, 334-336
 TouchPanel width, height, and orientation, 337
 sensors, 342, 344

WireFrame, 165

world space, vertex shaders, 62

Wrap, 169

wrap texture, 168

writing first game, 11-12

X-Z

X axis, 13

XACT (Microsoft Cross-Platform Audio Creation Tool), 353, 360

adding wave files to projects, 361, 363-364
 creating new projects, 360
 opening, 360
 sound playback using cue, 364-366
 streaming, 366-368

Xbox, recreating projects, 380-382**Xbox 360**

- chatpad input, 313
- devices, 382
- game data, 382
- shader models, 172
- writing first game, 11

Xbox 360 gamepad, 324-325

- connections, 333
- controllers, 332-333
- moving sprites, 329-331
- reading gamepad state, 325-326
 - gamepad buttons, 326-328
 - gamepad direction pad, 328
 - gamepad thumb sticks, 329
 - gamepad triggers, 329
- thumb stick dead zones, 332

XNA Game Studio

- 3D graphics, 41-42
- vectors, 51, 53

XNA Game Studio 4.0, installing, 5

- App Hub membership, 6-7
- downloading tools, 6
- Windows Phone Developer Registration tool, 11
- XNA Game Studio Connect, 9-10

XNA Game Studio Connect, installing XNA Game Studio 4.0, 9-10**XNA Game Studio Device Center, 10****Y axis, 13**