



Learn by doing: less theory, more results

XNA 4.0 Game Development by Example

Create exciting games with Microsoft XNA 4.0

Beginner's Guide

Kurt Jaegers

[PACKT]
PUBLISHING

XNA 4.0 Game Development by Example

Beginner's Guide

Create exciting games with Microsoft XNA 4.0

Kurt Jaegers



BIRMINGHAM - MUMBAI

XNA 4.0 Game Development by Example

Beginner's Guide

Copyright © 2010 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2010

Production Reference: 1180910

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-84969-066-9

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Credits

Author

Kurt Jaegers

Reviewers

Rodolfo Rigo Calabrezi

Dustin Heffron

Alexandre Ribeiro de Sá

Acquisition Editor

David Barnes

Development Editor

Wilson D'souza

Technical Editor

Dayan Hyames

Copy Editor

Laxmi Subramanian

Indexers

Hemangini Bari

Monica Ajmera Mehta

Editorial Team Leader

Aditya Belpathak

Project Team Leader

Ashwin Shetty

Proofreaders

Jacqueline McGhee

Chris Smith

Graphics

Nilesh R. Mohite

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Kurt Jaegers is a database administrator by day, and a long-time hobbyist game developer, having built games for everything from the Commodore 64 to the Xbox 360. He is the owner of `xnaresources.com`, one of the earliest XNA-focused tutorial websites.

I would like to thank my wife Linda for proofreading the earliest versions of this book, as well as my brother Jason for providing the graphics for Flood Control, Asteroid Belt Assault, and the map tiles for Gemstone Hunter, along with the title screens for all of the games.

I would also like to thank the team at Packt Publishing for guiding me through my first book-authoring experience.

About the Reviewers

Rodolfo Rigo Calabrezi has worked in desktop publishing and web advertising campaigns, PC and mobile applications, critical systems and web 2.0 frontend. In the 90s, he made games for the MSX and TRS-Color platforms. In 2004, he worked as the CTO at Eleven Cells, a game development studio. He built an impressive professional network. He was a member of IGDA, working directly on programming, game design, art production, and QA. He teaches and gives lectures in schools about game development. He is in charge of Quantix Games, a Brazilian game development studio. He knows PHP, HTML, CSS, JavaScript, C/C++, C#, HLSL, engine architecture, graphics pipeline, ODE, Ageia, Direct3D, OpenGL, Jupiter, CIPHER, PopCap, HGE, XNA, and Flash and is learning UDK and Unity. For DCC, he works with Photoshop, Flash, modo, Reason, and Nuendo, and is learning XSI and Maya. He is familiar with the following platforms—PC, PocketPC, Amiga, Xbox 360, and PS2, and is new to iPhone and iPad.

He is interested in development for other console platforms like PS3, Nintendo Wii, and DS.

Dustin Heffron is an avid gamer and programmer. He graduated in 2010 from Penn State University with a Bachelor's of Science degree in Computer Science. He has had 7+ years of experience programming in various languages and environments, including all C-based languages and most .NET languages. He has worked with XNA since 2007 and applied it in a variety of military and video game related projects.

Currently, Dustin works for the Applied Research Lab at Penn State as a Graphical User Interface programmer and Acid Lab Studios as Lead Programmer. He is planning on releasing his first episodic commercial game, "The Scarab Gauntlet" in late 2010, with additional episodes to follow.

I would like to thank my boss Mitch Lebold for giving me the opportunity to research and apply XNA to a variety of real-world projects. Without this experience, I wouldn't be the developer I am today. I would also like to thank Andy Strauch, producer and founder of Acid Lab Studios for giving me the opportunity to program such a great game and being patient with me during the busier times of life. Lastly, I would like to thank Packt Publishing for letting me review this insightful book, which I've greatly enjoyed reading.

Alexandre Ribeiro de Sá is a Producer, Director, Programmer and self-proclaimed frustrated artist. Alexandre was present in many game development community websites in Brazil giving support in translation and writing articles about software development and CG. Today as a producer, programmer, and sometimes as a game designer in his own game studio, the Vortex Game Studios in São Paulo, Brazil, he works with outsourcing and in his own games titles for computer and video game consoles.

Table of Contents

Preface	1
Chapter 1: Introducing XNA Game Studio	7
Overview of the games	8
System requirements	9
Installing XNA Game Studio	10
Time for action – installing XNA Game Studio	11
Building your first game	12
Time for action – creating a new Windows game project	12
Anatomy of an XNA game	13
The declarations area	14
Time for action – adding variables to the class declaration area	14
The Game1 class constructor	15
The Initialize() method	16
Time for action – customizing the Initialize() method	16
The LoadContent() method	16
Time for action – creating the squareTexture	17
The Update() method	17
Time for action – coding Update() for SquareChase	18
The Draw() method	19
Time for action – draw SquareChase!	20
Time for action – play SquareChase!	20
Summary	22
Chapter 2: Flood Control – Underwater Puzzling	23
Designing a puzzle game	24
Time for action – set up the Flood Control project	24
Introducing the Content Pipeline	25
Time for action – reading textures into memory	26
Sprites and sprite sheets	27
Classes used in Flood Control	29

The GamePiece class	30
Identifying a GamePiece	30
Time for action – build a GamePiece class – declarations	30
Creating a GamePiece	33
Time for action – building a GamePiece class: constructors	33
Updating a GamePiece	33
Time for action – GamePiece class methods – part 1 – updating	33
Rotating pieces	34
Time for action – GamePiece class methods – part 2 – rotation	35
Pipe connectors	36
Time for action – GamePiece class methods – part 3 – connection methods	36
Sprite sheet coordinates	37
Time for action – GamePiece class methods – part 4 – GetSourceRect	37
The GameBoard class	39
Time for action – create the GameBoard.cs class	39
Creating the game board	40
Time for action – initialize the game board	40
Updating GamePieces	41
Time for action – manipulating the game board	41
Filling in the gaps	42
Time for action – filling in the gaps	42
Generating new pieces	43
Time for action – generating new pieces	43
Water filled pipes	44
Time for action – water in the pipes	44
Propagating water	45
Time for action – making the connection	45
Building the game	47
Declarations	47
Time for action – Game1 declarations	47
Initialization	48
Time for action – updating the Initialize() method	48
The Draw() method – the title screen	49
Time for action – drawing the screen – the title screen	49
The Draw() method – the play screen	50
Time for action – drawing the screen – the play screen	50
Keeping score	52
Time for action – scores and scoring chains	52
Input handling	53
Time for action – handling mouse input	54
Letting the player play!	55

Time for action – letting the player play	55
Play the game	56
Summary	56
Chapter 3: Flood Control – Smoothing Out the Rough Edges	57
Animated pieces	58
Classes for animated pieces	58
Time for action – rotating pieces	58
Time for action – falling pieces	60
Time for action – fading pieces	61
Managing animated pieces	62
Time for action – updating GameBoard to support animated pieces	63
Fading pieces	66
Time for action – generating fading pieces	66
Falling pieces	67
Time for action – generating falling pieces	67
Rotating pieces	68
Time for action – modify Game1 to generate rotating pieces	68
Calling UpdateAnimatedPieces()	68
Time for action – updating Game1 to update animated pieces	68
Drawing animated pieces	69
Time for action – update Game1 to draw animated pieces	69
SpriteBatch overloads	72
SpriteFonts	73
Time for action – add SpriteFonts to Game1	74
Score display	75
Time for action – drawing the score	75
ScoreZooms	76
Time for action – creating the ScoreZoom class	76
Time for action – updating and displaying ScoreZooms	77
Adding the GameOver game state	79
Time for action – game over	80
The flood	81
Time for action – tracking the flood	81
Displaying the flood	82
Time for action – displaying the flood	83
Difficulty levels	84
Time for action – adding difficulty levels	84
Summary	87
Chapter 4: Asteroid Belt Assault – Lost in Space	89
Creating the project	89

Time for action – creating the Asteroid Belt Assault project	90
Another definition for "sprite"	91
Building the Sprite class	91
Time for action – declarations for the Sprite class	92
Time for action – Sprite constructor	93
Time for action – basic Sprite properties	93
Time for action – animation and drawing properties	94
Collision detection	96
Time for action – supporting collision detection	97
Animation and movement	98
Time for action – adding animation frames	98
Time for action – updating the Sprite	98
Time for action – drawing the Sprite	99
A sprite-based star field	100
Time for action – creating the StarField class	100
Time for action – updating and drawing the StarField	102
Time for action – viewing the StarField in action	102
Animated sprites – asteroids	104
Time for action – building the AsteroidManager class	104
Positioning the asteroids	106
Time for action – positioning the asteroids	107
Time for action – checking the asteroid's position	109
Time for action – updating and drawing asteroids	110
Colliding asteroids	111
Time for action – bouncing asteroids – part 1	112
Time for action – bouncing asteroids – part 2	115
Player and enemy shots	116
Time for action – adding the ShotManager class	116
Time for action – firing shots	117
Time for action – updating and drawing shots	118
Adding the player	119
Time for action – creating the PlayerManager class	119
Handling user input	121
Time for action – handling user input	121
Time for action – updating and drawing the player's ship	123
Enemy ships	126
Time for action – creating the Enemy class	127
Time for action – waypoint management	128
Time for action – enemy update and draw	130
The EnemyManager class	132
Time for action – creating the EnemyManager class	132

Managing waypoints	134
Time for action – setting up the EnemyManager class	134
Time for action – spawning enemies	135
Time for action – updating and drawing the EnemyManager	137
Summary	139
Chapter 5: Asteroid Belt Assault – Special Effects	141
Explosion effects	141
Expanding on sprites – particles	142
Time for action – constructing the Particle class	142
Time for action – updating and drawing particles	144
Particle explosions	145
Time for action – the ExplosionManager class	146
Time for action – creating explosions	147
Time for action – updating and drawing explosions	149
The collision manager	151
Time for action – creating the CollisionManager class	151
Handling collisions	152
Time for action – player shot collisions	153
Time for action – player collisions	154
Time for action – using the CollisionManager class	156
Sound effects	157
Generating and finding sound effects	157
Sound in XNA	157
Time for action – building a sound effects manager	158
Time for action – using the SoundManager class	160
The game structure	161
Time for action – structuring the game	162
Time for action – drawing the game structure	166
Summary	168
Chapter 6: Robot Rampage – Multi-Axis Mayhem	169
Statics, statics, everywhere	170
Time for action – creating the Robot Rampage project	170
A world larger than the screen	171
Defining a camera	172
Time for action – creating the Camera class	172
World-aware sprites	176
Time for action – building a new Sprite class	177
Visualizing the view	184
Time for action – viewing the Sprite and Camera classes in action	184
The game world – tile-based maps	187

Tiles	188
The tile map	188
Time for action – creating the TileMap class	189
Map squares	191
Time for action – dealing with map squares	191
Dealing with tiles	194
Time for action – handling tiles	194
Drawing the map	196
Time for action – drawing the tile map	196
Generating a random map	198
Time for action – random wall placement	198
Adding the player	200
Building the Player class	201
Time for action – building the Player class	201
Moving around the world	205
Time for action – handling input	206
Staying in bounds	209
Time for action – staying in bounds	210
Running into tiles	212
Time for action – accounting for walls	212
Summary	216
Chapter 7: Robot Rampage – Lots and Lots of Bullets	217
Visual effects	217
Revisiting particles	218
Time for action – the Particle class	218
The EffectsManager class	221
Time for action – the EffectsManager class	221
Time for action – building explosions	223
Time for action – spark effects	226
Adding weaponry	228
The WeaponManager class	228
Time for action – beginning the WeaponManager class	228
Weapon upgrades	232
Time for action – new weapons	233
Shot to map collisions	236
Time for action – shots colliding with tiles	236
Power-ups	238
Time for action – power-ups	239
We still have a problem...	243
Path finding	244

The A* path-finding algorithm	244
Implementing A*	248
Time for action – the PathNode class	248
Time for action – beginning the implementation of A*	251
Time for action – finding the path	253
Time for action – adjacent squares	256
Safely placing power-ups	261
Time for action – updating the WeaponManager class	261
Player goals	262
Computer terminals	263
Time for action – building a computer terminal	263
Spawning computer terminals	265
Time for action – the GoalManager class	266
Enemy robots	272
Enemy basics	272
Time for action – building the Enemy class	273
Moving enemies	275
Time for action – enemy AI methods	275
The enemy manager	277
Time for action – the enemy manager	277
Updating the WeaponManager	281
Time for action – destroying enemies	281
Game structure	283
Time for action – the GameManager class	283
Keeping score	285
Time for action – awarding points	285
Updating Game1	285
Time for action – updating the Game1 class	285
Summary	290
Chapter 8: Gemstone Hunter – Put on Your Platform Shoes	291
Borrowing graphics	292
Time for action – creating projects	292
A more advanced tile engine	294
Time for action – the MapSquare class	294
Rebuilding the camera	296
Time for action – the Camera class	296
Constructing the tile engine	299
Time for action – the TileMap class – part 1	299
Time for action – the TileMap class – part 2	302
Drawing the tile map	306

Time for action – the TileMap class – part 3	306
Time for action – adding the tile map to the game project	308
The map editor project	310
Creating the map editor project	310
Time for action – creating the level editor project	311
Adding a form	313
Time for action – adding a form	313
Time for action – adding event handlers	318
Filling Out Our Form	319
Time for action – creating the menu bar	320
Time for action – tile selection controls	321
Time for action – scroll bars	325
Time for action – final controls	327
Updating the Game1 class	331
Time for action – updating Game1	331
Time for action – the Game1 Update method	333
Connecting the form to the game	336
Time for action – completing the editor – part 1	336
Time for action – fixing the scrolling delay	340
Loading and Saving Maps	341
Time for action – implementing loading and saving	341
Passability	344
Map codes	345
One last issue	346
Time for action – handling the FormClosed event	346
Summary	347
Chapter 9: Gemstone Hunter – Standing on Your Own Two Pixels	349
Animation strips	349
Time for action – building the Animation class	350
Animated game objects	354
Time for action – building the GameObject class – Part 1	354
Drawing, animation, and movement	357
Time for action – building the GameObject class – part 2	357
Map-based collision detection	360
Time for action – building the GameObject class – part 3	361
The Player	364
Time for action – creating the Player class	364
Running...	367
Time for action – overriding the Update() method – part 1	367
...and jumping	370

Time for action – overriding the Update() method – part 2	370
Staying on the screen	371
Time for action – repositioning the camera	371
Loading levels	372
The LevelManager class	372
Time for action – building the LevelManager class	373
Gemstones	377
Time for action – building the Gemstone class	377
Scoring	380
Time for action – implementing score tracking	380
Enemies	382
Time for action – summoning up the zombies	382
Player-enemy interaction	386
Time for action – interacting with zombies	386
Level transitions	388
Time for action – supporting map transitions	388
Processing other codes	390
Time for action – handling codes	390
Game structure	391
Time for action – implementing game states	391
Summary	395
Index	397

Preface

The Microsoft XNA Framework provides a powerful set of tools to allow development teams of any size, from the individual developer to larger independent teams, to rapidly develop high performance quality games for multiple Microsoft-related platforms.

This book will present a series of video games, utilizing XNA Game Studio and the XNA Framework to delve into the world of 2D game development targeting the Microsoft Windows environment. We will utilize XNA's 2D graphics capabilities to present our games to the player, and look at the fundamental systems behind several game design challenges, such as path-finding, collision detection, special effects, and more.

Each of the four games in this book cover a new gaming style, and introduce progressively more advanced techniques and systems to provide a foundation for bringing your own creations to life.

What this book covers

Chapter 1, Introducing XNA Game Studio begins by looking at the history of the XNA Framework and its predecessors, and installing the Windows Phone Development Tools package that includes the version 4.0 release of the XNA tools. We wrap up this chapter by looking at the building blocks of an XNA game and putting together an XNA mini-game called SquareChase.

Chapter 2, Flood Control – Underwater Puzzling introduces a board-based puzzle game called Flood Control. We introduce the XNA Content Pipeline, and build a recursive function to determine the state of the game board while playing.

Chapter 3, Flood Control – Smoothing out the Rough Edges refines and completes the Flood Control game, adding animated rotation, movement, and fading of game pieces. We will implement a scoring system and cover displaying text to the screen.

Chapter 4, Asteroid Belt Assault – Lost in Space begins developing our second game. This time we put together a space-based shooter. We will create a basic moving star field using a simple particle system, and introduce frame-based sprite animation. We will add moving asteroids to our star field and examine how to detect collisions between asteroids and make them respond realistically. We add a player-controlled spaceship, and enemies that can fly across the screen following pre-defined waypoints.

Chapter 5, Asteroid Belt Assault – Special Effects wraps up Asteroid Belt Assault. We implement collision detection between the player, enemies, asteroids, and bullets and create particle-based explosions. Finally, we will look at loading and playing sound effects to bring life to our in-game events.

Chapter 6, Robot Rampage – Multi-Axis Mayhem begins the construction of a tank-based game in which the player can move and fire independently using either an Xbox 360 controller or the keyboard. We build a tile-map based game world and a camera class to view a screen-sized area of the larger world, and implement player collision with the walls of the tile map.

Chapter 7, Robot Rampage – Lots and Lots of Bullets completes Robot Rampage by expanding on our particle-based explosion system, and adding enemies, player goals, and weapon upgrades to our tile map. We allow the player to fire at the enemies with several different weapons. Finally, we create an implementation of the A* path-finding algorithm to allow the enemy tanks to track down the player.

Chapter 8, Gemstone Hunter – Put on your Platform Shoes introduces a side-scrolling, jump-and-run platform game. We start by evolving our tile-based mapping system to allow multiple layers of tiles, and look at combining XNA and Windows Forms to produce a map editor for Gemstone Hunter. As part of this process, we will look at building more complex solutions that contain multiple projects, and separate our game's tile engine into a Game Library project.

Chapter 9, Gemstone Hunter – Standing on Your Own Two Pixels concludes the Gemstone Hunter project by examining an alternative method for frame-based sprite animation using player and enemy graphics from the XNA Platform Starter Kit. We implement platform physics and bring the game together by loading levels and reacting to their embedded code values.

What you need for this book

In order to install and use the Microsoft XNA 4.0 tools, you will need a Windows PC with either Microsoft Windows Vista or Microsoft Windows 7, and a video card supporting DirectX 9 or later. Shader Model 1.1 is required for XNA, but it is highly recommended that your video card support Shader Model 2.0 or later, as many of the XNA samples available online require 2.0 support.

Who this book is for

If you are an aspiring game developer who wants to take a shot at creating games for the Microsoft Windows platform with the XNA Framework, then this book is for you. Using this book, you can get started with creating games without any game development experience. A basic knowledge of C# would be helpful to kick start your game development, but it's not essential.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

- 1.** Action 1
- 2.** Action 2
- 3.** Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

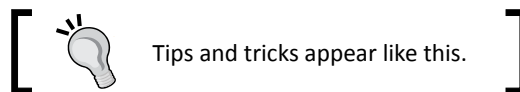
You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "If the mouse reports that the left button is pressed, the code checks with the `currentSquare` object by calling its `Contains()` method to determine if the mouse's coordinates fall within its area."

A block of code is set as follows:

```
spriteBatch.Begin();  
spriteBatch.Draw(  
    squareTexture,  
    currentSquare,  
    colors[playerScore % 3]);  
spriteBatch.End();
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Run your game by clicking on **Start Debugging** from the **Debug** menu".



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

**Downloading the example code for this book**

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing XNA Game Studio

With the release of Windows 95, Microsoft also introduced the DirectX Application Programming Interface (API), which allowed Windows-based applications to integrate closely, in a standard way, with the graphics hardware available on the system. Prior to DirectX, most PC game development targeted MS-DOS, as Windows-based graphics were too slow for most gaming needs.

Although faster, working with the DirectX API could be challenging. The DirectX Software Development Kit (SDK) is targeted at C++, with no official support for other languages. The developer is also faced with large volumes of background work to get a DirectX project to the point where he can display images on the screen before ever considering the logic of the game itself.

In 2002, Microsoft released Managed DirectX as an interface to the API from its new .NET development environment. The .NET Framework consists of a set of code libraries to perform common programming tasks, and the Common Language Runtime (CLR) which allows code written in the various .NET languages (including Visual Basic .NET and C#) to be compiled into common runtime code. In order to support devices such as Windows Mobile phones, a subset of the .NET Framework was released, called the .NET Compact Framework. The .NET CF, as it is often abbreviated, removed non-essential components of the full Framework in the interest of saving storage space on handheld devices.

While Managed DirectX 2.0 was still in the beta phase, the project was cancelled, and Microsoft XNA was introduced in its place. XNA consists of the XNA Framework, a set of code libraries to perform common graphics, sound, and other game related tasks, and XNA Game Studio, an extension of the Visual Studio C# interface that includes a number of project templates to make use of the XNA Framework.

The XNA project templates include an integrated game loop, easy to use (and fast) methods to display graphics, full support for 3D models, and simple access to multiple types of input devices.

In addition to Windows games, XNA allows deployment to both the Xbox 360, the Zune handheld media player (with XNA 3.1) and Windows Phone 7 Series phones (with XNA 4.0). For the first time, a game console manufacturer has released a supported method for individual game developers to create (and sell!) content for their game console. Microsoft has even established the Xbox Indie Games system on Xbox Live to allow you to sell your creations to the world.



What does XNA stand for, anyway?

According to the developers, XNA is an acronym for "XNA's Not Acronymed".

In this introductory chapter you will:

- ◆ Look at an overview of the games presented in this book
- ◆ Download and install XNA Game Studio
- ◆ Create a new Windows Game project
- ◆ Modify the default Windows Game template to build your first XNA game

Overview of the games

Many beginning developers make the mistake of attempting to tackle far too large a project early on. Modern blockbuster video games are the result of the efforts of hundreds of programmers, designers, graphic artists, sound effects technicians, producers, directors, actors, and many other vocations, often working for years to create the game.

That does not mean that the efforts of a solo developer or small team need to be dull, boring, and unplayable. This book is designed to help you develop a solid understanding of 2D game development with XNA Game Studio. By the time you have completed the projects in this book, you will have the knowledge necessary to create games that you can complete without an army of fellow game developers at your back.

In this chapter, you will build your first XNA mini game, chasing squares around the screen with your mouse cursor. In subsequent chapters the following four more detailed games are presented:

Flood Control: An explosion in one of the research laboratories has cracked the pressure dome protecting your underwater habitat. Work quickly to construct a series of pipes to pump water out of the habitat before it floods. Flood Control is a board-based puzzle game with simple game mechanics and slowly increasing difficulty.

Asteroid Belt Assault: After being separated from your attack fleet in Hyper Space, you find yourself lost in an asteroid field without communications or navigation systems. Work your way through the chaos of the asteroid belt while combating alien pilots intent upon your destruction. A vertically scrolling space shooter, Asteroid Belt Assault introduces scrolling backgrounds, along with player and computer controlled characters.

Robot Rampage: In the secret depths of a government defence facility, a rogue computer has taken control of robotic factories across the world, constructing an army of mechanical soldiers. Your mission—infiltrate these factories and shut down their network links to break the computer's control. A multi-axis shooter utilizing both of the analog control sticks on the Xbox 360 gamepad controller, Robot Rampage generates and manages dozens of on-screen sprites and introduces world map construction.

Gemstone Hunter: Explore the Australian wilderness, abandoned mines and ancient caves in a search for fabulous treasures. In Gemstone Hunter you will construct a classic platform-style game, including a Windows Forms-based level editor and a multi-map "world" to challenge the player.

The games are each presented over two chapters. In the first chapter, the basics are implemented to the point where the game is playable. In the second chapter, features and polish are added to the game.

Each game introduces new concepts and expands on topics covered in the previous games. At the end of each game chapter, you will find a list of exercises challenging you to use your newly gained knowledge to enhance previous games in the book.

We will focus on Windows as our platform for the games presented in this book. That said, the code presented in this book requires very little in the way of changes for other XNA platforms, generally only requiring implementation of platform-specific controls (gamepads, touch screen, and so on) and consideration of the differences in display sizes and orientation on non-Windows devices.

System requirements

In order to develop games using XNA Game Studio, you will need a computer capable of running both Visual C# 2010 Express and the XNA Framework extensions. The general requirements are:

Component	Minimum requirement	Notes
Operating System	Windows Vista SP2 or Windows 7 (All editions except Starter)	As of XNA 4.0, Windows XP is no longer officially supported.

Component	Minimum requirement	Notes
Graphics card	Shader Model 1.1 support DirectX 9.0 support	Microsoft recommends Shader Model 2.0 support as it is required for many of the XNA Starter Kits and code samples. The projects in this book similarly require Shader Model 2.0 support.
Optional requirements		
Windows Phone	DirectX 10 or later, Compatible Video Card	Development tools include a Windows Phone emulator to test applications without deployment to a physical device.
Zune platform	Zune Software 3.0 or higher, Visual C# 2008, XNA Game Studio 3.1	Only required if you plan to deploy games to a Zune handheld device. Zune development is supported under XNA 3.1.
Xbox Live	Xbox Live Silver membership, XNA Creator's Club Premium membership	Xbox Live Silver is free. The XNA Creator's Club Premium membership costs \$49 for 4 months or \$99 for 1 year.



HiDef vs. Reach

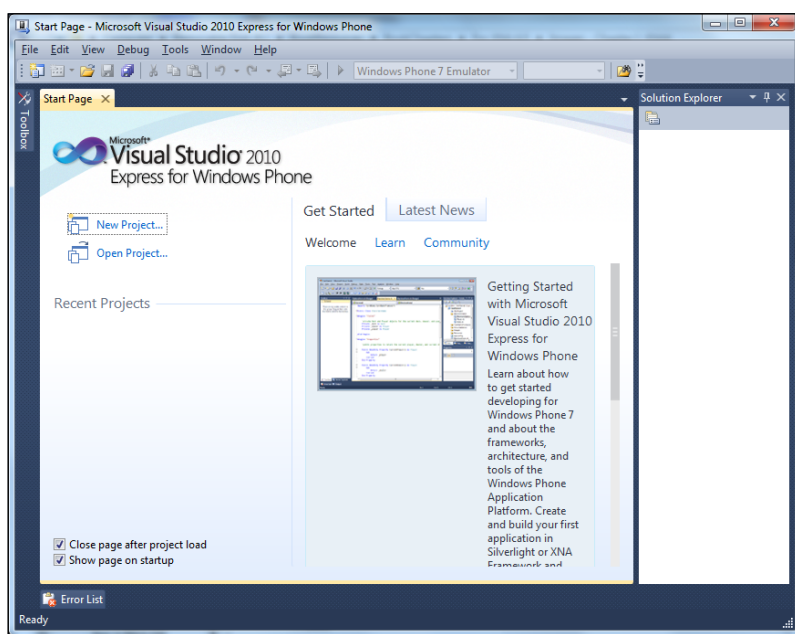
As of version 4.0, XNA now supports two different rendering profiles. The HiDef profile is available on the Xbox 360 and Windows PCs with DirectX 10 or better video cards, and uses Shader Model 3.0. The Reach profile is available on all XNA platforms, and uses Shader Model 2.0. If you have a DirectX 9 video card, or wish to distribute your games to computers with DirectX 9 support, you will need to right-click on your project in Solution Explorer and select **Properties**. On the **XNA Game Studio** tab, select the **Reach** profile.

Installing XNA Game Studio

To get started developing games in XNA, you will need to download and install the software. You will need both Visual C# and XNA Game Studio. With the release of XNA 4.0, the install packages have been consolidated, and both required components are included in the Windows Phone Developer Tools package.

Time for action – installing XNA Game Studio

1. Visit <http://www.microsoft.com/express/Phone/> and download the Windows Phone Developer Tools package. Run the setup wizard and allow the installation package to complete.
2. Open Visual Studio Express. Click on the **Help** menu and select **Register Product**. Click on the **Register Now** link to go to the Visual Studio Express registration page. After you have completed the registration process, return to Visual Studio Express and enter the registration number into the registration dialog box.
3. Close Visual Studio Express.
4. Download the Font Pack from <http://go.microsoft.com/fwlink/?LinkId=104778>.
5. Extract the ZIP file contents to a temporary folder (leave this folder open).
6. From the **Start** Menu, select **Control Panel**. Under **Classic View**, choose **Fonts**.
7. Drag the fonts from the temporary folder to the **Fonts** folder.
8. Close both Explorer windows.
9. Launch Visual Studio Express, and the Integrated Development Environment (IDE) will be displayed as seen in the following screenshot:





Other versions of Visual Studio and XNA

Different versions of Visual Studio and XNA can be installed on the same PC without interfering with each other. If you wish to target the Zune platform, you will need to install Visual C# 2008 Express and XNA 3.1. Additionally, Visual Studio Express and Visual Studio Professional can coexist on the same PC, and XNA will integrate with both of them if it is installed after Visual Studio.

What just happened?

You have now successfully installed the Windows Phone Developers Tools, including XNA Game Studio 4.0 and the Redistributable Font Pack provided by Microsoft for XNA developers.



The redistributable fonts package

To use its integrated text drawing methods, XNA games need to convert normal Windows fonts into an internal format called a SpriteFont. These SpriteFonts get distributed with your game, which means you will not be able to use most of the fonts on your computer due to licensing restrictions. For this reason, Microsoft has provided a selection of fonts that XNA developers can freely distribute without purchasing an individual license to do so.

Building your first game

XNA attempts to simplify many of the basic elements of game development by handling things like the game update loop and simplifying the display of graphical objects. To illustrate just how much of the background work is integrated into the XNA project templates, let's jump in straight away and create your first game within a few minutes of finishing the installation.

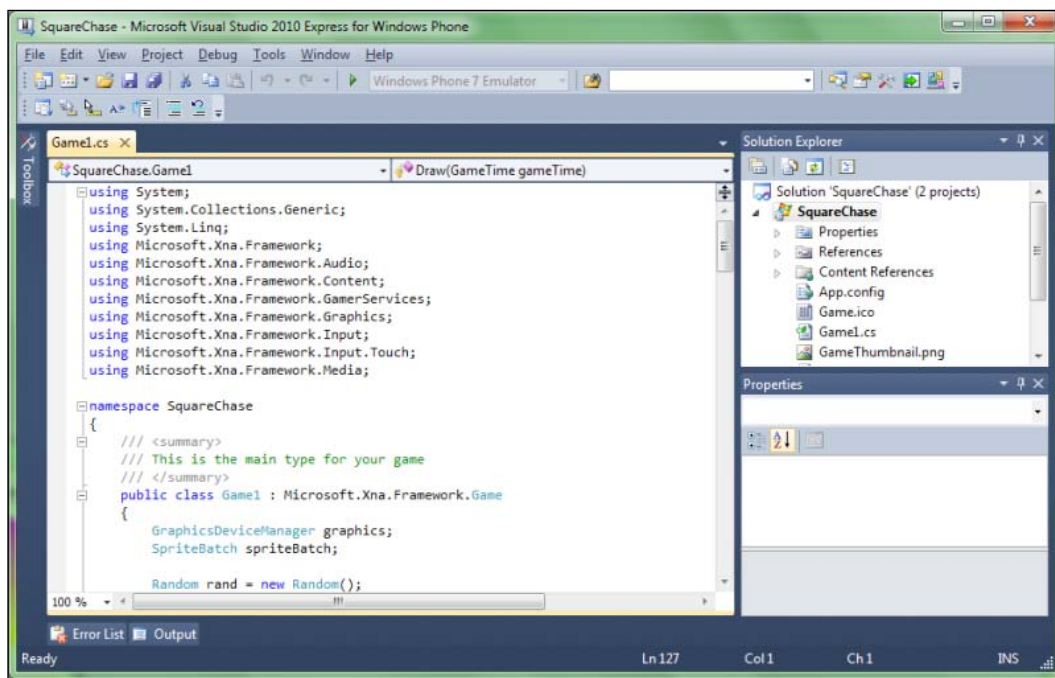
In SquareChase, we will generate randomly positioned squares of different colors while the user attempts to catch them with their mouse pointer before they disappear. While building the project, we will discuss each of the major code sections pre-defined by the XNA templates.

Time for action – creating a new Windows game project

In the Visual C# window, open the **File** menu and select **New Project...**

1. Under **Project Type**, make sure **XNA Game Studio 4.0** is selected.
2. Under **Templates**, select **Windows Game (4.0)**.
3. Name the project **SquareChase** (This will automatically update the **Solution Name**).

4. Click on **OK**:



What just happened?

Each of the XNA project templates is a series of files and settings that get copied to your new project folder. Included in this set of files is the `Game1.cs` file, which is the heart of your XNA game.



Backup your projects

When you create your project, the **Location** field specifies where it will be saved. By default, Visual Studio creates a folder in your user documents area called `Visual Studio 2010` to store both programs and configuration information. Under this folder is a `Projects` folder that contains subfolders for each new project you create. Make backups of your projects on a regular basis. You do not want to lose your hard work to a disk failure!

Anatomy of an XNA game

The most basic XNA game will have all of its code contained in the file called `Game1.cs`. This file is generated when you create a new project, and contains override declarations for the methods used to manage your game. In addition to the `Game1` class' declarations area, there are five primary methods you will customize for any XNA project.

The declarations area

Right below the class declaration for Game1 is the class level declarations area. By default, this area contains two variables:

```
GraphicsDeviceManager graphics;  
SpriteBatch spriteBatch;
```

The `graphics` object provides access to, not surprisingly, the system's video card. It can be used to alter the video mode, the size of the current viewport (the area that all drawing work will be clipped to if specified), and retrieve information about Shader Models the video card supports.

XNA provides the `SpriteBatch` class to allow you to (very quickly) draw 2D images (called "sprites") to the screen.

The declarations area is the spot for any variables that need to be maintained outside of any of the individual methods listed below. In practice, any data you need to keep track of throughout your game will be referenced, in some way, in your declarations section.

Time for action – adding variables to the class declaration area

1. Right below the `SpriteBatch spriteBatch;` line, add the following:

```
Random rand = new Random();  
Texture2D squareTexture;  
Rectangle currentSquare;  
int playerScore = 0;  
float timeRemaining = 0.0f;  
const float TimePerSquare = 0.75f;  
Color[] colors = new Color[3] { Color.Red, Color.Green,  
    Color.Blue };
```

What just happened?

These are all the variables you will need for the SquareChase mini game. Here is a quick breakdown:

rand: This instance of the `Random` class is used to generate random numbers via the `Next()` method. You will use this to generate random coordinates for the squares that will be drawn to the screen.

squareTexture: The `Texture2D` class holds a two dimensional image. We will define a small texture in memory to use when drawing the square.

currentSquare: The XNA Framework defines a structure called `Rectangle` that can be used to represent an area of the display by storing the x and y position of the upper left corner along with a width and height. `SquareChase` will generate random squares and store the location in this variable.

playerScore: Players will score one point each time they successfully "catch" a square by clicking on it with their mouse. Their score accumulates in this integer variable.

timeRemaining: When a new square is generated, this float will be set to a value representing how many seconds it will remain active. When the counter reaches zero, the square will be removed and a new square generated.

TimePerSquare: This constant is used to set the length of time that a square will be displayed before it "runs away" from the player.

colors: This array of `Color` objects will be used when a square is drawn to cycle through the three colors in the array. The `Color` structure identifies a color by four components: Red, Green, Blue, and Alpha. Each of these components can be specified as a byte from 0 to 255 representing the intensity of that component in the color. The Alpha component determines the transparency of the color, with a value of 0 indicating that the color is fully transparent and 255 indicating a fully opaque color. Alternatively, each component of a color can be specified as a float between 0.0f (fully transparent) and 1.0f (fully opaque).

The Game1 class constructor

The XNA templates define an instance of the `Microsoft.Xna.Framework.Game` class with the default name "Game1" as the primary component of your new game. Slightly more goes on behind the scenes, as we will see when we add an XNA game to a Windows Form in *Chapter 8*, but for now, we can consider the `Game1` constructor as the first thing that happens when our XNA game is executed. The class constructor is identified as `public Game1()`, and by default, it contains only two lines:

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

For most of the games in this book, we will not need to make extensive modifications to the `Game1` constructor, as its only job is to establish a link to the `GraphicsDeviceManager` object and set the default directory for the `Content` object which is used to load images, sound, and other game content.

The Initialize() method

After the constructor has finished and your XNA game begins to run, the `Initialize()` method is called. This method only runs once, and the default code created with a new project template simply calls the base version of the method. The `Initialize()` method is the ideal place to set up things like the screen resolution, toggle full screen mode, and enable the mouse in a Windows project. Other game objects that do not rely on external content such as graphics and sound resources can also be initialized here.

Time for action – customizing the Initialize() method

1. Make the mouse pointer visible by adding the following before the call to

```
base.Initialize():  
this.IsMouseVisible = true;
```

What just happened?

By default, the mouse is not visible inside the XNA game window. Setting the `IsMouseVisible` property of the running instance of the `Game1` class enables the mouse cursor in Windows.



Input types on other platforms
The Xbox, Zune, and Windows Phone do not support a mouse, so what happens when the code to enable the mouse runs on these platforms? Nothing! It just gets ignored. It is also safe to ask other platforms about their non-existent keyboards and check the state of a gamepad on a Windows PC without one attached.

The LoadContent() method

Part of the responsibility of the base `Initialize()` method is to call `LoadContent()` when the normal initialization has completed. The method is used to read in any graphical and audio resources your game will need. The default `LoadContent()` method is also where the `spriteBatch` object gets initialized. You will use the `spriteBatch` instance to draw objects to the screen during execution of the `Draw()` method.

Time for action – creating the squareTexture

1. Open Microsoft Paint or your favourite image editor and create a new 16 by 16 pixel image and fill it with white.
2. Save the image as **SQUARE.BMP** in a temporary location.
3. Back in Visual C# Express, right-click on **SquareChaseContent (Content)** in **Solution Explorer** (you may need to scroll down to see it) and select **Add | Existing Item**. Browse to the image you created and click on **Ok**.
4. Add the following code to the `LoadContent()` method after the `spriteBatch` initialization:


```
squareTexture = Content.Load<Texture2D>(@"SQUARE");
```

What just happened?

To load content, it must first exist. In steps 1 and 2, you created a bitmap image for the square texture. In step 3, you added the bitmap image as a piece of content to your project.



Powers of two

Very old graphics cards required that all texture images be sized to "powers of two" (2, 4, 8, 16, 32, 64, 128, 256, etc). This limitation is largely non-existent with modern video hardware, especially for 2D graphics. In fact, the sample code in the XNA Platform Starter Kit uses textures that do not conform to the "powers of two" limitation. In our case, the size of the image we created previously is not critical, as we will be scaling the output when we draw squares to the screen.

Finally, in step 4 you used the `Content` instance of the `ContentManager` class to load the image from the disk and into the memory when your game runs. The `Content` object is established automatically by XNA for you when you create a new project. When we add content items, such as images and sound effects to our game project, the XNA Content Pipeline converts our content files into an intermediate format that we can read via the `Content` object. These XNB files get deployed alongside the executable for our game to provide their content data at runtime.

The Update() method

Once `LoadContent()` has finished doing its job, an XNA game enters an endless loop in which it attempts to call the `Update()` method 60 times per second. This default update rate can be changed by setting the `TargetElapsedTime` property of the `Game1` object, but for our purposes, the default time step will be fine. If your `Update()` logic starts to take too long to run, your game will begin skipping calls to the `Draw()` method in favour of multiple calls to `Update()` in an attempt to catch up with the current game time.

All of your game logic gets built into the `Update()` method. It is here that you check for player input, move sprites, spawn enemies, track scores, and everything else except draw to the display. `Update()` receives a single parameter called `gameTime`, which can be used to determine how much time has elapsed since the previous call to `Update()` or to determine if your game is skipping `Draw()` calls by checking its `IsRunningSlowly` property.

The default `Update()` method contains code to exit the game if the player presses the "Back" button on the first gamepad controller.



Exiting a game under Windows

The default `Update()` code provides anyone with a gamepad a way to end the game, but what if you do not have a gamepad? Press **Alt + F4** on the keyboard or click on the standard Windows close button to exit your game when running in Windows.

Time for action – coding `Update()` for `SquareChase`

1. Add the following to `Update()` right before the call to `base.Update(gameTime);`

```
if (timeRemaining == 0.0f)
{
    currentSquare = new Rectangle(
        rand.Next(0, this.Window.ClientBounds.Width - 25),
        rand.Next(0, this.Window.ClientBounds.Height - 25),
        25, 25);
    timeRemaining = TimePerSquare;
}

MouseState mouse = Mouse.GetState();

if ((mouse.LeftButton == ButtonState.Pressed) &&
    (currentSquare.Contains(mouse.X, mouse.Y)))
{
    playerScore++;
    timeRemaining = 0.0f;
}

timeRemaining = MathHelper.Max(0, timeRemaining -
    (float)gameTime.ElapsedGameTime.TotalSeconds);

this.Window.Title = "Score : " + playerScore.ToString();
```

What just happened?

The first thing the `Update()` routine does is check to see if the current square has expired by checking to see if `timeRemaining` has been reduced to zero. If it has, a new square is generated using the `Next()` method of the `rand` object. In this form, `Next()` takes two parameters: an (inclusive) minimum value and a (non-inclusive) maximum value. In this case, the minimum is set to 0, while the maximum is set to the size of the `this.Window.ClientSize` property minus 25 pixels. This ensures that the square will always be fully within the game window.

Next, the current position and button state of the mouse is captured into the "mouse" variable via `Mouse.GetState()`. Both the `Keyboard` and the `GamePad` classes also use a `GetState()` method that captures all of the data about that input device when the method is executed.

If the mouse reports that the left button is pressed, the code checks with the `currentSquare` object by calling its `Contains()` method to determine if the mouse's coordinates fall within its area. If they do, then the player has "caught" the square and scores a point. The `timeRemaining` counter is set to 0, indicating that the next time `Update()` is called it should create a new square.

After dealing with the user input, the `MathHelper.Max()` method is used to decrease `timeRemaining` by an amount equal to the elapsed game time since the last call to `Update()`. `Max()` is used to ensure that the value does not go below zero.

Finally, the game window title bar is updated to display the player's score.



MathHelper

The `Microsoft.Xna.Framework` namespace provides a class called `MathHelper` that contains lots of goodies to make your life easier when dealing with numeric data, including converting degrees to and from radians, clamping values between a certain range, and generating smooth arcs between a starting and ending value.

The Draw() method

The final method in the default `Game1.cs` file is responsible, not surprisingly, for drawing the current game state to the display. `Draw()` is normally called once after each call to `Update()` unless something is happening to slow down the execution of your game. In that case, `Draw()` calls may be skipped in order to call `Update()` more frequently. There will always be at least one call to `Update()` between calls to `Draw()`, however, as sequential `Draw()` calls would provide no benefit—nothing in the game state will have changed.

The default `Draw()` method simply clears the display window in the Cornflower Blue color.

Time for action – draw SquareChase!

1. Alter the `GraphicsDevice.Clear(Color.CornflowerBlue)`; call and replace `Color.CornflowerBlue` with `Color.Gray` to make the game a bit easier on the eyes.
2. Add the following code after the call to clear the display:

```
spriteBatch.Begin();  
spriteBatch.Draw(  
    squareTexture,  
    currentSquare,  
    colors[playerScore % 3]);  
spriteBatch.End();
```

What just happened?

Any time you use a `SpriteBatch` object to draw to the display, you need to wrap the calls inside a `Begin()` and `End()` pair. Any number of calls to `spriteBatch.Draw()` can be included in a single batch and it is common practice to simply start a `Begin()` at the top of your `Draw()` code, use it for all of your drawing, and then `End()` it right before the `Draw()` method exits. While not benefiting our `SquareChase` game, batching sprite drawing calls greatly speeds up the process of drawing a large number of images by submitting them to the rendering system all at once instead of processing each image individually.

The `SpriteBatch.Draw()` method is used to draw a `Texture2D` object to the screen. There are a number of different options for how to specify what will be drawn. In this case, the simplest call requires a `Texture2D` object (`squareTexture`), a destination `Rectangle` (`currentSquare`), and a tint color to apply to the sprite. The expression `playerScore % 3` takes the player's score, divides it by 3, and returns the remainder. The result will always be 0, 1, or 2. This fits perfectly as an index to the elements in the `colors` array, allowing us to easily change the color of the square each time the player catches one.

Finally, the `spriteBatch.End()` tells XNA that we have finished queuing up sprites to draw and it should actually push them all out to the graphics card.

Time for action – play SquareChase!

1. Run your game by clicking on **Start Debugging** from the **Debug** menu or hit *F5* on the keyboard.
2. Play an exciting game of `SquareChase` by holding down the mouse button and trying to catch the squares with your mouse cursor:



What just happened?

You just finished your first XNA game, that's what!

Granted it is not exactly the next blockbuster, but at only 33 lines of code, it implements a simple game mechanic, user input, score tracking and display, and clock-based timing. Not bad for a few minutes work.

Have a go hero

As simple as it is, here are a couple of enhancements you could make to SquareChase:

- ◆ Vary the size of the square, making it smaller every few times the player catches one, until you reach a size of 10 pixels.
- ◆ Start off with a higher setting for `TimePerSquare` and decrease it a little each time the player catches a square. (Hint: You'll need to remove the `const` declaration in front of `TimePerSquare` if you wish to change it at runtime).

Summary

You now have a development environment set up for working on your XNA game projects, including Visual Studio Express and XNA Game Studio 4.0.

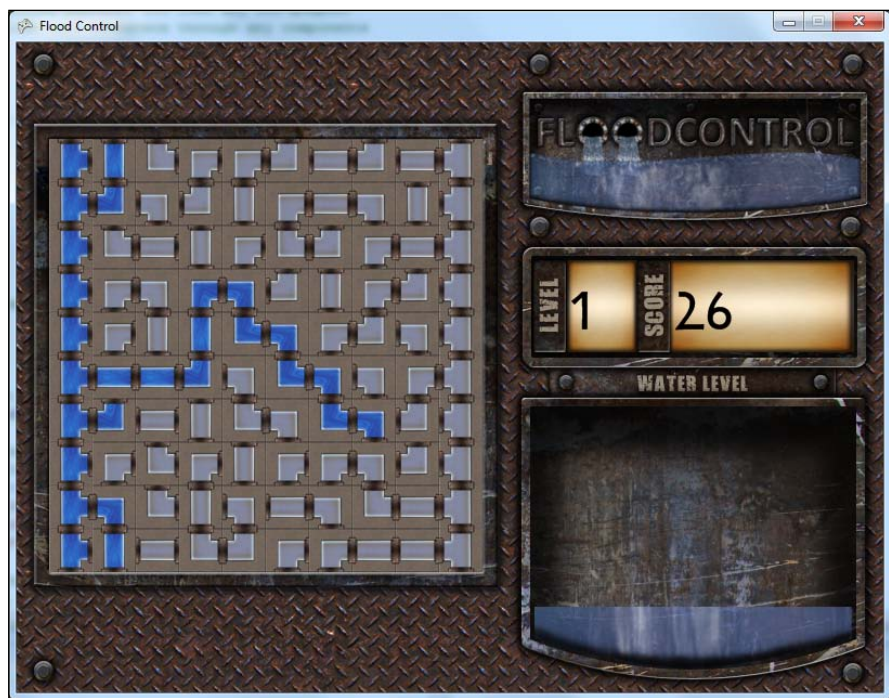
We also saw how the XNA game loop initializes and executes, and constructs an elementary game by expanding on the default methods provided by the Windows Game template.

It is time to dive head first into game creation with XNA. In the next chapter, we will begin building the puzzle game Flood Control in which the player is challenged to pump water out of their flooding underwater research station before the entire place really is underwater!

2

Flood Control – Underwater Puzzling

It was just another day at the bottom of the ocean until an explosion in one of the storage bays cracked the protective dome around Deep Sea Research Lab Alpha. Now the entire place is flooding, and the emergency pump system is a chaotic jumble of loose parts.



This chapter focuses on the following concepts:

- ◆ Using the Content Pipeline to load textures from disk
- ◆ Creating classes to divide code into logical units
- ◆ Recursively evaluating the status of the game board to check for scoring chains
- ◆ Drawing textures using the `SpriteBatch.Draw()` method
- ◆ Managing simple game states

Designing a puzzle game

The Puzzler has always been a popular game genre. From old standbys like Tetris to modern crazes like Bejeweled, puzzle games are attractive to players because they do not require a long-term time investment or a steep learning curve.

The game mechanic is the heart of any good puzzle game. This mechanic is usually very simple, with perhaps a few twists to keep the players on their toes.

In Flood Control, the player will be faced with a board containing 80 pieces of pipe. Some will be straight pipes and some will be curved. The objective of the game is to rotate the pipes to form a continuous line to pump water from the left side of the board to the right side of the board.

Completing a section of pipe drains water out of the base and scores points for the player, but destroys the pipes used. New pipes will fall into place for the player to begin another row.

Time for action – set up the Flood Control project

- 1.** Open Visual Studio Express Edition (If it is already open, select **Close Solution** from the **File** menu so you are starting with an empty slate).
- 2.** In the Visual Studio window, open the **File** menu and select **New Project...**
- 3.** Under Project Type, make sure **XNA Game Studio 4.0** is selected.
- 4.** Under Templates, select **Windows Game (4.0)**.
- 5.** Name the project **Flood Control**.

6. Click on **OK**.
7. Right-click on **Flood ControlContent (Content)** in the **Solution Explorer** window and select **Add | New Folder**. Name the folder `Textures`.
8. Add another folder under **Flood ControlContent (Content)** and name the folder `Fonts`.
9. Download the `0669_02_GRAPHICPACK.zip` file from the book's companion website and extract the files to a temporary folder.
10. Back in Visual Studio, right-click on **Textures** in the Content project and click on **Add | Existing Item**. Browse to the folder where you extracted the `0669_02_GRAPHICPACK` files and highlight all of them. Click on **Add** to add them to your project.

What just happened?

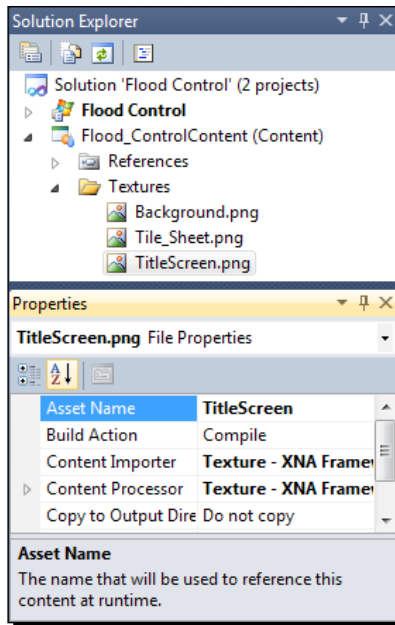
You have now set up a workspace for building Flood Control, and created a couple of folders for organizing game content. You have also imported the sample graphics for the Flood Control game into the project.

Introducing the Content Pipeline

The **Flood ControlContent (Content)** project inside **Solution Explorer** is a special kind of project called a Content Project. Items in your game's content project are converted into `.XNB` resource files by Content Importers and Content Processors.

If you right-click on one of the image files you just added to the Flood Control project and select **Properties**, you will see that for both the Importer and Processor, the Content Pipeline will use **Texture – XNA Framework**. This means that the Importer will take the file in its native format (`.PNG` in this case) and convert it to a format that the Processor recognizes as an image. The Processor then converts the image into an `.XNB` file which is a compressed binary format that XNA's content manager can read directly into a `Texture2D` object.

There are Content Importer/Processor pairs for several different types of content—images, audio, video, fonts, 3D models, and shader language effects files. All of these content types get converted to .XNB files which can be used at runtime.



In order to see how to use the Content Pipeline at runtime, let's go ahead and write the code to read these textures into memory when the game starts:

Time for action – reading textures into memory

1. Double-click on `Game1.cs` in **Solution Explorer** to open it or bring it to the front if it is already open.
2. In the Class Declarations area of `Game1` (right below `SpriteBatch spriteBatch;`), add:

```
Texture2D playingPieces;  
Texture2D backgroundScreen;  
Texture2D titleScreen;
```
3. Add code to load each of the `Texture2D` objects at the end of `LoadContent()`:

```
playingPieces = Content.Load<Texture2D>(@"Textures\Tile_Sheet");  
backgroundScreen =  
    Content.Load<Texture2D>(@"Textures\Background");  
titleScreen = Content.Load<Texture2D>(@"Textures\TitleScreen");
```

What just happened?

In order to load the textures from disk, you need an in-memory object to hold them. These are declared as instances of the `Texture2D` class.

A default XNA project sets up the `Content` instance of the `ContentManager` class for you automatically. The `Content` object's `Load()` method is used to read `.XNB` files from disk and into the `Texture2D` instances declared earlier.

One thing to note here is that the `Load()` method requires a type identifier, specified in angled brackets (`<>`), before the parameter list. Known in C# as a "Generic", many classes and methods support this kind of type specification to allow code to operate on a variety of data types. We will make more extensive use of Generics later when we need to store lists of objects in memory. The `Load()` method is used not only for textures, but also for all other kinds of content (sounds, 3D models, fonts, etc.) as well. It is important to let the `Load()` method know what kind of data you are reading so that it knows what kind of object to return.

Sprites and sprite sheets

As far as XNA and the `SpriteBatch` class are concerned, a sprite is a 2D bitmapped image that can be drawn either with or without transparency information to the screen.



Sprites vs. Textures

XNA defines a "sprite" as a 2D bitmap that is drawn directly to the screen. While these bitmaps are stored in `Texture2D` objects, the term "texture" is used when a 2D image is mapped onto a 3D object, providing a visual representation of the surface of the object. In practice, all XNA graphics are actually performed in 3D, with 2D sprites being rendered via special configurations of the XNA rendering engine.

The simple form of the `SpriteBatch.Draw()` call that you used in *Chapter 1* when drawing squares only needed three parameters: a `Texture2D` to draw, a `Rectangle` indicating where to draw it, and a `Color` to specify the tint to overlay onto the sprite.

Other overloads of the `Draw()` method, however, also allow you to specify a `Rectangle` representing the source area within the `Texture2D` to copy from. If no source `Rectangle` is specified, the entire `Texture2D` is copied and resized to fit the destination `Rectangle`.

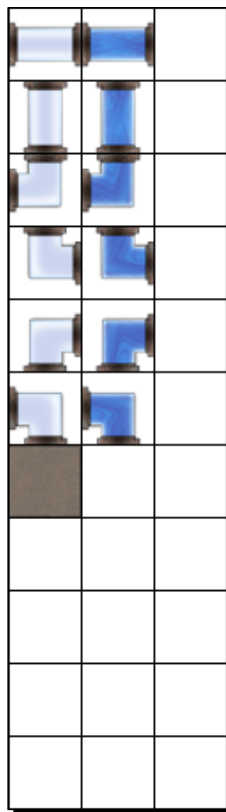


Overloads

When multiple versions of the same method are declared with either different parameters lists or different return values, each different declaration is called an "overload" of the method. Overloads allow methods to work with different types of data (for example, when setting a position you could accept two separate X and Y coordinates or a Vector2 value), or leave out parameters that can then be assigned default values.

By specifying a source `Rectangle`, however, individual pieces can be pulled from a large image. A bitmap with multiple sprites on it that will be drawn this way is called a "sprite sheet".

The `Tile_Sheet.png` file for the Flood Control project is a sprite sheet containing 13 different sprites that will be used to represent the pieces of pipe used in the game. Each image is 40 pixels wide and 40 pixels high, with a one pixel border between each sprite and also around the entire image. When we call `SpriteBatch.Draw()` we can limit what gets drawn from our texture to one of these 40 by 40 squares, allowing a single texture to hold all of the playing piece images that we need for the game:



The `Tile_Sheet.png` file was created with alpha-based transparency. When it is drawn to the screen, the alpha level of each pixel will be used to merge that pixel with any color that already occupies that location on the screen.

Using this fact, you can create sprites that don't look like they are rectangular. Internally, you will still be drawing rectangles, but visually the image can be of any shape.

What we really need now to be able to work with the playing pieces is a way to reference an individual piece, knowing not only what to draw to the screen, but what ends of the pipe connect to adjacent squares on the game board.



Alpha blending

Each pixel in a sprite can be fully opaque, fully transparent, or partially transparent. Fully opaque pixels are drawn directly, while fully transparent pixels are not drawn at all, leaving whatever has already been drawn to that pixel on the screen unchanged. In 32-bit color mode, each channel of a color (Red, Green, Blue, and Alpha) are represented by 8 bits, meaning that there are 256 different degrees of transparency between fully opaque (255) and fully transparent (0). Partially transparent pixels are combined with the current pixel color at that location to create a mixed color as if the pixels below were being seen through the new color.

Classes used in Flood Control

While it would certainly be possible to simply pile all of the game code into the `Game1` class, the result would be difficult to read and manage later on. Instead, we need to consider how to logically divide the game into classes that can manage themselves and help to organize our code.

A good rule of thumb is that a class should represent a single thing or type of thing. If you can say "This object is made up of these other objects" or "This object contains these objects", consider creating classes to represent those relationships.

The Flood Control game contains a game board made up of 80 pipes. We can abstract these pipes as a class called `GamePiece`, and provide it with the code it needs to handle rotation and provide the code that will display the piece with a `Rectangle` that can be used to pull the sprite off the sprite sheet.

The game board itself can be represented by a `GameBoard` class, which will handle managing individual `GamePiece` objects and be responsible for determining which pieces should be filled with water and which ones should be empty.

The GamePiece class

The GamePiece class represents an individual pipe on the game board. One GamePiece has no knowledge of any other game pieces (that is the responsibility of the GameBoard class), but it will need to be able to provide information about the pipe to objects that use the GamePiece class. Our class has the following requirements:

- ◆ Identify the sides of each piece that contain pipe connectors
- ◆ Differentiate between game pieces that are filled with water and that are empty
- ◆ Allow game pieces to be updated
- ◆ Automatically handle rotation by changing the piece type to the appropriate new piece type
- ◆ Given one side of a piece, provide the other sides of the piece in order to facilitate determining where water can flow through the game board
- ◆ Provide a `Rectangle` that will be used when the piece is drawn, to locate the graphic for the piece on the sprite sheet

Identifying a GamePiece

While the sprite sheet contains thirteen different images, only twelve of them are actual game pieces (the last one is an empty square). Of the twelve remaining pieces, only six of them are unique pieces. The other six are the water-filled versions of the first six images.

Each of the game pieces can be identified by which sides of the square contain a connecting pipe. This results in two straight pieces and four pieces with 90 degree bends in them.

A second value can be tracked to determine if the piece is filled with water or not instead of treating filled pieces as separate types of pieces.

Time for action – build a GamePiece class – declarations

1. Switch back to your Visual C# window if you have your image editor open.
2. Right-click on **Flood Control** in **Solution Explorer** and select **Add | Class...**
3. Name the class **GamePiece.cs** and click on **Add**.
4. At the top of the `GamePiece.cs` file, add the following to the `using` directives already in the class:

```
using Microsoft.Xna.Framework.Graphics;  
using Microsoft.Xna.Framework;
```

5. In the class declarations section, add the following:

```

public static string[] PieceTypes =
{
    "Left,Right",
    "Top,Bottom",
    "Left,Top",
    "Top,Right",
    "Right,Bottom",
    "Bottom,Left",
    "Empty"
};

public const int PieceHeight = 40;
public const int PieceWidth = 40;

public const int MaxPlayablePieceIndex = 5;
public const int EmptyPieceIndex = 6;

private const int textureOffsetX = 1;
private const int textureOffsetY = 1;
private const int texturePaddingX = 1;
private const int texturePaddingY = 1;

private string pieceType = "";
private string pieceSuffix = "";

```

6. Add two properties to retrieve information about the piece:

```

public string PieceType
{
    get { return pieceType; }
}

public string Suffix
{
    get { return pieceSuffix; }
}

```

What just happened?

You have created a new code file called `GamePiece.cs` and included the `using` statements necessary to access the pieces of the XNA Framework that the class will use.



Using Directives

Adding the XNA Framework using directives at the top of the class file allows you to access classes like `Rectangle` and `Vector2` without specifying their full assembly names. Without these statements, you would need `Microsoft.Xna.Framework.Rectangle` in your code every time you reference the type, instead of simply typing `Rectangle`.

In the declarations area, you have added an array called `PieceTypes` that gives a name to each of the different types of game pieces that will be added to the game board. There are two straight pieces, four angled pieces, and an empty tile with a background image on it, but no pipe. The array is declared as `static` because all instances of the `GamePiece` class will share the same array. A `static` member can be updated at execution time, but all members of the class will see the same changes.

Then, you have declared two integer constants that specify the height and width of an individual playing piece in pixels, along with two variables that specify the array index of the last piece that can be placed on the board (`MaxPlayablePieceIndex`) and of the fake "Empty" piece.

Next are four integers that describe the layout of the texture file you are using. There is a one pixel offset from the left and top edge of the texture (the one pixel border) and a single pixel of padding between each sprite on the sprite sheet.



Constants vs. Numeric literals

Why create constants for things like `PieceWidth` and `PieceHeight` and have to type them out when you could simply use the number 40 in their place? If you need to go back and resize your pieces later, you only need to change the size in one place instead of hoping that you find each place in the code where you entered 40 and change them all to something else. Even if you do not change the number in the game you are working on, you may reuse the code for something else later and having easily changeable parameters will make the job much easier.

There are only two pieces of information that each instance of `GamePiece` will track about itself—the type of the piece and any suffix associated with the piece. The instance members `pieceType` and `pieceSuffix` store these values. We will use the suffix to determine if the pipe that the piece represents is empty or filled with water.

However, these members are declared as `private` in order to prevent code outside the class from directly altering the values. To allow them to be read but not written to, we create a pair of properties (`pieceType` and `pieceSuffix`) that contain get blocks but no set blocks. This makes these values accessible in a read-only mode to code outside the `GamePiece` class.

Creating a GamePiece

The only information we need to create an instance of GamePiece is the piece type and, potentially, the suffix.

Time for action – building a GamePiece class: constructors

1. Add two constructors to your GamePiece.cs file after the declarations:

```
public GamePiece(string type, string suffix)
{
    pieceType = type;
    pieceSuffix = suffix;
}

public GamePiece(string type)
{
    pieceType = type;
    pieceSuffix = "";
}
```

What just happened?

A constructor is run when an instance of the GamePiece class is created. By specifying two constructors, we will allow future code to create a GamePiece by specifying a piece type with or without a suffix. If no suffix is specified, an empty suffix is assumed.

Updating a GamePiece

When a GamePiece is updated, you can change the piece type, the suffix, or both.

Time for action – GamePiece class methods – part 1 – updating

1. Add the following methods to the GamePiece class:

```
public void SetPiece(string type, string suffix)
{
    pieceType = type;
    pieceSuffix = suffix;
}

public void SetPiece(string type)
{
    SetPiece(type, "");
}
```



```

    }

    public void AddSuffix(string suffix)
    {
        if (!pieceSuffix.Contains(suffix))
            pieceSuffix += suffix;
    }

    public void RemoveSuffix(string suffix)
    {
        pieceSuffix = pieceSuffix.Replace(suffix, "");
    }

```

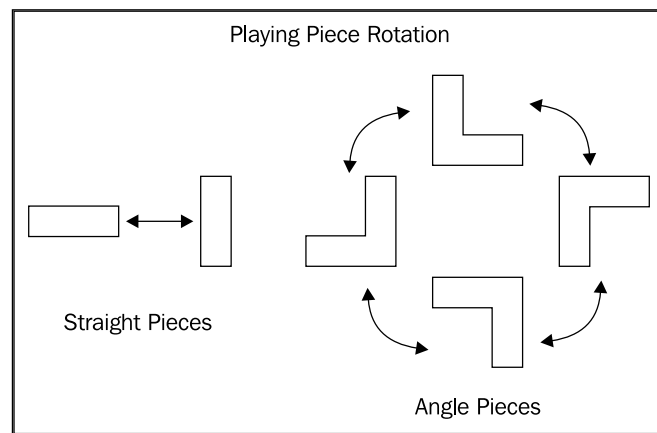
The first two methods are overloads with the same name, but different parameter lists. In a manner similar to the `GamePiece` constructors, code that wishes to update a `GamePiece` can pass it a piece type, and optionally a suffix.

Additional methods have been added to modify suffixes without changing the `pieceType` associated with the piece. The `AddSuffix()` method first checks to see if the piece already contains the suffix. If it does, nothing happens. If it does not, the suffix value passed to the method is added to the `pieceSuffix` member variable.

The `RemoveSuffix()` method uses the `Replace()` method of the string class to remove the passed suffix from the `pieceSuffix` variable.

Rotating pieces

The heart of the Flood Control play mechanic is the ability of the player to rotate pieces on the game board to form continuous pipes. In order to accomplish this, we can build a table that, given an existing piece type and a rotation direction, supplies the name of the piece type after rotation. We can then implement this code as a switch statement:



Time for action – GamePiece class methods – part 2 – rotation

1. Add the `RotatePiece()` method to the `GamePiece` class:

```
public void RotatePiece(bool Clockwise)
{
    switch (pieceType)
    {
        case "Left,Right":
            pieceType = "Top,Bottom";
            break;
        case "Top,Bottom":
            pieceType = "Left,Right";
            break;
        case "Left,Top":
            if (Clockwise)
                pieceType = "Top,Right";
            else
                pieceType = "Bottom,Left";
            break;
        case "Top,Right":
            if (Clockwise)
                pieceType = "Right,Bottom";
            else
                pieceType = "Left,Top";
            break;
        case "Right,Bottom":
            if (Clockwise)
                pieceType = "Bottom,Left";
            else
                pieceType = "Top,Right";
            break;
        case "Bottom,Left":
            if (Clockwise)
                pieceType = "Left,Top";
            else
                pieceType = "Right,Bottom";
            break;
        case "Empty":
            break;
    }
}
```

What just happened?

The only information the `RotatePiece()` method needs is a rotation direction. For straight pieces, rotation direction doesn't matter (a left/right piece will always become a top/bottom piece and vice versa).

For angled pieces, the piece type is updated based on the rotation direction and the diagram above.



Why all the strings?

It would certainly be reasonable to create constants that represent the various piece positions instead of fully spelling out things like `Bottom`, `Left` as strings. However, because the Flood Control game is not taxing on the system, the additional processing time required for string manipulation will not impact the game negatively and helps clarify how the logic works.

Pipe connectors

Our `GamePiece` class will need to be able to provide information about the connectors it contains (`Top`, `Bottom`, `Left`, and `Right`) to the rest of the game. Since we have represented the piece types as simple strings, a string comparison will determine what connectors the piece contains.

Time for action – GamePiece class methods – part 3 – connection methods

1. Add the `GetOtherEnds()` method to the `GamePiece` class:

```
public string[] GetOtherEnds(string startingEnd)
{
    List<string> opposites = new List<string>();
    foreach (string end in pieceType.Split(','))
    {
        if (end != startingEnd)
            opposites.Add(end);
    }
    return opposites.ToArray();
}
```

2. Add the `HasConnector()` method to the `GamePiece` class:

```
public bool HasConnector(string direction)
{
    return pieceType.Contains(direction);
}
```

The `GetOtherEnds()` method creates an empty `List` object for holding the ends we want to return to the calling code. It then uses the `Split()` method of the `string` class to get each end listed in the `pieceType`. For example, the `Top, Bottom` piece will return an array with two elements. The first element will contain `Top` and the second will contain `Bottom`. The comma delimiter will not be returned with either string.

If the end in question is not the same as the `startingEnd` parameter that was passed to the method, it is added to the list. After all of the items in the string have been examined, the list is converted to an array and returned to the calling code.

In the previous example, requesting `GetOtherEnds("Top")` from a `GamePiece` with a `pieceType` value of `Top, Bottom` will return a string array with a single element containing `Bottom`.

We will need this information in a few moments when we have to figure out which pipes are filled with water and which are empty.

The second function, `HasConnector()` simply returns "true" if the `pieceType` string contains the string value passed in as the direction parameter. This will allow code outside the `GamePiece` class to determine if the piece has a connector facing in any particular direction.

Sprite sheet coordinates

Because we set up the `PieceTypes` array listing the pieces in the same order that they exist on the sprite sheet texture, we can calculate the position of the rectangle to draw from based on the `pieceType`.

Time for action – GamePiece class methods – part 4 – GetSourceRect

1. Add the `GetSourceRect()` method to the `GamePiece` class:

```
public Rectangle GetSourceRect()
{
    int x = textureOffsetX;
    int y = textureOffsetY;

    if (pieceSuffix.Contains("W"))
        x += PieceWidth + texturePaddingX;

    y += (Array.IndexOf(PieceTypes, pieceType) *
        (PieceHeight + texturePaddingY));

    return new Rectangle(x, y, PieceWidth, PieceHeight);
}
```

What just happened?

Initially, the `x` and `y` variables are set to the `textureOffsets` that are listed in the `GamePiece` class declaration. This means they will both start with a value of one.

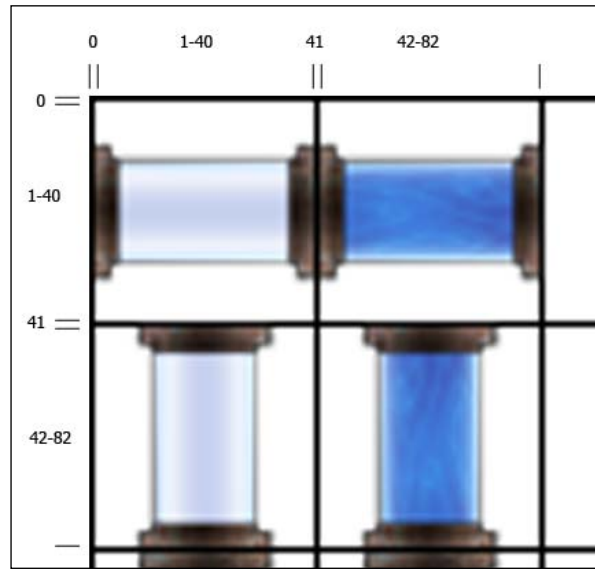
Because the sprite sheet is organized with a single type of pipe on each row, the `x` coordinate of the `Rectangle` is the easiest to determine. If the `pieceSuffix` variable does not contain a `w` (signifying that the piece is filled with water), the `x` coordinate will simply remain 1.

If the `pieceSuffix` does contain the letter `w` (indicating the pipe is filled), the width of a piece (40 pixels), along with the padding between the pieces (1 pixel), are added to the `x` coordinate of the source `Rectangle`. This shifts the `x` coordinate from 1 to a value of $1 + 40 + 1$, or **42** which corresponds to the second column of images on the sprite sheet.

To determine the `y` coordinate for the source rectangle, `Array.IndexOf(PieceTypes, pieceType)` is used to locate the `pieceType` within the `PieceTypes` array. The index that is returned represents the position of the tile on the sprite sheet (because the array is organized in the same order as the pieces on the image). For example, `Left, Right` returns zero, while `Top, Bottom` returns one and `Empty` returns six.

The value of this index is multiplied by the height of a game piece plus the padding between pieces. For our sprite sheet, an index of 2 (the `Left, Top` piece) would be multiplied by **41** (`PieceHeight` of **40** plus `texturePaddingY` of **1**) resulting in a value of **82** being added to the `y` variable.

Finally, the new `Rectangle` is returned, comprised of the calculated `x` and `y` coordinates and the predefined width and height of a piece:



The GameBoard class

Now that we have a way to represent pieces in memory, the next logical step is to create a way to represent an entire board of playing pieces.

The game board is a two-dimensional array of GamePiece objects, and we can build in some additional functionality to allow our code to interact with pieces on the game board by their X and Y coordinates.

The GameBoard class needs to:

- ◆ Store a GamePiece object for each square on the game board
- ◆ Provide methods for code using the GameBoard to update individual pieces by passing calls through to the underlying GamePiece instances
- ◆ Randomly assign a piece type to a GamePiece
- ◆ Set and clear the "Filled with water" flags on individual GamePieces
- ◆ Determine which pipes should be filled with water based on their position and orientation and mark them as filled
- ◆ Return lists of potentially scoring water chains to code using the GameBoard

Time for action – create the GameBoard.cs class

1. As you did to create the GamePiece class, right-click on **Flood Control** in **Solution Explorer** and select **Add | Class...** Name the new class file **GameBoard.cs**.
2. Add the `using` directive for the XNA framework at the top of the file:
`using Microsoft.Xna.Framework;`
3. Add the following declarations to the GameBoard class:

```
Random rand = new Random();

public const int GameBoardWidth = 8;
public const int GameBoardHeight = 10;

private GamePiece[,] boardSquares =
    new GamePiece[GameBoardWidth, GameBoardHeight];

private List<Vector2> WaterTracker = new List<Vector2>();
```

What just happened?

We used the `Random` class in `SquareChase` to generate random numbers. Since we will need to randomly generate pieces to add to the game board, we need an instance of `Random` in the `GameBoard` class.

The two constants and the `boardSquares` array provide the storage mechanism for the `GamePiece` objects that make up the 8 by 10 piece board.

Finally, a `List` of `Vector2` objects is declared that we will use to identify scoring pipe combinations. The `List` class is one of C#'s Generic Collection classes—classes that use the Generic templates (angle brackets) we first saw when loading a texture for `SquareChase`. Each of the Collection classes can be used to store multiple items of the same type, with different methods to access the entries in the collection. We will use several of the Collection classes in our projects. The `List` class is much like an array, except that we can add any number of values at runtime, and remove values in the `List` if necessary.

A `Vector2` is a structure defined by the XNA Framework that holds two floating point values, `X` and `Y`. Together the two values represent a vector pointing in any direction from an imaginary origin (0, 0) point. We will use `Vector2` structures to represent the locations on our game board in `Flood Control`, placing the origin in the upper left corner of the board.

Creating the game board

If we were to try to use any of the elements in the `boardSquares` array, at this point, we would get a `Null Reference` exception because none of the `GamePiece` objects in the array have actually been created yet.

Time for action – initialize the game board

1. Add a constructor to the `GameBoard` class:

```
public GameBoard()
{
    ClearBoard();
}
```

2. Add the `ClearBoard()` helper method to the `GameBoard` class:

```
public void ClearBoard()
{
    for (int x = 0; x < GameBoardWidth; x++)
        for (int y = 0; y < GameBoardHeight; y++)
            boardSquares[x, y] = new GamePiece("Empty");
}
```

What just happened?

When a new instance of the `GameBoard` class is created, the constructor calls the `ClearBoard()` helper method, which simply creates 80 empty game pieces and assigns them to each element in the array.



Helper methods

Why not simply put the two for loops that clear the board into the `GameBoard` constructor? Splitting the work into methods that accomplish a single purpose greatly helps to keep your code both readable and maintainable. Additionally, by splitting `ClearBoard()` out as its own method we can call it separately from the constructor. When we add increasing difficulty levels in *Chapter 3*, we will make use of this call when a new level starts.

Updating GamePieces

The `boardSquares` array in the `GameBoard` class is declared as a private member, meaning that the code that uses the `GameBoard` will not have direct access to the pieces contained on the board.

In order for code in our `Game1` class to interact with a `GamePiece`, we will need to create public methods in the `GameBoard` class that expose the pieces in `boardSquares`.

Time for action – manipulating the game board

1. Add public methods to the `GameBoard` class to interact with `GamePiece`:

```
public void RotatePiece(int x, int y, bool clockwise)
{
    boardSquares[x, y].RotatePiece(clockwise);
}

public Rectangle GetSourceRect(int x, int y)
{
    return boardSquares[x, y].GetSourceRect();
}

public string GetSquare(int x, int y)
{
    return boardSquares[x, y].PieceType;
}

public void SetSquare(int x, int y, string pieceName)
{

```



```
        boardSquares[x, y].SetPiece(pieceName);
    }

    public bool HasConnector(int x, int y, string direction)
    {
        return boardSquares[x, y].HasConnector(direction);
    }

    public void RandomPiece(int x, int y)
    {
        boardSquares[x, y].SetPiece(GamePiece.PieceTypes[rand.Next(0,
            GamePiece.MaxPlayablePieceIndex+1)]);
    }
}
```

What just happened?

`RotatePiece()`, `GetSourceRect()`, `GetSquare()`, `SetSquare()`, and `HasConnector()` methods simply locate the appropriate `GamePiece` within the `boardSquares` array and pass on the function request to the piece.

The `RandomPiece()` method uses the `rand` object to get a random value from the `PieceTypes` array and assign it to a `GamePiece`. It is important to remember that with the `Random.Next()` method overload used here, the second parameter is non-inclusive. In order to generate a random number from 0 through 5, the second parameter needs to be 6.

Filling in the gaps

Whenever the player completes a scoring chain, the pieces in that chain are removed from the board. Any pieces above them fall down into the vacated spots and new pieces are generated.

Time for action – filling in the gaps

1. Add the `FillFromAbove()` method to the `GameBoard` class.

```
public void FillFromAbove(int x, int y)
{
    int rowLookup = y - 1;

    while (rowLookup >= 0)
    {
        if (GetSquare(x, rowLookup) != "Empty")
        {
            SetSquare(x, y,
```

```

        GetSquare(x, rowLookup));
        SetSquare(x, rowLookup, "Empty");
        rowLookup = -1;
    }
    rowLookup--;
}
}

```

What just happened?

Given a square to fill, `FillFromAbove()` looks at the piece directly above to see if it is marked as `Empty`. If it is, the method will subtract one from `rowLookup` and start over until it reaches the top of the board. If no non-empty pieces are found when the top of the board is reached, the method does nothing and exits.

When a non-empty piece is found, it is copied to the destination square, and the copied piece is changed to an empty piece. The `rowLookup` variable is set to `-1` to ensure that the loop does not continue to run.

Generating new pieces

We can create a single method that will fill any empty spaces on the game board, and use it when the game begins and when pieces are removed from the board after scoring.

Time for action – generating new pieces

1. Add the `GenerateNewPieces()` method to the `GameBoard` class:

```

public void GenerateNewPieces(bool dropSquares)
{
    if (dropSquares)
    {
        for (int x = 0; x < GameBoard.GameBoardWidth; x++)
        {
            for (int y = GameBoard.GameBoardHeight - 1; y >= 0; y--)
            {
                if (GetSquare(x, y) == "Empty")
                {
                    FillFromAbove(x, y);
                }
            }
        }
    }
}

```

```
    }

    for (int y = 0; y < GameBoard.GameBoardHeight; y++)
        for (int x = 0; x < GameBoard.GameBoardWidth; x++)
        {
            if (GetSquare(x, y) == "Empty")
            {
                RandomPiece(x, y);
            }
        }
    }
```

What just happened?

When `GenerateNewPieces()` is called with "true" passed as `dropSquares`, the looping logic processes one column at a time from the bottom up. When it finds an empty square it calls `FillFromAbove()` to pull a filled square from above into that location.

The reason the processing order is important here is that, by filling a lower square from a higher position, that higher position will become empty. It, in turn, will need to be filled from above.

After the holes are filled (or if `dropSquares` is set to false) `GenerateNewPieces()` examines each square in `boardSquares` and asks it to generate random pieces for each square that contains an empty piece.

Water filled pipes

Whether or not a pipe is filled with water is managed separately from its orientation. Rotating a single pipe could change the water-filled status of any number of other pipes without changing their rotation.

Instead of filling and emptying individual pipes, however, it is easier to empty all of the pipes and then refill the pipes that need to be marked as having water in them.

Time for action – water in the pipes

1. Add a method to the `GameBoard` class to clear the water marker from all pieces:

```
public void ResetWater()
{
    for (int y = 0; y < GameBoardHeight; y++)
        for (int x = 0; x < GameBoardWidth; x++)
            boardSquares[x,y].RemoveSuffix("W");
}
```

2. Add a method to the GameBoard class to fill an individual piece with water:

```
public void FillPiece(int X, int Y)
{
    boardSquares[X,Y].AddSuffix("W");
}
```

What just happened?

The `ResetWater()` method simply loops through each item in the `boardSquares` array and removes the `W` suffix from the `GamePiece`. Similarly, to fill a piece with water, the `FillPiece()` method adds the `W` suffix to the `GamePiece`. Recall that by having a `W` suffix, the `GetSourceRect()` method of `GamePiece` shifts the source rectangle one tile to the right on the sprite sheet, returning the image for a pipe filled with water instead of an empty pipe.

Propagating water

Now that we can fill individual pipes with water, we can write the logic to determine which pipes should be filled depending on their orientation.

Time for action – making the connection

1. Add the `PropagateWater()` method to the `GameBoard` class:

```
public void PropagateWater(int x, int y, string fromDirection)
{
    if ((y >= 0) && (y < GameBoardHeight) &&
        (x >= 0) && (x < GameBoardWidth))
    {
        if (boardSquares[x,y].HasConnector(fromDirection) &&
            !boardSquares[x,y].Suffix.Contains("W"))
        {
            FillPiece(x, y);
            WaterTracker.Add(new Vector2(x, y));
            foreach (string end in
                boardSquares[x,y].
                    GetOtherEnds(fromDirection))
            switch (end)
            {
                case "Left": PropagateWater(x - 1, y,
                    "Right");
                    break;
                case "Right": PropagateWater(x + 1, y,
                    "Left");
                    break;
            }
        }
    }
}
```

```

        case "Top": PropagateWater(x, y - 1,
            "Bottom");
            break;
        case "Bottom": PropagateWater(x, y + 1,
            "Top");
            break;
    }
}
}

```

2. Add the GetWaterChain() method to the GameBoard class:

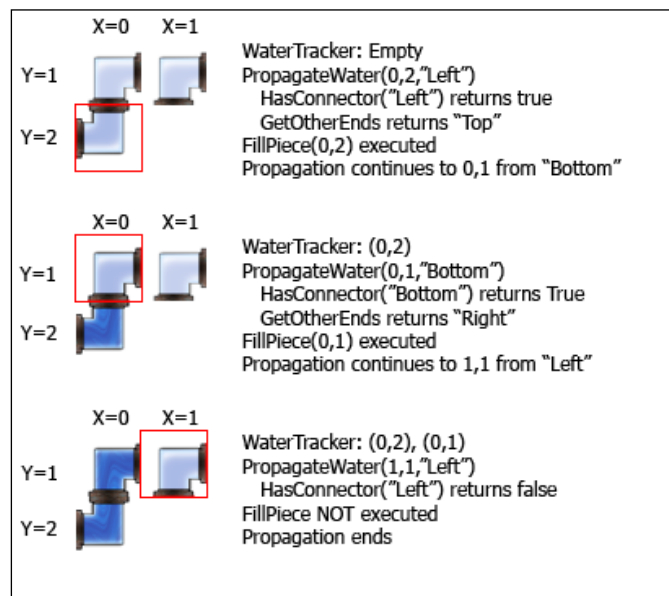
```

public List<Vector2> GetWaterChain(int y)
{
    WaterTracker.Clear();
    PropagateWater(0, y, "Left");
    return WaterTracker;
}

```

What just happened?

Together, `GetWaterChain()` and `PropagateWater()` are the keys to the entire Flood Control game, so understanding how they work is vital. When the game code wants to know if the player has completed a scoring row, it will call the `GetWaterChain()` method once for each row on the game board:



The `WaterTracker` list is cleared and `GetWaterChain()` calls `PropagateWater()` for the first square in the row, indicating that the water is coming from the **Left** direction.

The `PropagateWater()` method checks to make sure that the **x** and **y** coordinates passed to it exist within the board and, if they do, checks to see if the piece at that location has a connector matching the `fromDirection` parameter and that the piece is not already filled with water. If all of these conditions are met, that piece gets filled with water and added to the `WaterTracker` list.

Finally, `PropagateWater()` gets a list of all other directions that the piece contains (in other words, all directions the piece contains that do not match `fromDirection`). For each of these directions `PropagateWater()` recursively calls itself, passing in the new **x** and **y** location as well as the direction the water is coming from.

Building the game

We now have the component classes we need to build the Flood Control game, so it is time to bring the pieces together in the `Game1` class.

Declarations

We only need a handful of game-wide declarations to manage things like the game board, the player's score, and the game state.

Time for action – Game1 declarations

1. Double click on the `Game1.cs` file in **Solution Explorer** to reactivate the **Game1.cs** code file window.
2. Add the following declarations to the `Game1` class member declaration area:

```
GameBoard gameBoard;

Vector2 gameBoardDisplayOrigin = new Vector2(70, 89);

int playerScore = 0;

enum GameStates { TitleScreen, Playing };
GameStates gameState = GameStates.TitleScreen;

Rectangle EmptyPiece = new Rectangle(1, 247, 40, 40);

const float MinTimeSinceLastInput = 0.25f;
float timeSinceLastInput = 0.0f;
```

What just happened?

The `gameBoard` instance of `GameBoard` will hold all of the playing pieces, while the `gameBoardDisplayOrigin` vector points to where on the screen the board will be drawn. Using a vector like this makes it easy to move the board in the event that you wish to change the layout of your game screen.

As we did in `SquareChase`, we store the player's score and will display it in the window title bar.

In order to implement a simple game state mechanism, we define two game states. When in the `TitleScreen` state, the game's title image will be displayed and the game will wait until the user presses the *Space bar* to start the game. The state will then switch to `Playing`, which will display the game board and allow the user to play.

If you look at the sprite sheet for the game, the pipe images themselves do not cover the entire 40x40 pixel area of a game square. In order to provide a background, an empty tile image will be drawn in each square first. The `EmptyPieceRectangle` is a convenient pointer to where the empty background is located on the sprite sheet.

Just as we used an accumulating timer in `SquareChase` to determine how long to leave a square in place before moving it to a new location, we will use the same timing mechanism to make sure that a single click by the user does not send a game piece spinning unpredictably. Remember that the `Update()` method will be executing up to 60 times each second, so slowing the pace of user input is necessary to make the game respond in a way that feels natural.

Initialization

Before we can use the `gameBoard` instance, it needs to be initialized. We will also need to enable the mouse cursor.

Time for action – updating the `Initialize()` method

1. Update the `Initialize()` method to include the following:

```
this.IsMouseVisible = true;
graphics.PreferredBackBufferWidth = 800;
graphics.PreferredBackBufferHeight = 600;
graphics.ApplyChanges();
gameBoard = new GameBoard();
```

What just happened?

After making the mouse cursor visible, we set the size of the `BackBuffer` to 800 by 600 pixels. On Windows, this will size the game window to 800 by 600 pixels as well.

The constructor for the `GameBoard` class calls the `ClearBoard()` member, so each of the pieces on the `gameBoard` instance will be set to `Empty`.

The Draw() method – the title screen

In the declarations section, we established two possible game states. The first (and default) state is `GameStates.TitleScreen`, indicating that the game should not be processing actual game play, but should instead be displaying the game's logo and waiting for the user to begin the game.

Time for action – drawing the screen – the title screen

1. Modify the `Draw()` method of `Game1` to include the code necessary to draw the game's title screen after `GraphicsDevice.Clear(Color.CornflowerBlue);`

```
if (gameState == GameStates.TitleScreen)
{
    spriteBatch.Begin();
    spriteBatch.Draw(titleScreen,
        new Rectangle(0, 0,
            this.Window.ClientBounds.Width,
            this.Window.ClientBounds.Height),
        Color.White);
    spriteBatch.End();
}
```
2. Run the game and verify that the title screen is displayed. You will not be able to start the game however, as we haven't written the `Update()` method yet.

3. Stop the game by pressing *Alt + F4*.



What just happened?

The title screen is drawn with a single call to the `Draw()` method of the `spriteBatch` object. Since the title screen will cover the entire display, a rectangle is created that is equal to the width and height of the game window.

The Draw() method – the play screen

Finally, we are ready to display the playing pieces on the screen. We will accomplish this by using a simple loop to display all of the playing pieces in the `gameBoard` object.

Time for action – drawing the screen – the play screen

1. Update the `Draw()` method of the `Game1` class to add the code to draw the game board after the code that draws the title screen:

```
if (gameState == GameStates.Playing)
{
    spriteBatch.Begin();

    spriteBatch.Draw(backgroundScreen,
        new Rectangle(0, 0,
            this.Window.ClientBounds.Width,
            this.Window.ClientBounds.Height),
```

```

        Color.White);

    for (int x = 0; x < GameBoard.GameBoardWidth; x++)
        for (int y = 0; y < GameBoard.GameBoardHeight; y++)
        {
            int pixelX = (int)gameBoardDisplayOrigin.X +
                (x * GamePiece.PieceWidth);
            int pixelY = (int)gameBoardDisplayOrigin.Y +
                (y * GamePiece.PieceHeight);

            spriteBatch.Draw(
                playingPieces,
                new Rectangle(
                    pixelX,
                    pixelY,
                    GamePiece.PieceWidth,
                    GamePiece.PieceHeight),
                EmptyPiece,
                Color.White);

            spriteBatch.Draw(
                playingPieces, new Rectangle(
                    pixelX,
                    pixelY,
                    GamePiece.PieceWidth,
                    GamePiece.PieceHeight),
                gameBoard.GetSourceRect(x, y),
                Color.White);
        }

    this.Window.Title = playerScore.ToString();

    spriteBatch.End();
}

```

What just happened?

As you can see, the code to draw the game board begins exactly like the code to draw the title screen. Since we are using a background image that takes up the full screen, we draw it exactly the same way as the title screen.

Next, we simply loop through `gameBoard` to draw the squares. The `pixelX` and `pixelY` variables are calculated to determine where on the screen each of the game pieces will be drawn. Since both `x` and `y` begin at 0, the `(x * GamePiece.PieceWidth)` and `(y * GamePiece.PieceHeight)` will also be equal to zero, resulting in the first square being drawn at the location specified by the `gameBoardDisplayOrigin` vector.

As x increments, each new piece is drawn 40 pixels further to the right than the previous piece. After a row has been completed, the y value increments, and a new row is started 40 pixels below the previous row.

The first `spriteBatch.Draw()` call uses `Rectangle(pixelX, pixelY, GamePiece.PieceWidth, GamePiece.PieceHeight)` as the destination rectangle and `EmptyPiece` as the source rectangle. Recall that we added this `Rectangle` to our declarations area as a shortcut to the location of the empty piece on the sprite sheet.

The second `spriteBatch.Draw()` call uses the same destination rectangle, overlaying the playing piece image onto the empty piece that was just drawn. It asks the `gameBoard` to provide the source rectangle for the piece it needs to draw.

The player's score is displayed in the window title bar, and `spriteBatch.End()` is called to finish up the `Draw()` method.

Keeping score

Longer chains of filled water pipes score the player more points. However, if we were to simply assign a single point to each piece in the pipe chain, there would be no scoring advantage to making longer chains versus quickly making shorter chains.

Time for action – scores and scoring chains

1. Add a method to the `Game1` class to calculate a score based on the number of pipes used:

```
private int DetermineScore(int SquareCount)
{
    return (int)((Math.Pow((SquareCount/5), 2) + SquareCount)*10);
}
```

2. Add a method to evaluate a chain to determine if it scores and process it:

```
private void CheckScoringChain(List<Vector2> WaterChain)
{
    if (WaterChain.Count > 0)
    {
        Vector2 LastPipe = WaterChain[WaterChain.Count - 1];

        if (LastPipe.X == GameBoard.GameBoardWidth - 1)
        {
            if (gameBoard.HasConnector(
                (int)LastPipe.X, (int)LastPipe.Y, "Right"))
            {

```

```

        playerScore += DetermineScore(WaterChain.Count);

        foreach (Vector2 ScoringSquare in WaterChain)
        {
            gameBoard.SetSquare((int)ScoringSquare.X,
                                (int)ScoringSquare.Y, "Empty");
        }
    }
}

```

What just happened?

`DetermineScore()` accepts the number of squares in a scoring chain and returns a score value for that chain. The number of squares in the chain is divided by 5, and that number is squared. The initial number of squares is added to the result, and the final amount is multiplied by 10.

$$\text{Score} = (((\text{Squares} / 5) ^ 2) + \text{Squares}) * 10$$

For example, a minimum scoring chain would be 8 squares (forming a straight line across the board). This would result in 1 squared plus 8 times 10, or 90 points. If a chain had 18 squares the result would be 3 squared plus 18 times 10, or 270 points. This makes longer scoring chains (especially increments of five squares) award much higher scores than a series of shorter chains.

The `CheckScoringRow()` method makes sure that there are entries in the `WaterChain` list, and then examines the last piece in the chain and checks to see if it has an `x` value of 7 (the right-most column on the board). If it does, the `HasConnector()` method is checked to see if the last pipe has a connector to the right, indicating that it completes a chain across the board.

After updating `playerScore` for the scoring row, `CheckScoringRow()` sets all of the pieces in the scoring row to `Empty`. They will be refilled by a subsequent call to the `GenerateNewPieces()` method.

Input handling

The player interacts with Flood Control using the mouse. For readability reasons, we will create a helper method that deals with mouse input and call it when appropriate from the `Update()` method.

Time for action – handling mouse input

1. Add the `HandleMouseInput()` helper method to the `Game1` class:

```
private void HandleMouseInput(MouseState mouseState)
{
    int x = ((mouseState.X -
               (int)gameBoardDisplayOrigin.X) / GamePiece.PieceWidth);

    int y = ((mouseState.Y -
               (int)gameBoardDisplayOrigin.Y) / GamePiece.PieceHeight);

    if ((x >= 0) && (x < GameBoard.GameBoardWidth) &&
        (y >= 0) && (y < GameBoard.GameBoardHeight))
    {
        if (mouseState.LeftButton == ButtonState.Pressed)
        {
            gameBoard.RotatePiece(x, y, false);
            timeSinceLastInput = 0.0f;
        }

        if (mouseState.RightButton == ButtonState.Pressed)
        {
            gameBoard.RotatePiece(x, y, true);
            timeSinceLastInput = 0.0f;
        }
    }
}
```

What just happened?

The `MouseState` class reports the `x` and `y` position of the mouse relative to the upper left corner of the window. What we really need to know is what square on the game board the mouse was over.

We calculate this by taking the mouse position and subtracting the `gameBoardDisplayOrigin` from it and then dividing the remaining number by the size of a game board square.

If the resulting `x` and `y` locations fall within the game board, the left and right mouse buttons are checked. If the left button is pressed, the piece is rotated counterclockwise. The right button rotates the piece clockwise. In either case, the input delay timer is reset to `0.0f` since input was just processed.

Letting the player play!

Only one more section to go and you can begin playing Flood Control. We need to code the `Update()` method to tie together all of the game logic we have created so far.

Time for action – letting the player play

1. Modify the `Update()` method of `Game1.cs` by adding the following before the call to `base.Update(gameTime)`:

```
switch (gameState)
{
    case GameStates.TitleScreen:
        if (Keyboard.GetState().IsKeyDown(Keys.Space))
        {
            gameBoard.ClearBoard();
            gameBoard.GenerateNewPieces(false);
            playerScore = 0;
            gameState = GameStates.Playing;
        }
        break;

    case GameStates.Playing:
        timeSinceLastInput +=
            (float)gameTime.ElapsedGameTime.TotalSeconds;

        if (timeSinceLastInput >= MinTimeSinceLastInput)
        {
            HandleMouseInput(Mouse.GetState());
        }

        gameBoard.ResetWater();

        for (int y = 0; y < GameBoard.GameBoardHeight; y++)
        {
            CheckScoringChain(gameBoard.GetWaterChain(y));
        }

        gameBoard.GenerateNewPieces(true);

        break;
}
```

What just happened?

The `Update()` method performs two different functions, depending on the current `gameState` value. If the game is in `TitleScreen` state, `Update()` examines the keyboard, waiting for the *Space bar* to be pressed. When it is, `Update()` clears the `gameBoard`, generates a new set of pieces, resets the player's score, and changes `gameState` to `Playing`.

While in the `Playing` state, `Update()` accumulates time in `timeSinceLastInput` in order to pace the game play properly. If enough time has passed, the `HandleMouseInput()` method is called to allow the player to rotate game pieces.

`Update()` then calls `ResetWater()` to clear the water flags for all pieces on the game board. This is followed by a loop that processes each row, starting at the top and working downward, using `CheckScoringChain()` and `GetWaterChain()` to "fill" any pieces that should have water in them and check the results of each row for completed chains.

Finally, `GenerateNewPieces()` is called with the "true" parameter for `dropSquares`, which will cause `GenerateNewPieces()` to fill the empty holes from the squares above, and then generate new pipes to replace the empty squares.

Play the game

You now have all of the components assembled, and can run Flood Control and play!

Summary

You now have a working Flood Control game. In this chapter we have looked at:

- ◆ Adding content objects to your project and loading them into textures at runtime using an instance of the `ContentManager` class
- ◆ Dividing the code into classes that represent objects in the game
- ◆ Building a recursive method
- ◆ Use the `SpriteBatch.Draw()` method to display images
- ◆ Divide the `Update()` and `Draw()` code into different units based on the current game state

In *Chapter 3*, we will spruce up the Flood Control game, adding animation by modifying the parameters of the `SpriteBatch.Draw()` method and creating text effects in order to make the game visually more appealing.

3

Flood Control – Smoothing Out the Rough Edges

While playable, Flood Control in its current form is rather rough. When the player clicks on game pieces, they simply flop to their new orientation. Completed rows vanish without a trace, being filled in so rapidly that it is hard to tell if they actually disappeared at all. The game never ends! Once you have started, you can play forever, not worrying about the underwater research lab actually filling up with water.

In this chapter, we will address these issues by:

- ◆ Animating the rotation of pieces when manipulated by the player
- ◆ Gradually fading out pieces of completed scoring chains
- ◆ Animating the falling of pieces into place on the board
- ◆ Implementing the flooding of the dome and adding increasing difficulty levels
- ◆ Adding a SpriteFont to the game and displaying the current level and score in their appropriate positions on the screen

All of these enhancements will give the player a better game experience, as well as give us the opportunity to learn more about how the SpriteBatch class can be used for animation and text display.

Animated pieces

We will define three different types of animated pieces: rotating, falling, and fading. The animation for each of these types will be accomplished by altering the parameters of the `SpriteBatch.Draw()` call.

Classes for animated pieces

In order to represent the three types of animated pieces, we will create three new classes. Each of these classes will inherit from the `GamePiece` class, meaning they will contain all of the methods and members of the `GamePiece` class, but add additional information to support the animation.



Child classes

Child classes inherit all of their parent's members and methods. The `RotatingPiece` class can refer to the `pieceType` and `suffix` of the piece without recreating them within `RotatingPiece` itself. Additionally, child classes can extend the functionality of their base class, adding new methods and properties or overriding old ones. In fact, `Game1` itself is a child of the `Microsoft.Xna.Game` class, which is why all of the methods we use (`Update()`, `Draw()`, `LoadContent()`, and so on) are declared as "override".

Let's begin by creating the class we will use for rotating pieces.

Time for action – rotating pieces

1. Open your existing Flood Control project in Visual C# Express if it is not already active.
2. Add a new class to the project called "RotatingPiece".
3. Add `"using Microsoft.Xna.Framework;"` to the using area at the top of the class.
4. Update the declaration of the class to read `class RotatingPiece : GamePiece`.
5. Add the following declarations to the `RotatingPiece` class:

```
public bool clockwise;

public static float rotationRate = (MathHelper.PiOver2 / 10);
private float rotationAmount = 0;
public int rotationTicksRemaining = 10;
```

6. Add a property to retrieve the current rotation amount:

```
public float RotationAmount
{
    get
    {
        if (clockwise)
            return rotationAmount;
        else
            return (MathHelper.Pi*2) - rotationAmount;
    }
}
```

7. Add a constructor for the RotatingPiece class:

```
public RotatingPiece(string pieceType, bool clockwise)
    : base(pieceType)
{
    this.clockwise = clockwise;
}
```

8. Add a method to update the piece:

```
public void UpdatePiece()
{
    rotationAmount += rotationRate;
    rotationTicksRemaining = (int)MathHelper.Max(
        0,
        rotationTicksRemaining-1);
}
```

What just happened?

In step 2, we modified the declaration of the RotatingPiece class by adding `: GamePiece` to the end of it. This indicates to Visual C# that the RotatingPiece class is a child of the GamePiece class.

The clockwise variable stores a "true" value if the piece will be rotating clockwise and "false" if the rotation is counter-clockwise.

When a game piece is rotated, it will turn a total of 90 degrees (or $\pi/2$ radians) over 10 animation frames. The MathHelper class provides a number of constants to represent commonly used numbers, with `MathHelper.PiOver2` being equal to the number of radians in a 90 degree angle. We divide this constant by 10 and store the result as the `rotationRate` for use later. This number will be added to the `rotationAmount` float, which will be referenced when the animated piece is drawn.



Working with radians

All angular math is handled in radians from XNA's point of view. A complete (360 degree) circle contains 2π radians. In other words, one radian is equal to about 57.29 degrees. We tend to relate to circles more often in terms of degrees (a right angle being 90 degrees, for example), so if you prefer to work with degrees, you can use the `MathHelper.ToRadians()` method to convert your values when supplying them to XNA classes and methods.

The final declaration, `rotationTicksRemaining`, is reduced by one each time the piece is updated. When this counter reaches zero, the piece has finished animating.

When the piece is drawn, the `RotationAmount` property is referenced by a `spriteBatch.Draw()` call and returns either the `rotationAmount` property (in the case of a clockwise rotation) or 2π (a full circle) minus the `rotationAmount` if the rotation is counter-clockwise.

The constructor in step 7 illustrates how the parameters passed to a constructor can be forwarded to the class' parent constructor via the `:base` specification. Since the `GamePiece` class has a constructor that accepts a piece type, we can pass that information along to its constructor while using the second parameter (clockwise) to update the clockwise member that does not exist in the `GamePiece` class. In this case, since both the clockwise member and the clockwise parameter have identical names, we specify `this.clockwise` to refer to the clockwise member of the `RotatingPiece` class. Simply `clockwise` in this scope refers only to the parameter passed to the constructor.



this notation

You can see that it is perfectly valid C# code to have method parameter names that match the names of class variables, thus potentially hiding the class variables from being used in the method (since referring to the name inside the method will be assumed to refer to the parameter). To ensure that you can always access your class variables even when a parameter name conflicts, you can preface the variable name with `this.` when referring to the class variable. `this.` indicates to C# that the variable you want to use is part of the class, and not a local method parameter.

Lastly, the `UpdatePiece()` method simply increases the `rotationAmount` member while decreasing the `rotationTicksRemaining` counter (using `MathHelper.Max()` to ensure that the value does not fall below zero).

Time for action – falling pieces

1. Add a new class to the Flood Control project called "FallingPiece".
2. Add `using Microsoft.Xna.Framework;` to the using area at the top of the class.

3. Update the declaration of the class to read `class FallingPiece : GamePiece`

4. Add the following declarations to the `FallingPiece` class:

```
public int VerticalOffset;
public static int fallRate = 5;
```

5. Add a constructor for the `FallingPiece` class:

```
public FallingPiece(string pieceType, int verticalOffset)
    : base(pieceType)
{
    VerticalOffset = verticalOffset;
}
```

6. Add a method to update the piece:

```
public void UpdatePiece()
{
    VerticalOffset = (int)MathHelper.Max(
        0,
        VerticalOffset - fallRate);
}
```

What just happened?

Simpler than a `RotatingPiece`, a `FallingPiece` is also a child of the `GamePiece` class. A falling piece has an offset (how high above its final destination it is currently located) and a falling speed (the number of pixels it will move per update).

As with a `RotatingPiece`, the constructor passes the `pieceType` parameter to its base class constructor and uses the `verticalOffset` parameter to set the `VerticalOffset` member. Note that the capitalization on these two items differs. Since `VerticalOffset` is declared as public and therefore capitalized by common C# convention, there is no need to use the "this" notation, since the two variables technically have different names.

Lastly, the `UpdatePiece()` method subtracts `fallRate` from `VerticalOffset`, again using the `MathHelper.Max()` method to ensure the offset does not fall below zero.

Time for action – fading pieces

1. Add a new class to the Flood Control project called "FadingPiece".
2. Add `using Microsoft.Xna.Framework;` to the using area at the top of the class.
3. Update the declaration of the class to read `class FadingPiece : GamePiece`

4. Add the following declarations to the `FadingPiece` class:

```
public float alphaLevel = 1.0f;
public static float alphaChangeRate = 0.02f;
```

5. Add a constructor for the `FadingPiece` class:

```
public FadingPiece(string pieceType, string suffix)
    : base(pieceType, suffix)
{
}
}
```

6. Add a method to update the piece:

```
public void UpdatePiece()
{
    alphaLevel = MathHelper.Max(
        0,
        alphaLevel - alphaChangeRate);
}
```

What just happened?

The simplest of our animated pieces, the `FadingPiece` only requires an alpha value (which always starts at `1.0f`, or fully opaque) and a rate of change. The `FadingPiece` constructor simply passes the parameters along to the base constructor.

When a `FadingPiece` is updated, `alphaLevel` is reduced by `alphaChangeRate`, making the piece more transparent.

Managing animated pieces

Now that we can create animated pieces, it will be the responsibility of the `GameBoard` class to keep track of them. In order to do that, we will define a `Dictionary` object for each type of piece.

A `Dictionary` is a collection object similar to a `List`, except that instead of being organized by an index number, a dictionary consists of a set of key and value pairs. In an array or a `List`, you might access an entity by referencing its index as in `dataValues[2] = 12`; With a `Dictionary`, the index is replaced with your desired key type. Most commonly this will be a string value. This way, you can do something like `fruitColors["Apple"]="red"`;

Time for action – updating GameBoard to support animated pieces

1. In the declarations section of the GameBoard class, add three dictionaries:

```
public Dictionary<string, FallingPiece> fallingPieces =
    new Dictionary<string, FallingPiece>();
public Dictionary<string, RotatingPiece> rotatingPieces =
    new Dictionary<string, RotatingPiece>();
public Dictionary<string, FadingPiece> fadingPieces =
    new Dictionary<string, FadingPiece>();
```

2. Add methods to the GameBoard class to create new falling piece entries in the dictionaries:

```
public void AddFallingPiece(int X, int Y,
    string PieceName, int VerticalOffset)
{
    fallingPieces[X.ToString() + "_" + Y.ToString()] = new
        FallingPiece(PieceName, VerticalOffset);
}

public void AddRotatingPiece(int X, int Y,
    string PieceName, bool Clockwise)
{
    rotatingPieces[X.ToString() + "_" + Y.ToString()] = new
        RotatingPiece(PieceName, Clockwise);
}

public void AddFadingPiece(int X, int Y, string PieceName)
{
    fadingPieces[X.ToString() + "_" + Y.ToString()] = new
        FadingPiece(PieceName, "W");
}
```

3. Add the ArePiecesAnimating() method to the GameBoard class:

```
public bool ArePiecesAnimating()
{
    if ((fallingPieces.Count == 0) &&
        (rotatingPieces.Count == 0) &&
        (fadingPieces.Count == 0))
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

4. Add the UpdateFadingPieces() method to the GameBoard class:

```
private void UpdateFadingPieces()
{
    Queue<string> RemoveKeys = new Queue<string>();

    foreach (string thisKey in fadingPieces.Keys)
    {
        fadingPieces[thisKey].UpdatePiece();

        if (fadingPieces[thisKey].alphaLevel == 0.0f)
            RemoveKeys.Enqueue(thisKey.ToString());
    }

    while (RemoveKeys.Count > 0)
        fadingPieces.Remove(RemoveKeys.Dequeue());
}
```

5. Add the UpdateFallingPieces() method to the GameBoard class:

```
private void UpdateFallingPieces()
{
    Queue<string> RemoveKeys = new Queue<string>();

    foreach (string thisKey in fallingPieces.Keys)
    {
        fallingPieces[thisKey].UpdatePiece();

        if (fallingPieces[thisKey].VerticalOffset == 0)
            RemoveKeys.Enqueue(thisKey.ToString());
    }

    while (RemoveKeys.Count > 0)
        fallingPieces.Remove(RemoveKeys.Dequeue());
}
```

6. Add the UpdateRotatingPieces() method to the GameBoard class:

```
private void UpdateRotatingPieces()
{
    Queue<string> RemoveKeys = new Queue<string>();

    foreach (string thisKey in rotatingPieces.Keys)
    {
        rotatingPieces[thisKey].UpdatePiece();

        if (rotatingPieces[thisKey].rotationTicksRemaining == 0)
            RemoveKeys.Enqueue(thisKey.ToString());
    }

    while (RemoveKeys.Count > 0)
        rotatingPieces.Remove(RemoveKeys.Dequeue());
}
```

7. Add the `UpdateAnimatedPieces()` method to the `GameBoard` class:

```

public void UpdateAnimatedPieces()
{
    if (fadingPieces.Count == 0)
    {
        UpdateFallingPieces();
        UpdateRotatingPieces();
    }
    else
    {
        UpdateFadingPieces();
    }
}

```

What just happened?

After declaring the three `Dictionary` objects, we have three methods used by the `GameBoard` class to create them when necessary. In each case, the key is built in the form "X_Y", so an animated piece in column 5 on row 4 will have a key of "5_4". Each of the three `Add...` methods simply pass the parameters along to the constructor for the appropriate piece types after determining the key to use.

When we begin drawing the animated pieces, we want to be sure that animations finish playing before responding to other input or taking other game actions (like creating new pieces). The `ArePiecesAnimating()` method returns "true" if any of the `Dictionary` objects contain entries. If they do, we will not process any more input or fill empty holes on the game board until they have completed.

The `UpdateAnimatedPieces()` method will be called from the game's `Update()` method and is responsible for calling the three different update methods above (`UpdateFadingPiece()`, `UpdateFallingPiece()`, and `UpdateRotatingPiece()`) for any animated pieces currently on the board. The first line in each of these methods declares a `Queue` object called `RemoveKeys`. We will need this because C# does not allow you to modify a `Dictionary` (or `List`, or any of the similar "generic collection" objects) while a `foreach` loop is processing them.

A `Queue` is yet another generic collection object that works like a line at the bank. People stand in a line and await their turn to be served. When a bank teller is available, the first person in the line transacts his/her business and leaves. The next person then steps forward. This type of processing is known as FIFO, or First In, First Out.

Using the `Enqueue()` and `Dequeue()` methods of the `Queue` class, objects can be added to the `Queue` (`Enqueue()`) where they await processing. When we want to deal with an object, we `Dequeue()` the oldest object in the `Queue` and handle it. `Dequeue()` returns the first object waiting to be processed, which is the oldest object added to the `Queue`.



Collection classes

C# provides a number of different "collection" classes, such as the `Dictionary`, `Queue`, `List`, and `Stack` objects. Each of these objects provides different ways to organize and reference the data in them. For information on the various collection classes and when to use each type, see the following MSDN entry: [http://msdn.microsoft.com/en-us/library/6tc79sx1\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/6tc79sx1(VS.80).aspx)

Each of the update methods loops through all of the keys in its own `Dictionary` and in turn calls the `UpdatePiece()` method for each key. Each piece is then checked to see if its animation has completed. If it has, its key is added to the `RemoveKeys` queue. After all of the pieces in the `Dictionary` have been processed, any keys that were added to `RemoveKeys` are then removed from the `Dictionary`, eliminating those animated pieces.

If there are any `FadingPieces` currently active, those are the only animated pieces that `UpdateAnimatedPieces()` will update. When a row is completed, the scoring tiles fade out, the tiles above them fall into place, and new tiles fall in from above. We want all of the fading to finish before the other tiles start falling (or it would look strange as the new tiles pass through the fading old tiles).

Fading pieces

In the discussion of `UpdateAnimatedPieces()`, we stated that fading pieces are added to the board whenever the player completes a scoring chain. Each piece in the chain is replaced with a fading piece.

Time for action – generating fading pieces

1. In the `Game1` class, modify the `CheckScoringChain()` method by adding the following call inside the `foreach` loop before the square is set to "Empty":

```
gameBoard.AddFadingPiece(  
    (int) ScoringSquare.X,  
    (int) ScoringSquare.Y,  
    gameBoard.GetSquare(  
        (int) ScoringSquare.X,  
        (int) ScoringSquare.Y));
```

What just happened?

Adding fading pieces is simply a matter of getting the square (before it is replaced with an empty square) and adding it to the `FadingPieces` dictionary. We need to use the `(int)` typecasts because the `ScoringSquare` variable is a `Vector2` value, which stores its X and Y components as floats.

Falling pieces

Falling pieces are added to the game board in two possible locations: From the `FillFromAbove()` method when a piece is being moved from one location on the board to another, and in the `GenerateNewPieces()` method, when a new piece falls in from the top of the game board.

Time for action – generating falling pieces

1. Modify the `FillFromAbove()` method of the `GameBoard` class by adding a call to generate falling pieces right before the `rowLookup = -1;` line:

```
AddFallingPiece(x, y, GetSquare(x, y),
    GamePiece.PieceHeight * (y-rowLookup));
```
2. Update the `GenerateNewPieces()` method by adding the following call right after the `RandomPiece(x,y)` line:

```
AddFallingPiece(x, y, GetSquare(x, y),
    GamePiece.PieceHeight * GameBoardHeight);
```

What just happened?

When `FillFromAbove()` moves a piece downward, we now create an entry in the `FallingPieces` dictionary that is equivalent to the newly moved piece. The vertical offset is set to the height of a piece (40 pixels) times the number of board squares the piece was moved. For example, if the empty space was at location 5,5 on the board, and the piece above it (5,4) is being moved down one block, the animated piece is created at 5,5 with an offset of 40 pixels ($5-4 = 1$, times 40).

When new pieces are generated for the board, they are added with an offset equal to the height (in pixels) of the game board, determined by multiplying the `GamePiece.PieceHeight` value by the `GameBoardHeight`. This means they will always start above the playing area and fall into it.

Rotating pieces

The last type of animated piece we need to deal with adding during play is the rotation piece. This piece type is added whenever the user clicks on a game piece.

Time for action – modify Game1 to generate rotating pieces

1. Update the `HandleMouseInput()` method in the `Game1` class to add rotating pieces to the board by adding the following inside the `if (mouseState.LeftButton == ButtonState.Pressed)` block before `gameBoard.RotatePiece()` is called:
2. Still in `HandleMouseInput()`, add the following in the same location inside the `if` block for the right mouse button:

```
gameBoard.AddRotatingPiece(x, y,  
    gameBoard.GetSquare(x, y), false);
```

```
gameBoard.AddRotatingPiece(x, y,  
    gameBoard.GetSquare(x, y), true);
```

What just happened?

Recall that the only difference between a clockwise rotation and a counter-clockwise rotation (from the standpoint of the `AddRotatingPiece()` method) is a `true` or `false` in the final parameter. Depending on which button is clicked, we simply add the current square (before it gets rotated, otherwise the starting point for the animation would be the final position) and `"true"` for right mouse clicks or `"false"` for left mouse clicks.

Calling UpdateAnimatedPieces()

In order for the `UpdateAnimatedPieces()` method of the `GameBoard` class to run, the game's `Update()` method needs to be modified to call it.

Time for action – updating Game1 to update animated pieces

1. Modify the `Update()` method of the `Game1` class by replacing the current `case` statement for the `GameState.Playing` state with:

```
case GameStates.Playing:  
    timeSinceLastInput +=  
        (float)gameTime.ElapsedGameTime.TotalSeconds;  
  
    if (gameBoard.ArePiecesAnimating())  
    {
```

```

        gameBoard.UpdateAnimatedPieces();
    }
    else
    {
        gameBoard.ResetWater();

        for (int y = 0; y < GameBoard.GameBoardHeight; y++)
        {
            CheckScoringChain(gameBoard.GetWaterChain(y));
        }

        gameBoard.GenerateNewPieces(true);

        if (timeSinceLastInput >= MinTimeSinceLastInput)
        {
            HandleMouseInput(Mouse.GetState());
        }
    }

    break;

```

What just happened?

This method is very similar to its previous incarnation. In this instance, we check to see if there are outstanding animated pieces to process. If there are, `UpdateAnimatedPieces()` is run. If no animated pieces currently exist, the previous behaviour of the `GameStates.Playing` case is executed.

Drawing animated pieces

Our animated pieces are almost completed. In fact, they all function right now but you cannot see them because we have not yet updated `Draw()` to take them into account.

Time for action – update Game1 to draw animated pieces

1. Add methods to the `Game1` class to draw each potential type of game piece (animated and non-animated):

```

private void DrawEmptyPiece(int pixelX, int pixelY)
{
    spriteBatch.Draw(
        playingPieces,
        new Rectangle(pixelX, pixelY,
            GamePiece.PieceWidth, GamePiece.PieceHeight),

```

```
        EmptyPiece,
        Color.White);
    }

    private void DrawStandardPiece(int x, int y,
        int pixelX, int pixelY)
    {
        spriteBatch.Draw(
            playingPieces, new Rectangle(pixelX, pixelY,
                GamePiece.PieceWidth, GamePiece.PieceHeight),
            gameBoard.GetSourceRect(x, y),
            Color.White);
    }

    private void DrawFallingPiece(int pixelX, int pixelY,
        string positionName)
    {
        spriteBatch.Draw(
            playingPieces,
            new Rectangle(pixelX, pixelY -
                gameBoard.fallingPieces[positionName].VerticalOffset,
                GamePiece.PieceWidth, GamePiece.PieceHeight),
            gameBoard.fallingPieces[positionName].GetSourceRect(),
            Color.White);
    }

    private void DrawFadingPiece(int pixelX, int pixelY,
        string positionName)
    {
        spriteBatch.Draw(
            playingPieces,
            new Rectangle(pixelX, pixelY,
                GamePiece.PieceWidth, GamePiece.PieceHeight),
            gameBoard.fadingPieces[positionName].GetSourceRect(),
            Color.White *
                gameBoard.fadingPieces[positionName].alphaLevel);
    }

    private void DrawRotatingPiece(int pixelX, int pixelY,
        string positionName)
    {
        spriteBatch.Draw(
            playingPieces,
            new Rectangle(pixelX + (GamePiece.PieceWidth / 2),
                pixelY + (GamePiece.PieceHeight / 2),
                GamePiece.PieceWidth,
                GamePiece.PieceHeight),
            gameBoard.rotatingPieces[positionName].GetSourceRect(),
            Color.White,
            gameBoard.rotatingPieces[positionName].RotationAmount,
```

```

        new Vector2(GamePiece.PieceWidth / 2,
                    GamePiece.PieceHeight / 2),
        SpriteEffects.None, 0.0f);
    }

```

- 2.** Modify the `Draw()` method of the `Game1` class by replacing the `for` loop that currently draws the playing pieces with:

```

for (int x = 0; x < GameBoard.GameBoardWidth; x++)
    for (int y = 0; y < GameBoard.GameBoardHeight; y++)
    {
        int pixelX = (int)gameBoardDisplayOrigin.X +
            (x * GamePiece.PieceWidth);
        int pixelY = (int)gameBoardDisplayOrigin.Y +
            (y * GamePiece.PieceHeight);

        DrawEmptyPiece(pixelX, pixelY);

        bool pieceDrawn = false;

        string positionName = x.ToString() + "_" + y.ToString();

        if (gameBoard.rotatingPieces.ContainsKey(positionName))
        {
            DrawRotatingPiece(pixelX, pixelY, positionName);
            pieceDrawn = true;
        }

        if (gameBoard.fadingPieces.ContainsKey(positionName))
        {
            DrawFadingPiece(pixelX, pixelY, positionName);
            pieceDrawn = true;
        }

        if (gameBoard.fallingPieces.ContainsKey(positionName))
        {
            DrawFallingPiece(pixelX, pixelY, positionName);
            pieceDrawn = true;
        }

        if (!pieceDrawn)
        {
            DrawStandardPiece(x, y, pixelX, pixelY);
        }
    }

```

- 3.** Try it out! Run your game and complete a few rows.

What just happened?

To keep things organized, we have split the drawing of each of the different potential piece types into its own small method. These methods (`DrawEmptyPiece()`, `DrawStandardPiece()`, `DrawFallingPiece()`, `DrawFadingPiece()`, and `DrawRotatingPiece()`) each contain only a single statement to draw the piece.

Before we look at how each of the pieces is actually drawn, let's examine the way we determine which of these methods to call when drawing a piece. The structure of the drawing loop is still the same as it was before we added animated pieces: each square on the board is looped through, with a blank square being drawn first in each position.

After the blank space, a new Boolean value called `pieceDrawn` is declared and set to false. If an animated piece occupies a square, only the animated piece will be drawn, and not the underlying game piece.

The reason for this is that when the user clicks on the mouse button to rotate a piece, in memory the piece is rotated immediately. The animated piece that the user sees is inserted into the drawing process so it looks like the piece is turning. If both the animated piece and the real underlying piece were to be drawn, the final rotation position would be visible overlaid on top of the rotating piece while the rotation animation was playing.

The `positionName` string contains the dictionary key for the space we are currently drawing (in "X_Y" format). We use this to check each of the animated piece dictionaries to see if they contain an entry for that key.

If they do, the animated piece is drawn and the `pieceDrawn` variable is set to true. If the piece still has not been drawn after all of the dictionaries have been checked, the base piece is drawn just as it was before.

SpriteBatch overloads

Both falling and fading pieces are drawn using the `SpriteBatch.Draw()` overload that we are already familiar with; where a `Texture2D`, destination `Rectangle`, source `Rectangle`, and `Color` are specified when drawing. By multiplying our base drawing color (white) by the alpha value for a fading piece, we cause the whole piece to be drawn partially transparent. As the time passes, the alpha value will reach zero, and the piece will be fully transparent.

However, rotated pieces need to use an extended version of the `SpriteBatch.Draw()` call. The first four parameters are the same as our existing `Draw()` calls. To these parameters, we add a float for the rotation amount, a `Vector2` for the origin around which the rotation takes place, a `SpriteEffects` property (set to `SpriteEffects.None` in this case) and a sorting depth (set to 0, or the top level).

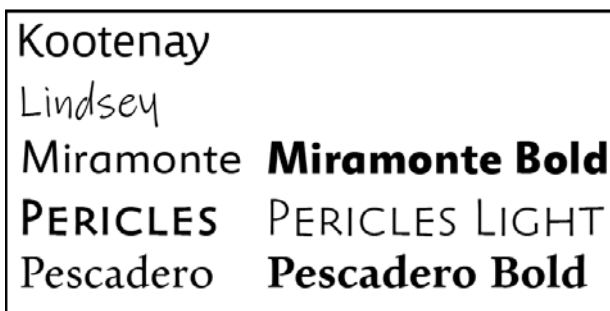
When using a rotation with this form of the `SpriteBatch.Draw()` call, it is necessary to specify the point around which the sprite should be rotated. If we were to set the origin to `Vector2.Zero` (equivalent to 0, 0) the sprite would rotate around the upper left corner of the image, swinging into the spaces of other tiles on the board. The center point of the sprite is specified in local sprite coordinates (as opposed to screen coordinates, or even coordinates within the texture the sprite is being pulled from). The local coordinates of the sprite range from 0, 0 in the upper left corner to the height and width of the sprite in the lower right. In our case, the lower right corner of the sprite is `GamePiece.PieceWidth`, `GamePiece.PieceHeight`, or 40, 40.

By specifying new `Vector2(GamePiece.PieceWidth/2, GamePiece.PieceHeight/2)` we are setting the origin to the center of the sprite, meaning it will rotate in place as expected.

SpriteFonts

Unlike a Windows Forms application, XNA cannot use the TrueType fonts that are installed on your computer. In order to use a font, it must first be converted into a `SpriteFont`, a bitmap based representation of the font in a particular size that can be drawn with the `SpriteBatch.DrawString()` command.

Technically, any Windows font can be turned into a `SpriteFont`, but licensing restrictions on most fonts will prevent you from using them in your XNA games. The redistributable font package you installed in *Chapter 1* is provided by Microsoft to address this problem and give XNA developers a range of usable fonts that can be included in XNA games. Following are samples of each of the fonts included in the font package:



Time for action – add SpriteFonts to Game1

1. Right click on the `Fonts` folder in the `Content` project in `Solution Explorer` and select **Add | New Item**.
2. From the **Add New Item** dialog, select **Sprite Font**.
3. Name the font **Pericles36.spritefont**. After adding the font, the `spritefont` file will open in the editor window.
4. In the `spritefont` file, change `<Fontname>Kootenay</Fontname>` to `<Fontname>Pericles</Fontname>`.
5. Change `<Size>14</Size>` to `<Size>36</Size>`.
6. Add the following declaration to the `Game1` class:

```
SpriteFont pericles36Font;
```
7. Update the `LoadContent()` method of the `Game1` class to load `spritefont` by adding:

```
pericles36Font = Content.Load<SpriteFont>(@"Fonts\Pericles36");
```

What just happened?

Adding a `SpriteFont` to your game is very similar to adding a texture image. Since both are managed by the `Content Pipeline`, working with them is identical from a code standpoint. In fact, `SpriteFonts` are really just specialized sprite sheets, similar to what we used for our game pieces, and are drawn via the same `SpriteBatch` class we use to draw our sprites.

The `.spritefont` file that gets added to your project is actually an XML document containing information that the `Content Pipeline` uses to create the `.XNB` file that holds the bitmap information for the font when you compile your code. The `.spritefont` file is copied from a template, so no matter what you call it, the XML will always default to 14 point `Kootenay`. In steps 4 and 5, we will edit the XML to generate 36 point `Pericles` instead.

Just as with a `Texture2D`, we declare a variable (this time a `SpriteFont`) to hold the `Pericles 36 point` font. The `Load()` method of the `Content` object is used to load the font.



SpriteFonts and extended characters

When a SpriteFont is built by the Content Processor, it actually generates bitmap images for each of the characters in the font. The range of characters generated is controlled by the `<CharacterRegions>` section in the SpriteFont's XML description. If you attempt to output a character not covered by this range, your game will crash. You can avoid this by removing the HTML comment characters (`<!--` and `-->`) from around the `<DefaultCharacter>` definition in the XML file. Whenever an unknown character is output, the character defined in `<DefaultCharacter>` will be used in its place.

Score display

Displaying the player's score with our new SpriteFont is simply a matter of calling the `SpriteBatch.DrawString()` method.

Time for action – drawing the score

1. Add a new `Vector2` to the declarations area of `Game1` to store the screen location where the score will be drawn:

```
Vector2 scorePosition = new Vector2(605, 215);
```

2. In the `Draw()` method, remove `"this.Window.Title = playerScore.ToString();"` and replace the line with:

```
spriteBatch.DrawString(pericles36Font,
    playerScore.ToString(),
    scorePosition,
    Color.Black);
```

What just happened?

Using named vectors to store things like text positions, allows you to easily move them around later if you decide to modify the layout of your game screen. It also makes code more readable, as we have the name `scorePosition` instead of a hard-coded vector value in the `spriteBatch.DrawString()` call. Since our window size is set to 800 by 600 pixels, the location we have defined above will place the score into the pre-defined score box on our background image texture.

The `DrawString()` method accepts a font to draw with (`pericles36Font`), a string to output (`playerScore.ToString()`), a `Vector2` specifying the upper left corner of the location to begin drawing (`scorePosition`), and a color for the text to be drawn in (`Color.Black`).

ScoreZooms

Simply drawing the player's score is not very exciting, so let's add another use for our `SpriteFont`. In some puzzle games, when the player scores, the number of points earned is displayed in the center of the screen, rapidly growing larger and expanding until it flies off of the screen toward the player.

We will implement this functionality with a class called `ScoreZoom` that will handle scaling the font.

Time for action – creating the `ScoreZoom` class

1. Add a new class file called `ScoreZoom.cs` to the `Game1` class.

2. Add the following `using` directive to the top of the file:

```
using Microsoft.Xna.Framework.Graphics;
```

3. Add the following declarations to the `ScoreZoom` class:

```
public string Text;
public Color DrawColor;
private int displayCounter;
private int maxDisplayCount = 30;
private float scale = 0.4f;
private float lastScaleAmount = 0.0f;
private float scaleAmount = 0.4f;
```

4. Add the `Scale` read-only property to the `ScoreZoom` class:

```
public float Scale
{
    get { return scaleAmount * displayCounter; }
}
```

5. Add a Boolean property to indicate when the `ScoreZoom` has finished displaying:

```
public bool IsCompleted
{
    get { return (displayCounter > maxDisplayCount); }
}
```

6. Create a constructor for the `ScoreZoom` class:

```
public ScoreZoom(string displayText, Color fontColor)
{
    Text = displayText;
    DrawColor = fontColor;
    displayCounter = 0;
}
```

7. Add an Update() method to the ScoreZoom class:

```
public void Update()
{
    scale += lastScaleAmount + scaleAmount;
    lastScaleAmount += scaleAmount;
    displayCounter++;
}
```

What just happened?

The ScoreZoom class holds some basic information about a piece of text and how it will be displayed to the screen. The number of frames the text will be drawn for are determined by `displayCounter` and `maxDisplayCount`.

To manage the scale, three variables are used: `scale` contains the actual scale size that will be used when drawing the text, `lastScaleAmount` holds the amount the scale was increased by during the previous frame, and `scaleAmount` determines the growth in the scale factor during each frame.

You can see how this is used in the `Update()` method. The current scale is increased by both the `lastScaleAmount` and `scaleAmount`. `lastScaleAmount` is then increased by the scale amount. This results in the scale growing in an exponential fashion instead of increasing linearly by a `scaleAmount` for each frame. This will give the text a zooming effect as it starts growing slowly and then speeds up rapidly to fill the screen.

Time for action – updating and displaying ScoreZooms**1. Add a Queue object to the Game1 class to hold active ScoreZooms:**

```
Queue<ScoreZoom> ScoreZooms = new Queue<ScoreZoom>();
```

2. Add a new helper method to the Game1 class to update the ScoreZooms queue:

```
private void UpdateScoreZooms()
{
    int dequeueCounter = 0;
    foreach (ScoreZoom zoom in ScoreZooms)
    {
        zoom.Update();
        if (zoom.IsCompleted)
            dequeueCounter++;
    }
    for (int d = 0; d < dequeueCounter; d++)
        ScoreZooms.Dequeue();
}
```

3. In the `Update()` method, inside the case section for `GameState.Playing`, add the call to update any active `ScoreZooms`. This can be placed right before the case's `break;` statement:

```
UpdateScoreZooms();
```

4. Add the following to the `CheckScoringChain()` method to create a `ScoreZoom` when the player scores. Add this right after the `playerScore` is increased:

```
ScoreZooms.Enqueue(new ScoreZoom("+" +  
    DetermineScore(WaterChain.Count).ToString(),  
    new Color(1.0f, 0.0f, 0.0f, 0.4f)));
```

5. Modify the `Draw()` method of the `Game1` class by adding the following right before the `SpriteBatch.DrawString()` call which draws the player's score:

```
foreach (ScoreZoom zoom in ScoreZooms)  
{  
    spriteBatch.DrawString(pericles36Font, zoom.Text,  
        new Vector2(this.Window.ClientBounds.Width / 2,  
            this.Window.ClientBounds.Height / 2),  
        zoom.DrawColor, 0.0f,  
        new Vector2(pericles36Font.MeasureString(zoom.Text).X / 2,  
            pericles36Font.MeasureString(zoom.Text).Y / 2),  
        zoom.Scale, SpriteEffects.None, 0.0f);  
}
```

What just happened?

Since all `ScoreZoom` objects "live" for the same amount of time, we can always be certain that the first one we create will finish before any created during a later loop. This allows us to use a simple `Queue` to hold `ScoreZooms` since a `Queue` works in a first-in-first-out manner.

When `UpdateScoreZooms()` is executed, the `dequeueCounter` holds the number of `ScoreZoom` objects that have finished updating during this cycle. It starts at zero, and while the `foreach` loop runs, any `ScoreZoom` that has an `IsCompleted` property of `true` increments the counter. When the `foreach` has completed, `ScoreZooms.Dequeue()` is run a number of times equal to `dequeueCounter`.

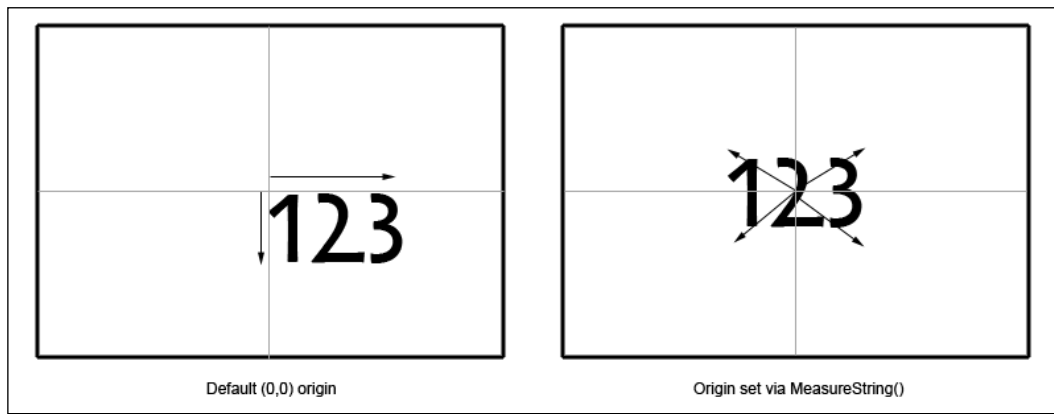
Adding new `ScoreZoom` objects is accomplished in step 4, with the `Enqueue()` method. The method is passed a new `ScoreZoom` object, which is constructed with a plus sign (+) and the score being added, followed by a red color with the alpha value set to `0.4f`, making it a little more than halfway transparent.

Just as the `SpriteBatch.Draw()` method has multiple overloads, so does the `SpriteBatch.DrawString()` method, and in fact, they follow much the same pattern. This form of the `DrawString()` method accepts the `SpriteFont` (`pericles36Font`), the text to display, a location vector, and a draw color just like the previous call.

For the draw location in this case, we use `this.Window.ClientBounds` to retrieve the width and height of the game window. By dividing each by two, we get the coordinates of the center of the screen.

The remaining parameters are the same as those of the extended `Draw()` call we used to draw rotated pieces. After the color value is rotation, which we have set to `0.0f`, followed by the origin point for that rotation. We have used the `MeasureString()` method of the `SpriteFont` class to determine both the height and width of the text that will be displayed and divided the value by two to determine the center point of the text. Why do this when there is no rotation happening? Despite what the order of the parameters might indicate, this origin also impacts the next parameter: the scale.

When the scale is applied, it sizes the text around the origin point. If we were to leave the origin at the default **(0, 0)**, the upper left corner of the text would remain in the center of the screen and it would grow towards the bottom right corner. By setting the origin to the center of the text, the scale is applied evenly in all directions:



Just as with the extended `Draw()` method earlier, we will use `SpriteEffects.None` for the `spriteEffects` parameter and `0.0f` for the layer depth, indicating that the text should be drawn on top of whatever has been drawn already.

Adding the `GameOver` game state

Now that we can draw text, we can add a new game state in preparation for actually letting the game end when the facility floods.

Time for action – game over

1. Modify the declaration of the `GameStates` enum in the `Game1` class to include the `GameOver` state:

```
enum GameStates { TitleScreen, Playing, GameOver };
```

2. Add the following declarations to the `Game1` class:

```
Vector2 gameOverLocation = new Vector2(200, 260);  
float gameOverTimer;
```

3. Modify the `Update()` method of `Game1` by adding a new `case` section for the `GameState.GameOver` state:

```
case GameStates.GameOver:  
    gameOverTimer -= (float)gameTime.ElapsedGameTime.TotalSeconds;  
    if (gameOverTimer <= 0)  
    {  
        gameState = GameStates.TitleScreen;  
    }  
    break;
```

4. Modify the `if` statement in the `Draw()` method of `Game1` for the `GameState.Playing` state from `if (gameState == GameStates.Playing)` to:

```
if ((gameState == GameStates.Playing) ||  
    (gameState == GameStates.GameOver))
```

5. Add a new `if` statement for the `GameState.GameOver` state to the `Draw()` method, right before the call to `Base.Draw(gameTime)`.

```
if (gameState == GameStates.GameOver)  
{  
    spriteBatch.Begin();  
    spriteBatch.DrawString(pericles36Font,  
        "G A M E O V E R!",  
        gameOverLocation,  
        Color.Yellow);  
    spriteBatch.End();  
}
```

What just happened?

With the addition of `GameOver`, we now have a complete cycle of game states. When the program is started, the game begins in the `TitleScreen` state. Pressing the *Space bar* switches from `TitleScreen` to `Playing` state. When the game ends, the state moves to `GameOver`.

The `Update()` method handles the `GameOver` state by decreasing the `gameOverTimer` value until it reaches zero, at which point the state is set back to `TitleScreen`.

While the `Update()` method handles each of the game states in a mutually exclusive manner (the update code for `Playing` will never run when in the `GameOver` state), the `Draw()` method handles things differently.

When in the `GameOver` state, we want to display the text `G A M E O V E R!` on top of the game board. The location of the text, defined as (200, 260) in our declarations area, places it in the upper half of the screen, covering the center horizontally. We need to execute the drawing code for the `Playing` state in both the `Playing` and `GameOver` states, as well as an additional section of code only for `GameOver`.

The flood

The background story of the game centers on an underwater research laboratory that is slowly flooding, with the player trying to empty out the flood waters before the place fills up.

Up to this point we do not have a representation of that flood in the game, or any incentive for the player to think quickly to find scoring chains.

Time for action – tracking the flood

1. Add the following declarations to the `Game1` class:

```
const float MaxFloodCounter = 100.0f;
float floodCount = 0.0f;
float timeSinceLastFloodIncrease = 0.0f;
float timeBetweenFloodIncreases = 1.0f;
float floodIncreaseAmount = 0.5f;
```

2. In the `Update()` method of `Game1.cs`, add the following code to keep track of the increasing flood waters right after the `timeSinceLastInput` variable is updated in the `GameState.Playing` case section:

```
timeSinceLastFloodIncrease +=
    (float)gameTime.ElapsedGameTime.TotalSeconds;

if (timeSinceLastFloodIncrease >= timeBetweenFloodIncreases)
{
    floodCount += floodIncreaseAmount;
    timeSinceLastFloodIncrease = 0.0f;
    if (floodCount >= MaxFloodCounter)
    {
        gameOverTimer = 8.0f;
        gameState = GameStates.GameOver;
    }
}
```


3. Update the `CheckScoringChain()` method of the `Game1` class by adding the following to decrease the flood counter when the player scores. Place this code right after `playerScore += DetermineScore(WaterChain.Count);`

```
floodCount = MathHelper.Clamp(floodCount -  
    (DetermineScore(WaterChain.Count)/10), 0.0f, 100.0f);
```

What just happened?

The flood itself is represented as a percentage. When the `floodCount` reaches 100 (`MaxFloodCounter`), the laboratory has completely flooded and the game is over. In addition to these two declarations, we also need to track how rapidly the flood increases (`timeSinceLastFloodIncrease` and `timeBetweenFloodIncreases`), and the rate at which the water rises (`floodIncreaseAmount`).

The timing on the flood increases is handled the same way input pacing is handled: a timer is incremented based on the elapsed game time until it reaches a threshold value. When it does, the timer is reset, and the `floodCount` variable is increased by the `floodIncreaseAmount` value.

When this increase takes place, we check to see if the `floodCount` has reached `MaxFloodCount`, indicating that the facility is flooded. If it has, an eight second timer is set for `gameOverTimer` and the game state is set to `GameOver`. Recall that in the `GameOver` handler, the `gameOverTimer` determines how long the `G A M E O V E R!` text will be displayed before the game switches back to the title screen.

Finally, in step three, the `floodCount` variable needs to be decreased each time the player completes a scoring chain. `MathHelper.Clamp()` is used to subtract the score value (divided by 10) from the `floodCount`, while keeping the value between 0.0f and 100.0f.

Displaying the flood

If you open the `Background.png` file in an image viewer, you will see that there is a full water tank floating inside the space on the playfield where game pieces get displayed. Since we always draw opaque game piece backgrounds over this area, so far we have not seen this portion of the image during game play.

We can use `SpriteBatch.Draw()` to cut out pieces of this full water tank and superimpose it over the empty tank on the right side of the game screen as the facility fills with water. The deeper the water gets, the more of the hidden water tank image we transfer to the visible tank on the screen, working our way up from the bottom:



Time for action – displaying the flood

1. Add the following declarations to the Game1 class:

```
const int MaxWaterHeight = 244;
const int WaterWidth = 297;
```

```
Vector2 waterOverlayStart = new Vector2(85, 245);
Vector2 waterPosition = new Vector2(478, 338);
```

2. Modify the Draw() method of the Game1 class by adding the following right after the spriteBatch.DrawString() call that displays the player's score:

```
int waterHeight = (int)(MaxWaterHeight * (floodCount / 100));

spriteBatch.Draw(backgroundScreen,
    new Rectangle(
        (int)waterPosition.X,
        (int)waterPosition.Y + (MaxWaterHeight - waterHeight),
        WaterWidth,
        waterHeight),
    new Rectangle(
        (int)waterOverlayStart.X,
        (int)waterOverlayStart.Y + (MaxWaterHeight - waterHeight),
        WaterWidth,
        waterHeight),
    new Color(255, 255, 255, 180));
```

3. Try it out! You should now be able to watch the flood slowly increase in the flood tank. When it reaches the top the game should switch to the `GameOver` state and, after an 8 second delay, back to the title screen.

What just happened?

The two `int` values, `MaxWaterHeight`, and `WaterWidth` refer to the size of the water image hidden inside the game board. It is 297 pixels wide, and the full water image is 244 pixels high.

Two vectors are used to store the location of the filled water image (85, 245) and the location that it will be drawn to on the screen (478, 338).

In order to draw the water in the water tank, the `waterHeight` variable, the `MaxWaterHeight` is multiplied by the percentage of water currently in the tank. This results in the number of pixels of water that need to be drawn into the tank.

When determining the source and destination rectangles, the X coordinates are dependant only on the location of the overlay and the drawing position, since they will not change.

The Y coordinates must be modified to pull pixels from the bottom of the image and expand upwards. In order to accomplish this, the current `waterHeight` is subtracted from the `MaxWaterHeight`, and this value is added to the Y coordinate of both vectors.

Difficulty levels

Now that the game can end, we need some way to make the game more difficult the longer the player plays.

After the player has completed 10 scoring chains, the water tank will be emptied, a new set of game pieces will be generated, and the flood will increase faster.

Time for action – adding difficulty levels

1. Add the following declarations to the `Game1` class:

```
int currentLevel = 0;
int linesCompletedThisLevel = 0;

const float floodAccelerationPerLevel = 0.5f;

Vector2 levelTextPosition = new Vector2(512, 215);
```

2. Add the `StartNewLevel()` method to the `Game1` class:

```
private void StartNewLevel()
{
    currentLevel++;
    floodCount = 0.0f;
    linesCompletedThisLevel = 0;
    floodIncreaseAmount += floodAccelerationPerLevel;
    gameBoard.ClearBoard();
    gameBoard.GenerateNewPieces(false);
}
```

3. Modify the `Update()` method of the `Game1` class by updating the case section for `GameState.TitleScreen` to include the following right before the game state is set to `GameState.Playing`:

```
currentLevel = 0;
floodIncreaseAmount = 0.0f;
StartNewLevel();
```

4. Modify the `CheckScoringChain()` method to increment the `linesCompletedThisLevel` variable right after `playerScore += DetermineScore(WaterChain.Count);`

```
linesCompletedThisLevel++;
```

5. Still in the `CheckScoringChain()` method, add the following to call the `StartNewLevel()` method if necessary. Place this code directly after the `foreach` loop that fades out tiles on the board:

```
if (linesCompletedThisLevel >= 10)
{
    StartNewLevel();
}
```

6. Update the `Draw()` method to display the current level in the appropriate location on the screen. Place this code right after the `spriteBatch.DrawString()` call that displays the player's score:

```
spriteBatch.DrawString(pericles36Font,
    currentLevel.ToString(),
    levelTextPosition,
    Color.Black);
```

7. Play! Flood Control is now completed, so try it out!

What just happened?

The current game level and the number of lines the player has completed in the current level are tracked as integers (`currentLevel` and `linesCompletedThisLevel`). The two constants, `baseFloodAmount` and `floodAccelerationPerLevel`, determine how much water is added to the facility every time the flood is updated. Finally, the `levelTextPosition` vector points to the location on the screen where the level number will be displayed.

The `StartNewLevel()` method increases the `currentLevel`, and clears the `floodCount` and `lineCompletedThisLevel` variables. It increases the `floodIncreaseAmount` by the value of `floodAccelerationPerLevel` and then clears the game board. Finally, new pieces are generated for each square on the board.

When beginning a new game (the updates in Step 3) we can simply set `currentLevel` and `floodIncreaseAmount` to zero, and then call the `StartNewLevel()` method. Since both of these variables are increased by `StartNewLevel()` the first level of a new game will begin with the appropriate values.

Step 4 increases the counter that tracks the number of lines the player has completed on the current level every time a scoring chain results in points. Step 5 checks to see if the player has completed 10 or more lines. If they have, a new level is started.

Finally, drawing the level number is a call to the simple form of `SpriteBatch.DrawString()` just as we did for displaying the player's score.

Have a go hero

There are a number of different things you could do to spruce up Flood Control. Here are a few suggestions to try using the knowledge you have gained over these two chapters:

- ◆ Basic—add a "Paused" game state that displays an indication that the game is paused and how to resume play. To prevent cheating, the game board should either not be visible or be obscured in some way while the game is paused.
- ◆ Intermediate—the **Game Over** screen is not very exciting. Create a new bitmap image indicating the aftermath of the flooded facility and display that image instead of the simple **Game Over!** text. You will need to load the image via the `LoadContent()` method and display it when appropriate.

- ◆ Advanced—create an additional "suffix" for pieces that are locked down and cannot be turned. You'll need to expand the `Tile_Sheet.png` file by adding an additional (fourth) column and then copying the first two columns to columns three and four. Draw bolts in the four corners of each of the twelve new piece images and modify the draw code to add an additional 40 pixels to the X value of the source `Rectangle` if the piece contains the locked suffix. Grant extra points for using locked pieces in a scoring chain.

Summary

Chapter 3 has looked at ways to add some polish to the basic Flood Control game that was presented in *Chapter 2*. We have looked at:

- ◆ Creating classes that inherit from existing classes to extend their functionality
- ◆ Using the advanced form of the `SpriteBatch.Draw()` method to add basic animations to the Flood Control game
- ◆ Adding `SpriteFonts` to the project and using them to draw text to the screen
- ◆ Expanding the basic Game State system used in *Chapter 2* to allow for a complete Title | Playing | Game Over cycle
- ◆ Adding increasing levels of difficulty as the player progresses through the game

4

Asteroid Belt Assault – Lost in Space

Mid-way through the hyperspace jump to the rim territories, something went wrong. Your hyperdrive engine shut down and you found yourself suddenly back in normal space, in the middle of a massive asteroid field.

What is worse, the field appears to have been a hiding spot for a group of enemy fighters intercepting communications from the Earth Fleet.

In this chapter, we will begin the construction of Asteroid Belt Assault by:

- ◆ Creating the base Asteroid Belt Assault project and structure
- ◆ Creating a class for animated sprite handling
- ◆ Building a scrolling sprite-based star field background
- ◆ Creating asteroids that can collide with each other
- ◆ Building a "Shot Manager" class that will track projectiles fired by both the player and enemy ships
- ◆ Adding a player controlled star fighter
- ◆ Adding enemy fighters that fly a set of waypoints across the screen

Creating the project

We will need a new Windows Game project for Asteroid Belt Assault. To this project, we need to add the graphics package we will use for the game.

Time for action – creating the Asteroid Belt Assault project

1. Visit <http://www.PacktPub.com> and download the 0669_04_GRAPHICPACK.ZIP file. Extract the file to a temporary location.
2. Open Visual Studio Express Edition and create a new XNA 4.0 Windows Game project called Asteroid Belt Assault.
3. In the Asteroid Belt AssaultContent project, right-click on the project name, select **Add | New Folder** and add a folder called Textures. Add another folder called Fonts.
4. Right-click on Textures and add the SpriteSheet.png and TitleScreen.png files from the graphics pack to the project.
5. Add declarations to the Game1 class for game states and textures:

```
enum GameStates { TitleScreen, Playing, PlayerDead, GameOver};
GameStates gameState = GameStates.TitleScreen;
Texture2D titleScreen;
Texture2D spriteSheet;
```
6. Update the LoadContent() method to load the sprite sheet:

```
titleScreen = Content.Load<Texture2D>(@"Textures\TitleScreen");
spriteSheet = Content.Load<Texture2D>(@"Textures\spriteSheet");
```
7. Add a basic structure for the Update() method before the call to base.Update():

```
switch (gameState)
{
    case GameStates.TitleScreen:
        break;

    case GameStates.Playing:
        break;

    case GameStates.PlayerDead:
        break;

    case GameStates.GameOver:
        break;
}
```
8. Add a basic structure for the Draw() method before the call to base.Draw():

```
spriteBatch.Begin();

if (gameState == GameStates.TitleScreen)
```

```

    {
        spriteBatch.Draw(titleScreen,
            new Rectangle(0, 0, this.Window.ClientBounds.Width,
                this.Window.ClientBounds.Height),
            Color.White);
    }

    if ((gameState == GameStates.Playing) ||
        (gameState == GameStates.PlayerDead) ||
        (gameState == GameStates.GameOver))
    {
    }

    if ((gameState == GameStates.GameOver))
    {
    }

    spriteBatch.End();

```

9. Execute your project to verify that the title screen is displayed. Since the `Update()` method is just a shell at this point, hit *Alt + F4* to close the game window.

What just happened?

The basic project setup and loading of content should be familiar by now. We have added the skeleton of our `Update()` and `Draw()` methods which we will flesh out as we build our game.

Another definition for "sprite"

The term "sprite" predates XNA itself, and it can be used to refer to any method of displaying a bitmap over an existing background. In *Asteroid Belt Assault*, we will expand our use of the term "sprite" to include a class that stores both the bitmap image associated with the sprite, information about its current location on the screen, animation frames, and the code necessary to update and draw the object.

Building the Sprite class

The `Sprite` class will be used as the basis for all of the objects drawn in *Asteroid Belt Assault*. It will support collision detection and frame-based animation.

Time for action – declarations for the Sprite class

1. Add a new class to the project called `Sprite.cs`.

2. Update the class' using area to include:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Add the following declarations to the Sprite class:

```
public Texture2D Texture;  
  
protected List<Rectangle> frames = new List<Rectangle>();  
private int frameWidth = 0;  
private int frameHeight = 0;  
private int currentFrame;  
private float frameTime = 0.1f;  
private float timeForCurrentFrame = 0.0f;  
  
private Color tintColor = Color.White;  
private float rotation = 0.0f;  
  
public int CollisionRadius = 0;  
public int BoundingXPadding = 0;  
public int BoundingYPadding = 0;  
  
protected Vector2 location = Vector2.Zero;  
protected Vector2 velocity = Vector2.Zero;
```

What just happened?

All of the animation frames for any individual sprite will be stored on the same sprite sheet, identified by the `Texture` variable. The `frames` list will hold a single `Rectangle` object for each animation frame defined for the sprite, while `currentFrame` stores the frame that is being displayed at any given time.

In order to control the animation, each frame is displayed for a pre-determined amount of time. This time, stored in `frameTime`, will be compared to `timeForCurrentFrame` to determine when `currentFrame` should be incremented. This is the same timing mechanism we used in Flood Control to pace input. The `frameWidth` and `frameHeight` variables will be assigned when the first frame of animation is established to provide shortcuts to these values.

The `tintColor` and `rotation` members store the implied information that will be used when the sprite is drawn to influence its appearance.

`CollisionRadius`, `BoundingXPadding`, and `BoundingYPadding` will all be used when we implement collision detection. We'll support both bounding circle and bounding box collisions. The `CollisionRadius` defines how large a circle to consider when determining if this sprite has collided with other sprites, while the padding values are used to "shrink" the frame size for collision detection, providing a cushion around the edges of the bounding box for collision purposes.

Finally, the location of the sprite on the screen is tracked via the `location` vector while the speed and direction at which the sprite is moving is stored in `velocity`. The `velocity` vector represents the distance (in pixels) that the sprite will travel in one second of game time.

Time for action – Sprite constructor

1. Add a constructor to the `Sprite` class:

```
public Sprite(
    Vector2 location,
    Texture2D texture,
    Rectangle initialFrame,
    Vector2 velocity)
{
    this.location = location;
    Texture = texture;
    this.velocity = velocity;

    frames.Add(initialFrame);
    frameWidth = initialFrame.Width;
    frameHeight = initialFrame.Height;
}
```

What just happened?

The constructor for the `Sprite` class directly sets the `location`, `texture`, and `velocity` members to the passed parameter values. It then sets the `frameWidth` and `frameHeight` variables by extracting that information from the `Rectangle`.

Time for action – basic Sprite properties

1. Add public properties to allow access to the `Sprite` class' members:

```
public Vector2 Location
{
    get { return location; }
```

```
        set { location = value; }
    }

    public Vector2 Velocity
    {
        get { return velocity; }
        set { velocity = value; }
    }

    public Color TintColor
    {
        get { return tintColor; }
        set { tintColor = value; }
    }

    public float Rotation
    {
        get { return rotation; }
        set { rotation = value % MathHelper.TwoPi; }
    }
}
```

What just happened?

The `Location`, `Velocity`, and `TintColor` properties are simple pass-throughs for their underlying private members as no additional code or checks need to be done when these values are manipulated.

When `Rotation` is set, the value is divided by `MathHelper.TwoPi` and the remainder of the result is stored in the `rotation` member. This is a shorthand way of keeping the value between 0 and 2π . If a value within this range is passed into `Rotation`, the remainder of the division will equal the passed value. If the value is larger than 2π , any full rotations will be removed by the division, leaving only the partial rotation value as the remainder. This method allows external code to increment the value beyond a full rotation and have the sprite class correct the value appropriately.

Time for action – animation and drawing properties

1. Add public properties to allow access to the Sprite class' members:

```
public int Frame
{
    get { return currentFrame; }
    set { currentFrame = (int)MathHelper.Clamp(value, 0,
        frames.Count - 1); }
}

public float FrameTime
{
    get { return frameTime; }
}
```

```

        set { frameTime = MathHelper.Max(0, value); }
    }

    public Rectangle Source
    {
        get { return frames[currentFrame]; }
    }

    public Rectangle Destination
    {
        get
        {
            return new Rectangle(
                (int)location.X,
                (int)location.Y,
                frameWidth,
                frameHeight);
        }
    }

    public Vector2 Center
    {
        get
        {
            return location +
                new Vector2(frameWidth / 2, frameHeight / 2);
        }
    }
}

```

What just happened?

The `set` portion of the `Frame` property uses `MathHelper.Clamp()` to ensure that when it is set, the value stored in `currentFrame` is valid for the `frames` list of `Rectangle`s. This will prevent, for example, setting the frame number to 10 for a sprite with 5 animation frames. The `FrameTime` property allows the speed at which the animation plays to be updated, using `MathHelper.Max()` to ensure the value is greater than or equal to zero. A `FrameTime` of zero will result in an animation that updates its frame during every `Update()` cycle.

The `Source` property returns the `Rectangle` associated with the current frame from the `frames` list, while the `Destination` property builds a new `Rectangle` based on the sprite's current screen location and the width and height of a frame. Because we will be supporting rotation of the sprite, we will not use the `Destination Rectangle` when drawing directly from the `Sprite` class, but it will be available for external code to determine the area containing the sprite on the screen without the padding modifiers of the collision bounding box rectangle which we will see in a moment.

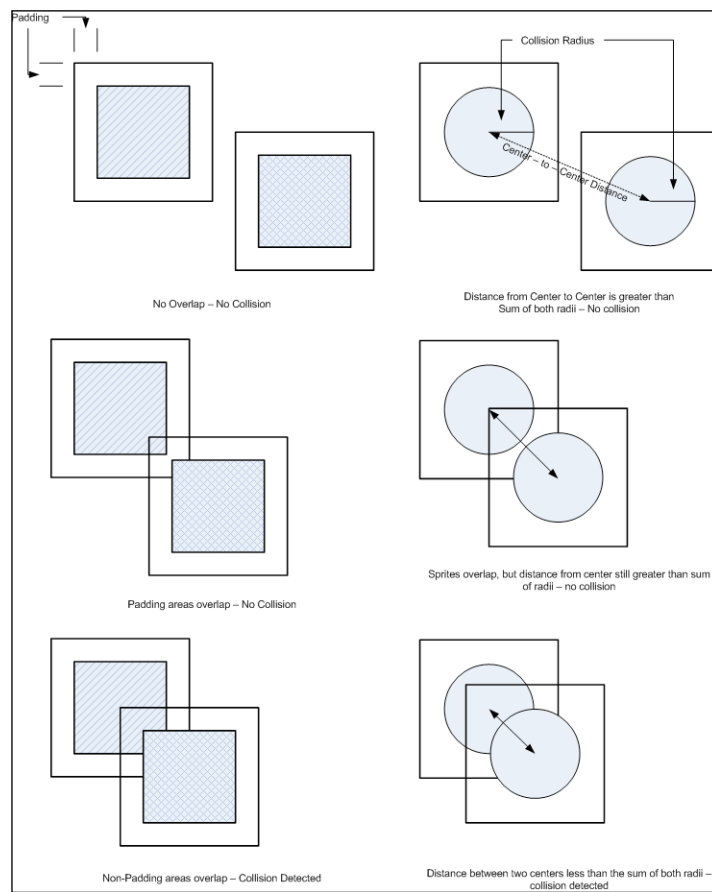
In order to support both rotation and bounding circle collision detection, we will need to be able to determine the screen coordinates of the center of the sprite, so the `Center` property returns the `location` member, offset by half of the width and height of the sprite.

Collision detection

Generally, when two objects in our game world come into contact with each other, we will need some way to know that they have collided so we can take an appropriate action. It might be a player-fired bullet hitting an alien ship, or an asteroid hitting the player, or a number of other events.

Asteroid Belt Assault's Sprite class will have support for both bounding box and bounding circle collisions. In both cases we define a basic shape (a box or a circle) that encompasses the area occupied by the sprite. A similar shape is defined for other objects in the game. When we need to check for collisions, we will compare the bounding shapes of each object with that of each other object. If any of them overlap, we know that a collision has taken place.

Because our sprites will often not fill the entire area allocated to them (otherwise they would simply be a square of color) the Sprite class' bounding box collision detection will support padding values around the edges of the sprite. These values allow for a cushion around the outer edges of a sprite where collisions will be ignored:



Time for action – supporting collision detection

1. Add the properties and methods needed to support collision detection to the `Sprite` class:

```
public Rectangle BoundingBoxRect
{
    get
    {
        return new Rectangle(
            (int)location.X + BoundingXPadding,
            (int)location.Y + BoundingYPadding,
            frameWidth - (BoundingXPadding * 2),
            frameHeight - (BoundingYPadding * 2));
    }
}

public bool IsBoxColliding(Rectangle OtherBox)
{
    return BoundingBoxRect.Intersects(OtherBox);
}

public bool IsCircleColliding(Vector2 otherCenter, float
otherRadius)
{
    if (Vector2.Distance(Center, otherCenter) <
        (CollisionRadius + otherRadius))
        return true;
    else
        return false;
}
```

What just happened?

The `BoundingBoxRect` property provides a `Rectangle` object equivalent to the location and size of the sprite accounting for the padding values around the edges. There is already a `BoundingBox` class as part of the XNA Framework, and while C# will not have a problem with us declaring a parameter with the name of an existing class, we are appending `Rect` to the end of the name here simply to avoid confusion.

When checking for a bounding box collision, the `IsBoxColliding()` method accepts another bounding box and returns true if the two rectangles overlap at any point. The `Rectangle` class' `Intersects()` method contains the logic to check for this overlap.

When performing bounding circle collisions, the `IsCircleColliding()` method accepts a `Vector2` representing the center of the object the sprite will be compared against, and the other object's radius. If the distance between the two centers is less than the sum of the radii of both objects, then the two circles overlap.

Animation and movement

In order to play an animation, we need to be able to add frames to the `frames` list, which requires a simple method.

Time for action – adding animation frames

1. Add the `AddFrame()` method to the `Sprite` class:

```
public void AddFrame(Rectangle frameRectangle)
{
    frames.Add(frameRectangle);
}
```

What just happened?

Adding a frame to the sprite's animation is as simple as adding the corresponding `Rectangle` to the `frames` list. When the animation is updated, the frame value will be compared to the number of entries in `frames` (via `frames.Count`) eliminating the need to store the number of animation frames as a separate value.

Time for action – updating the Sprite

1. Add the `Update()` method to the `Sprite` class:

```
public virtual void Update(GameTime gameTime)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

    timeForCurrentFrame += elapsed;

    if (timeForCurrentFrame >= FrameTime)
    {
        currentFrame = (currentFrame + 1) % (frames.Count);
        timeForCurrentFrame = 0.0f;
    }

    location += (velocity * elapsed);
}
```

What just happened?

When `Update()` is called for the sprite, the standard timing mechanism we have been using is implemented to determine when the frame should be updated. When it is, the frame is set to `(currentFrame + 1) % (frames.Count)`. This is a short-hand method of saying "add 1 to the current frame, divide the total by the number of frames in the animation and return the remainder".

As an example, consider an animation with five frames (numbered zero through four). When `currentFrame` needs to go from zero to one, the assignment looks like this:

```
currentFrame = ( 0 + 1 ) % 5
```

One divided by five is zero, with one as the remainder. When the `currentFrame` has reached the last frame of the animation (frame four), the assignment looks like this:

```
currentFrame = ( 4 + 1 ) % 5
```

Five divided by five is one, with zero as the remainder. Since the remainder is what we are using as the new `currentFrame`, this causes `currentFrame` to be reset back to zero to begin the animation loop again.

After `currentFrame` has been updated, `timeForCurrentFrame` is reset to 0.0f to begin the animation timing loop over again.

Finally, the `Update()` method adds the sprite's `velocity` (adjusted for the time elapsed since the last frame) to the sprite's `location`. Since `velocity` is stored as the change over one second, multiplying it by the `gameTime.ElapsedGameTime.TotalSeconds` (which will, if the game is not running slowly, be equal to 1/60th of a second—the default update rate for an XNA game) determines the distance moved over a single frame.

Why not just specify `velocity` in pixels per frame? In many cases, that may work out just fine, but since you cannot be sure that your game will be running on hardware that can support a full 60 frames per second, if you did not scale your movements to actual elapsed time, your objects would move differently when the game is running at different frame rates, potentially producing a jerky motion.

Time for action – drawing the Sprite

1. Add the `Draw()` method to the `Sprite` class:

```
public virtual void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(
        Texture,
        Center,
```

```
        Source,  
        tintColor,  
        rotation,  
        new Vector2(frameWidth / 2, frameHeight / 2),  
        1.0f,  
        SpriteEffects.None,  
        0.0f);  
    }
```

What just happened?

The `Draw()` method consists of a single call to the `SpriteBatch.Draw()` method, using an overload of the method that allows for rotation and scaling.

Because we are specifying a rotation, we need to identify a center point for the rotation to be based around (`new Vector2(frameWidth / 2, frameHeight / 2)`). Instead of specifying a destination rectangle, we specify a vector that points to the center of the area the object will occupy on the screen (the `Center` property of the `Sprite` class).

A sprite-based star field

Let's put our new `Sprite` class to use by creating a scrolling star field. On the `SpriteSheet.png` file we added to the project earlier, there is a 50 by 50 pixel empty white square located at 0,450 in the image. As we saw in the introductory `SquareChase` game, we can use an empty white sprite in combination with the `TintColor` parameter of the `SpriteBatch.Draw()` method to draw squares of any color we wish. We will make use of this ability to create slight color variations in the stars in the star field to make them look more realistic than if they were simply uniformly colored squares.

In order to create the star field, we will create 200 sprites and place them on the screen randomly. They will have a velocity that will slowly draw them down the screen. The `StarField` class will be responsible for monitoring the stars and determining when they fall off the bottom of the screen. When this happens, they will be placed back at the top of the screen at a random location.

Time for action – creating the StarField class

1. Add a new class called `StarField` to the Asteroid Belt Assault project.
2. Add the following `using` directives to the top of the class file:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Add the following declarations to the `StarField` class:

```
private List<Sprite> stars = new List<Sprite>();
private int screenWidth = 800;
private int screenHeight = 600;
private Random rand = new Random();
private Color[] colors = { Color.White, Color.Yellow,
                           Color.Wheat, Color.WhiteSmoke,
                           Color.SlateGray };
```

4. Add a constructor to the `StarField` class:

```
public StarField(
    int screenWidth,
    int screenHeight,
    int starCount,
    Vector2 starVelocity,
    Texture2D texture,
    Rectangle frameRectangle)
{
    this.screenWidth = screenWidth;
    this.screenHeight = screenHeight;
    for (int x = 0; x < starCount; x++)
    {
        stars.Add(new Sprite(
            new Vector2(rand.Next(0, screenWidth),
                        rand.Next(0, screenHeight)),
            texture,
            frameRectangle,
            starVelocity));
        Color starColor = colors[rand.Next(0, colors.Count())];
        starColor *= (float)(rand.Next(30, 80) / 100f);
        stars[stars.Count() - 1].TintColor = starColor;
    }
}
```

What just happened?

Each star in the star field will be stored in the `stars` list, making it simple to update and draw them using a `foreach` loop. When the stars are created, a color will be selected randomly from the `colors` array.

The class constructor assigns the `screenWidth` and `screenHeight` values to their local counterparts and then begins creating star sprites. Each sprite is assigned a random location and passed the `texture`, `frameRectangle`, and `starVelocity` parameters. Next, `starColor` is assigned a random color from the `colors` array. The color is then multiplied by a random value between `0.30f` and `0.79f`, making the star semi-transparent. This color is then assigned to the `TintColor` property of the star that was just added to the `stars` list.

Time for action – updating and drawing the StarField

1. Add the `Update()` and `Draw()` methods to the `StarField` class:

```
public void Update(GameTime gameTime)
{
    foreach (Sprite star in stars)
    {
        star.Update(gameTime);
        if (star.Location.Y > screenHeight)
        {
            star.Location = new Vector2(
                rand.Next(0, screenWidth), 0);
        }
    }
}

public void Draw(SpriteBatch spriteBatch)
{
    foreach (Sprite star in stars)
    {
        star.Draw(spriteBatch);
    }
}
```

What just happened?

When the star field needs to be updated, a `foreach` loop processes each item in the `stars` list, running the sprite's `Update()` method. The method then checks the star's `Location` property's `Y` component to determine if the star has moved off the bottom of the screen. If it has, the star is assigned a new `Location` with a random `X` component and a `Y` component of zero, placing the star at a random location along the top of the screen.

The `StarField.Draw()` method simply passes along the `spriteBatch` object to each of the individual stars in the `stars` list and instructs them to draw themselves.

Time for action – viewing the StarField in action

1. Add the following declaration to the `Game1` class:
`StarField starField;`
2. In the declarations area of the `Game1` class, temporarily modify the declaration for `gameState` from `GameStates.TitleScreen` to `GameStates.Playing`:
`GameStates gameState = GameStates.Playing;`

3. Update the `LoadContent()` method of the `Game1` class to initialize the `starField` object. Be sure to place this code after the `spriteSheet` texture is loaded.


```
starField = new StarField(
    this.Window.ClientBounds.Width,
    this.Window.ClientBounds.Height,
    200,
    new Vector2(0, 30f),
    spriteSheet,
    new Rectangle(0, 450, 2, 2));
```
4. In the `Update()` method, add the following line to the `GameStates.Playing` section of the switch statement you created earlier:


```
starField.Update(gameTime);
```
5. In the `Draw()` method, change the background color from `Color.CornflowerBlue` to `Color.Black`.
6. Still in the `Draw()` method, add the following line to the `if` block containing `GameStates.Playing`:


```
starField.Draw(spriteBatch);
```
7. Run the game by hitting `F5` and observe the star field. Exit the game by pressing `Alt + F4` on the keyboard.

What just happened?

Using our `StarField` class is simply a matter of creating an instance of the class and then calling its `Update()` and `Draw()` methods during the game's corresponding methods. When the `starField` object is constructed, we pass in the size of the screen, and specify that 200 stars will be created. Each star is assigned a velocity of `(0, 30f)`, meaning that in one second the star will move 30 pixels downward on the screen. With a screen height of 600 pixels, each star will take 20 seconds to travel from the top of the screen to the bottom.

The `spriteSheet` texture is specified, and the rectangle specifies an area on the textures that contains white pixels as we indicated above. The resulting sprite contains a single frame that is 2 by 2 pixels, located at `(0, 450)` on the sprite sheet.

Animated sprites – asteroids

The asteroids we create for Asteroid Belt Assault are included on the `SpriteSheet.png` texture as the top row of 50 by 50 pixel images. There are 20 individual animation frames of the asteroid spinning through space:



Much like we did with the `StarField` class, we will create a class that will automatically manage the game's asteroids for us, including their initial positioning, handling what happens when they collide with each other, and repositioning them when they have moved off the screen.

Time for action – building the `AsteroidManager` class

1. Add a new class called "AsteroidManager" to the Asteroid Belt Assault project.
2. Add the following using directives to the top of the AsteroidManager class:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Add the following declarations to the AsteroidManager class:

```
private int screenWidth = 800;  
private int screenHeight = 600;  
private int screenPadding = 10;  
  
private Rectangle initialFrame;  
private int asteroidFrames;  
private Texture2D texture;  
  
public List<Sprite> Asteroids = new List<Sprite>();  
private int minSpeed = 60;  
private int maxSpeed = 120;  
  
private Random rand = new Random();
```

4. Add a helper method that will be used in the AsteroidManager constructor:

```
public void AddAsteroid()  
{  
    Sprite newAsteroid = new Sprite(  
        new Vector2(-500, -500),  
        texture,
```

```

        initialFrame,
        Vector2.Zero);
    for (int x = 1; x < asteroidFrames; x++)
    {
        newAsteroid.AddFrame(new Rectangle(
            initialFrame.X + (initialFrame.Width * x),
            initialFrame.Y,
            initialFrame.Width,
            initialFrame.Height));
    }
    newAsteroid.Rotation =
        MathHelper.ToRadians((float)rand.Next(0, 360));
    newAsteroid.CollisionRadius = 15;
    Asteroids.Add(newAsteroid);
}

```

5. Add a second helper function to the AsteroidManager class:

```

public void Clear()
{
    Asteroids.Clear();
}

```

6. Add a constructor for the AsteroidManager class:

```

public AsteroidManager(
    int asteroidCount,
    Texture2D texture,
    Rectangle initialFrame,
    int asteroidFrames,
    int screenWidth,
    int screenHeight)
{
    this.texture = texture;
    this.initialFrame = initialFrame;
    this.asteroidFrames = asteroidFrames;
    this.screenWidth = screenWidth;
    this.screenHeight = screenHeight;
    for (int x = 0; x < asteroidCount; x++)
    {
        AddAsteroid();
    }
}

```


What just happened?

Asteroids will be stored in the `Asteroids` list object. The `AddAsteroid()` method provides a way to generate a new asteroid when necessary. As the game becomes increasingly more difficult, more asteroids can be added to make them harder to avoid.

When an asteroid is generated, its position is set to `(-500, -500)`, placing it well off the screen. Off screen asteroids will be repositioned during the `Update()` method, so instead of duplicating the code to position the asteroid we will simply start it off in a position that will force it to be moved during the first update cycle. The same is true for the asteroid's velocity. When the asteroid is positioned it will be assigned a random velocity, so initially we simply set the velocity to `(0, 0)`.

While the initial frame is added to the asteroid sprite during the sprite's construction, the other 19 frames need to be added in order for the animation to play properly. The `for` loop inside `AddAsteroid()` accomplishes this by pushing the X value of the rectangle representing the frame, the width of one frame to the right, for each added frame.



Sprite sheet organization

All of the animated sprites we will create for Asteroid Belt Assault are intentionally laid out on the `SpriteSheet.png` file so that each frame is located along a horizontal line in the file. This makes adding the subsequent animation frames as simple as moving the frame rectangle to the right by the width of the rectangle for each frame. The loop utilized in the `AddAsteroid()` method will be implemented for all of our animated game objects, requiring just an initial frame rectangle and a frame count to build all of the animation frames.

Finally, right before the asteroid is added to the `Asteroids` list, the sprite's rotation value is set randomly by generating a value in degrees and using `MathHelper.ToRadians()` to set the actual rotation value, and the asteroid's collision radius is assigned.

The constructor method copies all of the passed parameters to local variables, and then executes the `AddAsteroid()` method a number of times equal to the `asteroidCount` parameter.

Positioning the asteroids

Asteroids will be flying all over the screen while the game is running, and while we could generate new asteroids every time one is needed, and remove it when it leaves the screen, we are going to cheat a little. When an asteroid leaves the screen, we will simply generate a new (off screen) location for the asteroid and a new velocity and let it fly.

What if it goes straight off the screen again? That is ok too. It will be repositioned again as soon as it does.



Cheating

Do not shy away from cheating in your game design! Remember that the goal is to make the game *appear* to behave naturally, and not to write a realistic simulator. There are several instances of faking things to make them look good in Asteroid Belt Assault, including the random rotation set on each asteroid when it is generated to make it appear that there are several different asteroid graphics, and also the way we will create explosions in *Chapter 5*. Something that looks good and behaves properly because you do a little cheating is just as good (sometimes better!) than a complicated simulation.

Time for action – positioning the asteroids

1. Add methods to generate random locations and velocities to the AsteroidManager class:

```
private Vector2 randomLocation()
{
    Vector2 location = Vector2.Zero;
    bool locationOK = true;
    int tryCount = 0;

    do
    {
        locationOK = true;
        switch (rand.Next(0, 3))
        {
            case 0:
                location.X = -initialFrame.Width;
                location.Y = rand.Next(0, screenHeight);
                break;

            case 1:
                location.X = screenWidth;
                location.Y = rand.Next(0, screenHeight);
                break;

            case 2:
                location.X = rand.Next(0, screenWidth);
                location.Y = -initialFrame.Height;
                break;

        }
        foreach (Sprite asteroid in Asteroids)
```

```
        {
            if (asteroid.IsBoxColliding(
                new Rectangle(
                    (int)location.X,
                    (int)location.Y,
                    initialFrame.Width,
                    initialFrame.Height)))
            {
                locationOK = false;
            }
        }
        tryCount++;
        if ((tryCount > 5) && locationOK==false)
        {
            location = new Vector2(-500, -500);
            locationOK = true;
        }
    } while (locationOK == false);

    return location;
}

private Vector2 randomVelocity()
{
    Vector2 velocity = new Vector2(
        rand.Next(0, 101) - 50,
        rand.Next(0, 101) - 50);
    velocity.Normalize();
    velocity *= rand.Next(minSpeed, maxSpeed);
    return velocity;
}
```

What just happened?

When a random location is generated, we want to make sure it is not already colliding with an existing asteroid. In order to perform this check, a `do...while` loop surrounds the `switch` statement which randomly determines which side of the screen to use for the location. Here, we have chosen the left side of the screen (`case 0`), the right side of the screen (`case 1`) or the top of the screen (`case 2`). We have chosen not to have asteroids appear directly from the bottom of the screen, in order to give the player a little warning that they are coming, since the player ship will be near that side of the screen.

Each existing asteroid is checked to see if it would collide (using the Sprite class' `IsBoxColliding()` method) with a rectangle equal to the width and height of the asteroid frame placed at `location`. If it does collide with any asteroid, the `locationOK` Boolean is set to false, causing the loop to repeat. If the check has failed more than 5 times (determined by comparing against `tryCount`), the location is set to a distant off-screen location and the check is automatically passed in order to prevent the game getting stuck when trying to place too many asteroids in the limited off-screen area available. This is simply a safety measure. If we were to generate more than about 30 asteroids at the same time, it may become impossible for the code to place them all so that none of them are too close together. This would result in the game trying forever and locking up.

In order to generate a velocity, the `randomVelocity()` method creates a vector with both the X and Y components set randomly to a number between -50 and 50. That vector is then normalized, resulting in a vector that is exactly 1 unit long, but pointing in a random direction. The vector is multiplied by a random scaling factor, resulting in the final velocity which determines how far the asteroid moves in a single second of game time.

Time for action – checking the asteroid's position

1. Add the `isOnScreen()` helper method to the `AsteroidManager` class:

```
private bool isOnScreen(Sprite asteroid)
{
    if (asteroid.Destination.Intersects(
        new Rectangle(
            -screenPadding,
            -screenPadding,
            screenWidth + screenPadding,
            screenHeight + screenPadding)
        )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

What just happened?

The `isOnScreen()` method checks the passed asteroid to determine if its destination rectangle (where the sprite would be drawn to the screen) intersects a rectangle generated by expanding the screen by `screenPadding` pixels in all directions. This padding allows the asteroid to move several pixels off the screen before the game determines that it has actually left the playfield.

Time for action – updating and drawing asteroids

1. Add the `Update()` and `Draw()` methods to the `AsteroidManager` class:

```
public void Update(GameTime gameTime)
{
    foreach (Sprite asteroid in Asteroids)
    {
        asteroid.Update(gameTime);
        if (!isOnScreen(asteroid))
        {
            asteroid.Location = randomLocation();
            asteroid.Velocity = randomVelocity();
        }
    }
}

public void Draw(SpriteBatch spriteBatch)
{
    foreach (Sprite asteroid in Asteroids)
    {
        asteroid.Draw(spriteBatch);
    }
}
```

2. Add a declaration for the `AsteroidManager` in the declarations section of the `Game1` class:

```
AsteroidManager asteroidManager;
```

3. In the `Game1` class' `LoadContent()` method, initialize `asteroidManager` after the initialization of the `starField` object:

```
asteroidManager = new AsteroidManager(
    10,
    spriteSheet,
    new Rectangle(0, 0, 50, 50),
    20,
    this.Window.ClientBounds.Width,
    this.Window.ClientBounds.Height);
```

4. Add the following line to Game1's `Update()` method, immediately after the `starField.Update(gameTime)` line:
`asteroidManager.Update(gameTime);`
5. Add the following line to the `Draw()` method of Game1, right after the `starField.Draw(spriteBatch)` line:
`asteroidManager.Draw(spriteBatch);`
6. Execute the game to see the asteroids flying above the star field.

What just happened?

The `Update()` method passes the update along to each asteroid, and then checks to see if the asteroid is still on the screen. If the asteroid has moved off the screen, it is assigned a new random location and velocity.

As with the `StarField` class, the `AsteroidManager` class simply passes the draw call along to each of its component sprites when drawing.

Colliding asteroids

Asteroids fly nicely around the screen, but they do not interact with each other. When two asteroids collide, they simply pass right through each other and continue on their way. We will implement a simple elastic collision system, allowing the asteroids to bounce off of each other in a fairly realistic manner.

When two objects meet in an elastic collision, they rebound off of each other and preserve the total kinetic energy contained in the two asteroids after the impact. Energy may be transferred from one asteroid to the other, but the net energy will remain the same. While not perfectly realistic, this type of collision will suit our needs nicely because any individual asteroid is only visible to the player for a short time. When an asteroid leaves the screen and is repositioned, it is considered a "new" asteroid.

We will be implementing a special case elastic collision for our `AsteroidManager` class, in which all of the asteroids have equal mass, which simplifies the method involved in determining the velocities of the two asteroids after the collision.

Time for action – bouncing asteroids – part 1

1. Add the `BounceAsteroids()` method to the `AsteroidManager` class:

```
private void BounceAsteroids(Sprite asteroid1, Sprite asteroid2)
{
    {
        Vector2 cOfMass = (asteroid1.Velocity +
                           asteroid2.Velocity) / 2;

        Vector2 normal1 = asteroid2.Center - asteroid1.Center;
        normal1.Normalize();
        Vector2 normal2 = asteroid1.Center - asteroid2.Center;
        normal2.Normalize();

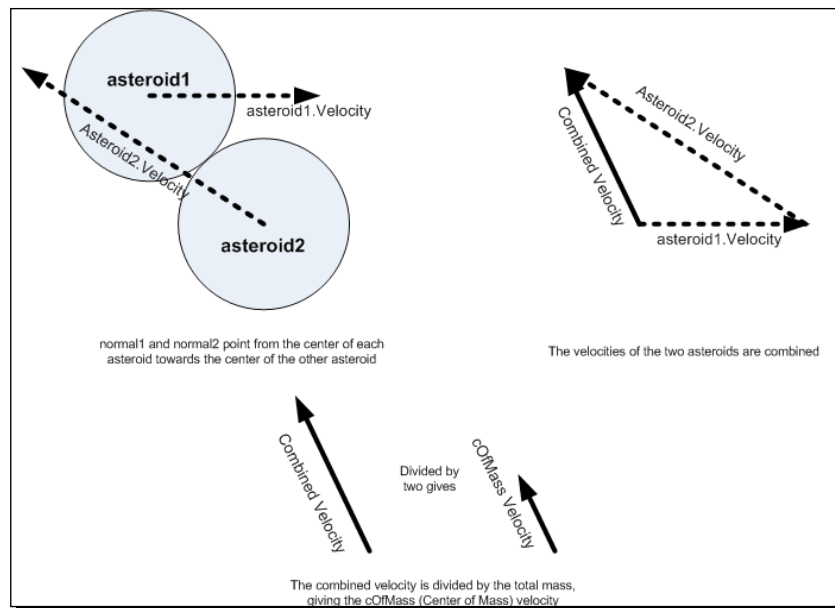
        asteroid1.Velocity -= cOfMass;
        asteroid1.Velocity =
            Vector2.Reflect(asteroid1.Velocity, normal1);
        asteroid1.Velocity += cOfMass;

        asteroid2.Velocity -= cOfMass;
        asteroid2.Velocity =
            Vector2.Reflect(asteroid2.Velocity, normal2);

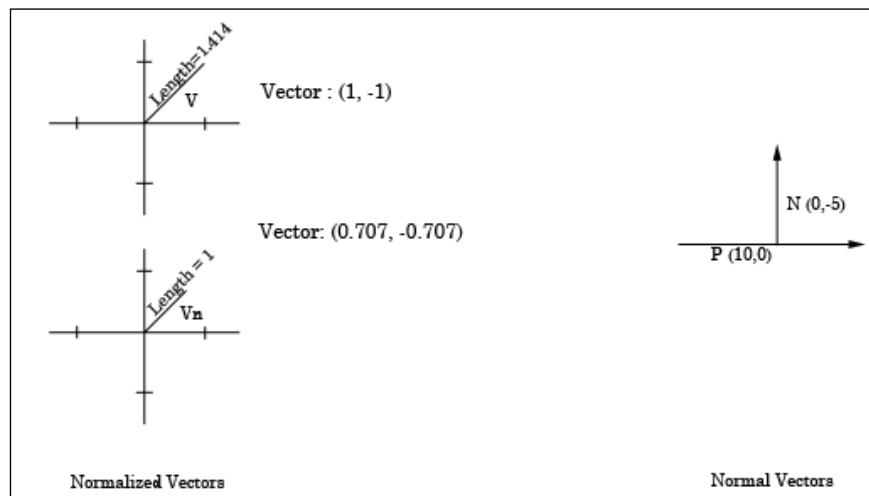
        asteroid2.Velocity += cOfMass;
    }
}
```

What just happened?

We begin by calculating the velocity of the center of mass for the collision, represented by the `cOfMass` vector variable. This value is determined by adding the velocities of the two colliding objects and then dividing by the total mass of the objects. In our special case, we are assuming both asteroids to have a mass of one, resulting in the vector sum being divided by two. If the two colliding objects hit each other and stuck together, this `cOfMass` velocity is the velocity that the resulting composite object would have:



The two vectors, `normal1` and `normal2` are generated by subtracting the locations of the centers of each asteroid from each other. This can be confusing because both vectors are "normalized normals". In vector terminology, a "normalized" vector (produced in XNA by the `Normalize()` method) is a vector whose length is exactly one unit. When a vector is normalized, it points in the same direction it initially pointed, but its length is increased or reduced to make it one unit long. In the following diagram, **V** is a vector **(1, -1)**. The length of the vector is equal to **1.414** units. Vector **Vn** is a normalized vector that points in the same direction, but is one unit long:



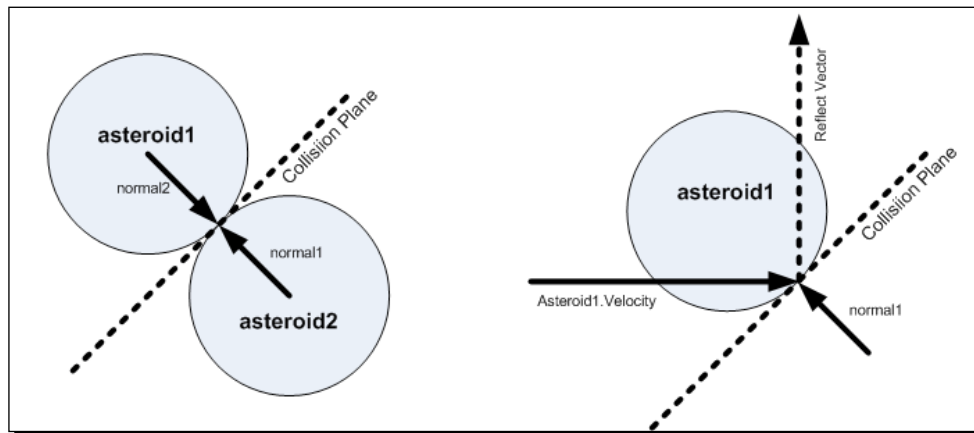


Vector normalization

XNA vectors support a `Length` property, returning the length of the vector as a float. Since all vectors can be thought of as the hypotenuse of a right-angled triangle, the length can be obtained by the Pythagorean Theorem $a^2+b^2=c^2$, with c being the length of the vector, while a and b represent the X and Y components. When a vector is normalized, it is simply divided by its length, scaling the X and Y components appropriately.

A "normal" vector is a vector that is perpendicular to another vector. In the diagram above, vector **N** is a normal vector to vector **P**. They are at right angles to each other. When used in calculations such as collision detection, it is often convenient for the normal vectors to also be normalized because multiplications involving the length of the vector end up multiplying by one, leaving the value unchanged.

In our case, **normal1** and **normal2** represent normalized (length of 1) vectors that are normal (perpendicular) to the plane along which the two asteroids are colliding, as in the following diagram:



In order to calculate the final velocity vector of an individual asteroid after the collision, the `cOfMass` vector is subtracted from the asteroid's velocity. The `Vector2` class' `Reflect()` method is then used to find the reflection vector between the asteroid's velocity (without the center of mass velocity) and the vector that is normal to the collision plane. Given a vector that is normal to the surface a vector is colliding with, `Reflect()` returns the result of that vector "bouncing off" the surface, as illustrated in the previous diagram.

This bit of vector magic produces a vector that points along the direction that the asteroid would move in if it had hit a solid, immobile plane. Since it is not impacting an immobile plane, but rather another moving object, the `cOfMass` vector is added back to the asteroid's velocity to represent the influence that the relative velocity of the other object has on the final direction and speed of the asteroid after the collision.

Time for action – bouncing asteroids – part 2

1. Replace the current `Update()` method of the `AsteroidManager` class:

```
public void Update(GameTime gameTime)
{
    foreach (Sprite asteroid in Asteroids)
    {
        asteroid.Update(gameTime);
        if (!isOnScreen(asteroid))
        {
            asteroid.Location=randomLocation();
            asteroid.Velocity=randomVelocity();
        }
    }

    for (int x = 0; x < Asteroids.Count; x++)
    {
        for (int y = x + 1; y < Asteroids.Count; y++)
        {
            if (Asteroids[x].IsCircleColliding(
                Asteroids[y].Center,
                Asteroids[y].CollisionRadius))
            {
                BounceAsteroids(Asteroids[x], Asteroids[y]);
            }
        }
    }
}
```

2. Execute the game and watch the asteroids bounce off of each other.
3. Close the game window.

What just happened?

Each asteroid is updated just as before. Afterwards, however, two nested loops process each asteroid in the list, checking for collisions with all of the other asteroids. If two asteroids collide, they are both passed to the `BounceAsteroids()` method. The second loop is written to minimize the collision comparisons by only checking asteroids with a higher index number than the current asteroid. When the first asteroid is checked, it is compared with all of the remaining asteroids, so when the second asteroid is processed, there is no need to check against the first asteroid again, as that has already been done and responded to.

Player and enemy shots

Before we add player or enemy ships, let's turn our attention to the shots that they will be firing. We will build the ShotManager class first, because both the player and enemy ships will need to use it to fire projectiles at each other. Instead of constructing the ship classes and then revisiting them, we can build the shots first and then build a way to utilize the manager in the ship classes.

Time for action – adding the ShotManager class

1. Add a new class called ShotManager to the Asteroid Belt Assault project.

2. Add the using directives to the top of the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the ShotManager class:

```
public List<Sprite> Shots = new List<Sprite>();
private Rectangle screenBounds;

private static Texture2D Texture;
private static Rectangle InitialFrame;
private static int FrameCount;
private float shotSpeed;
private static int CollisionRadius;
```

4. Add a constructor to the the ShotManager class:

```
public ShotManager(
    Texture2D texture,
    Rectangle initialFrame,
    int frameCount,
    int collisionRadius,
    float shotSpeed,
    Rectangle screenBounds)
{
    Texture = texture;
    InitialFrame = initialFrame;
    FrameCount = frameCount;
    CollisionRadius = collisionRadius;
    this.shotSpeed = shotSpeed;
    this.screenBounds = screenBounds;
}
```

What just happened?

ShotManager maintains a list of fired shots. An enemy fired shot will not hurt enemies, and a player fired shot will not hurt the player if they somehow manage to move into one. We will implement these properties when we build the CollisionManager class, which will check for collisions between fired shots and other game objects by creating two different instances of the ShotManager class.

When a shot leaves the `screenBounds` area, it will be removed from the list as it has left play.

Since shots are actually Sprites, they will use the standard sprite velocity system, represented in pixels per second. When a shot is created, a vector indicating the direction the shot is travelling in, will be multiplied by the appropriate shot speed value to determine its actual velocity.

The four static members, `Texture`, `InitialFrame`, `FrameCount`, and `CollisionRadius` are stored by the ShotManager class in order to assist it in creating new shots when necessary. As you may recognize, these are the parameters (minus the location and velocity) for creating a new Sprite instance. The ShotManager class constructor directly copies the passed values to the member variables of the class.

Time for action – firing shots

1. Add the `FireShot()` method to the ShotManager class:

```
public void FireShot(
    Vector2 location,
    Vector2 velocity,
    bool playerFired)
{
    Sprite thisShot = new Sprite(
        location,
        Texture,
        InitialFrame,
        velocity);

    thisShot.Velocity *= shotSpeed;
    for (int x = 1; x < FrameCount; x++)
    {
        thisShot.AddFrame(new Rectangle(
            InitialFrame.X + (InitialFrame.Width * x),
            InitialFrame.Y,
            InitialFrame.Width,
            InitialFrame.Height));
    }
    thisShot.CollisionRadius = CollisionRadius;
    Shots.Add(thisShot);
}
```

What just happened?

When a shot is fired, a new Sprite (called `thisShot`) is built using the parameters stored when the `ShotManager` class was constructed. The location and velocity are set according to the values passed to the `FireShot()` method. The velocity passed in will be normalized by the code that fires the shot, and is multiplied here by the `shotSpeed` to determine the shot's actual velocity. The shot is then added to the `Shots` list.

Time for action – updating and drawing shots

1. Add the `Update()` method to the `ShotManager` class:

```
public void Update(GameTime gameTime)
{
    for (int x = Shots.Count - 1; x >= 0; x--)
    {
        Shots[x].Update(gameTime);
        if (!screenBounds.Intersects(Shots[x].Destination))
        {
            Shots.RemoveAt(x);
        }
    }
}
```

2. Add the `Draw()` method to the `ShotManager` class:

```
public void Draw(SpriteBatch spriteBatch)
{
    foreach (Sprite shot in Shots)
    {
        shot.Draw(spriteBatch);
    }
}
```

What just happened?

Since we may be removing shots from the list, we cannot use a `foreach` loop to process the list during the `Update()` method. Looping backwards through the list allows us to process all of the shots, removing ones that have expired, without the need to track separate removal lists or restart the iteration after each removal.

After an individual shot has been updated, the `screenBounds` rectangle is checked to see if the shot is still visible on the screen. If it is not, the shot is removed from the list.

The `Draw()` method for the `ShotManager` is similar to the method used in the `AsteroidManager` class. It simply iterates through the list and instructs the sprites to draw themselves.

We will add instances of the `ShotManager` class to both our `PlayerManager` class and the `EnemyManager` class we will be building shortly.

Adding the player

The player's star fighter is a small, three frame animation that the player will control with either the keyboard or the gamepad:



As with both asteroids and shots, we will create a manager class for the player. The `PlayerManager` will be responsible for handling player input, firing shots, and limiting the player to a pre-defined area on the screen.

Time for action – creating the `PlayerManager` class

1. Add a new class called `PlayerManager` to the Asteroid Belt Assault project.

2. Add the following `using` directives to the top of the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the `PlayerManager` class:

```
public Sprite playerSprite;
private float playerSpeed = 160.0f;
private Rectangle playerAreaLimit;

public long PlayerScore = 0;
public int LivesRemaining = 3;
public bool Destroyed = false;

private Vector2 gunOffset = new Vector2(25, 10);
private float shotTimer = 0.0f;
private float minShotTimer = 0.2f;
private int playerRadius = 15;
public ShotManager PlayerShotManager;
```

4. Add a constructor to the PlayerManager class:

```
public PlayerManager(
    Texture2D texture,
    Rectangle initialFrame,
    int frameCount,
    Rectangle screenBounds)
{
    playerSprite = new Sprite(
        new Vector2(500, 500),
        texture,
        initialFrame,
        Vector2.Zero);

    PlayerShotManager = new ShotManager(
        texture,
        new Rectangle(0, 300, 5, 5),
        4,
        2,
        250f,
        screenBounds);

    playerAreaLimit =
        new Rectangle(
            0,
            screenBounds.Height / 2,
            screenBounds.Width,
            screenBounds.Height / 2);

    for (int x = 1; x < frameCount; x++)
    {
        playerSprite.AddFrame(
            new Rectangle(
                initialFrame.X + (initialFrame.Width * x),
                initialFrame.Y,
                initialFrame.Width,
                initialFrame.Height));
    }
    playerSprite.CollisionRadius = playerRadius;
}
```

What just happened?

As with all of our other objects, the player's ship is represented as a Sprite. When the player moves, their speed is limited to 160 pixels per second. This value will be used to scale the velocity vector associated with the player sprite while moving. In addition to limiting the speed at which the player can move, the player is limited to the area described by the `playerAreaLimit` rectangle. If the player's velocity carries their star fighter outside of this rectangle, their position will be adjusted to move the ship back into this area.

The `playerScore` and `livesRemaning` variables will be used to track the player's progress in the game. They will be reset when a new game is started and manipulated by other code such as the `CollisionManager`. When a player collides with a deadly object, the `Destroyed` member will be set to true, allowing the game management system to take appropriate actions such as decreasing the number of lives remaining, respawning the player, or ending the game.

The next four member variables allow the player to fire their star fighter's cannon. The `gunOffset` vector points to the location on the ship's sprite image where newly fired projectiles will be created. The `shotTimer` and `minShotTimer` floats will be used in our now familiar timing mechanism to prevent a constant stream of player-fired bullets from issuing from the player's cannon. The `playerRadius` value represents the size of the player's ship. This value will be copied to the sprite's `CollisionRadius` member in order to be checked against other objects during collision detection.

The `PlayerManager` constructor builds the `playerSprite` object from the passed variables and also creates an instance of the `ShotManager` class, allowing the `PlayerManager` class to handle any shots fired by the player.

Handling user input

Asteroid Belt Assault will support both keyboard and gamepad based control. If you have a wired Xbox 360 gamepad, it will plug into a USB port on your PC, and will be recognized by XNA for controlling your games.

We saw an example of mouse-based user input in the `SquareChase` and `Flood Control` games, using the `MouseState` object. Similar objects, `KeyboardState` and `GamePadState` exist for the keyboard and gamepad, respectively, and they are used in the same manner as the `MouseState` object: The state is captured via the `GetState()` method and members of the returned object can be checked to determine the state of keyboard keys or gamepad thumbsticks and buttons.

Time for action – handling user input

1. Add the `FireShot()` helper method to the `PlayerManager` class:

```
private void FireShot()
{
    if (shotTimer >= minShotTimer)
    {
        PlayerShotManager.FireShot(
            playerSprite.Location + gunOffset,
            new Vector2(0, -1),
            true);
        shotTimer = 0.0f;
    }
}
```


2. Add the `HandleKeyboardInput()` helper method to the `PlayerManager` class:

```
private void HandleKeyboardInput (KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Up))
    {
        playerSprite.Velocity += new Vector2(0, -1);
    }

    if (keyState.IsKeyDown(Keys.Down))
    {
        playerSprite.Velocity += new Vector2(0, 1);
    }

    if (keyState.IsKeyDown(Keys.Left))
    {
        playerSprite.Velocity += new Vector2(-1, 0);
    }

    if (keyState.IsKeyDown(Keys.Right))
    {
        playerSprite.Velocity += new Vector2(1, 0);
    }

    if (keyState.IsKeyDown(Keys.Space))
    {
        FireShot();
    }
}
```

3. Add the `HandleGamepadInput()` helper method to the `PlayerManager` class:

```
private void HandleGamepadInput (GamePadState gamePadState)
{
    playerSprite.Velocity +=
        new Vector2(
            gamePadState.ThumbSticks.Left.X,
            -gamePadState.ThumbSticks.Left.Y);

    if (gamePadState.Buttons.A == ButtonState.Pressed)
    {
        FireShot();
    }
}
```

What just happened?

The `FireShot()` helper method will be called from both the `HandleKeyboardInput()` and `HandleGamepadInput()` methods, and implements a portion of the standard timing mechanism we have been using throughout this book. The remainder of the mechanism (incrementing the `shotTimer` value) will take place in the `Update()` method we will be adding shortly. When a shot is fired, its velocity is passed as a normalized vector pointing straight up (the only direction the player can fire). Since we have this limitation in place, we do not need to actually normalize the vector as we can create it as a vector with a length of 1 by specifying `(0, -1)` as the vector components.

The `IsKeyDown()` method of the `KeyboardState` class returns true if the indicated key is down when the state is captured. As far as XNA is concerned, there is no keyboard buffer or individual key presses. Each key is either up or down when the state is read. The directional keys individually add to the player's velocity vector in directions corresponding to their key presses. For example, if the `Keys.Up` key is pressed, a vector of `(0, -1)` is added to the player's velocity. This means it is possible for the player to press both the Up and Down keys at the same time, though the two vectors will simply cancel each other out.

If the *Space bar* on the keyboard is pressed, the `FireShot()` helper method is called, indicating that the player wishes to fire a projectile if enough time has elapsed since the previous projectile was fired.

`HandleGamepadInput()` is somewhat simpler than the keyboard input handler because the left thumbstick, that we will use to control the player's ship, returns a vector that represents the direction the player has the stick pointed in. We cannot simply apply the thumbstick vector to the ship's velocity because the sign of the Y component of the vector on the gamepad is the reverse of its meaning on the screen. In other words, pressing up on the thumbstick returns a positive Y value, while moving up on the screen is a negative Y value. By building a new vector out of the components of the thumbstick vector and negating the Y value, we can add this new vector to the velocity vector of the ship. The A button on the gamepad triggers the call to the `FireShot()` method.

Time for action – updating and drawing the player's ship

1. Add the `imposeMovementLimits()` helper method to the `PlayerManager` class:

```
private void imposeMovementLimits()
{
    Vector2 location = playerSprite.Location;

    if (location.X < playerAreaLimit.X)
        location.X = playerAreaLimit.X;

    if (location.X >
        (playerAreaLimit.Right - playerSprite.Source.Width))
```

```
        location.X =
            (playerAreaLimit.Right - playerSprite.Source.Width);

        if (location.Y < playerAreaLimit.Y)
            location.Y = playerAreaLimit.Y;

        if (location.Y >
            (playerAreaLimit.Bottom - playerSprite.Source.Height))
            location.Y =
                (playerAreaLimit.Bottom - playerSprite.Source.Height);

        playerSprite.Location = location;
    }
```

2. Add the Update() method to the PlayerManager class:

```
public void Update(GameTime gameTime)
{
    PlayerShotManager.Update(gameTime);

    if (!Destroyed)
    {
        playerSprite.Velocity = Vector2.Zero;

        shotTimer += (float)gameTime.ElapsedGameTime.TotalSeconds;

        HandleKeyboardInput(Keyboard.GetState());
        HandleGamepadInput(GamePad.GetState(PlayerIndex.One));

        playerSprite.Velocity.Normalize();
        playerSprite.Velocity *= playerSpeed;

        playerSprite.Update(gameTime);
        imposeMovementLimits();
    }
}
```

3. Add the Draw() method to the PlayerManager class:

```
public void Draw(SpriteBatch spriteBatch)
{
    PlayerShotManager.Draw(spriteBatch);

    if (!Destroyed)
    {
        playerSprite.Draw(spriteBatch);
    }
}
```

4. Add a declaration to the `Game1` class for the `PlayerManager`:

```
PlayerManager playerManager;
```
5. In the `LoadContent()` method of the `Game1` class, set up the `PlayerManager` after the `AsteroidManager` is initialized:

```
playerManager = new PlayerManager(
    spriteSheet,
    new Rectangle(0, 150, 50, 50),
    3,
    new Rectangle(
        0,
        0,
        this.Window.ClientBounds.Width,
        this.Window.ClientBounds.Height));
```
6. In the `Update()` method of the `Game1` class, add an update line for the `PlayerManager` right after the `AsteroidManager` is updated:

```
playerManager.Update(gameTime);
```
7. In the `Draw()` method of the `Game1` class, add a draw line for the `PlayerManager` right after the `AsteroidManager` is drawn:

```
playerManager.Draw(spriteBatch);
```
8. Execute the game and fly your star fighter around in the asteroid field. Fire off a few shots with your cannon!

What just happened?

The `imposeMovementLimits()` method begins by making a copy of the `playerSprite`'s `Location` property. Since `Location` is a property and not a public member, we cannot modify the components of the vector (X and Y) individually. Creating a temporary copy allows us to independently modify these values and then save the whole vector back to the property.

The X and Y values of the vector are checked against the edges of the `playerAreaLimit` rectangle. On the right and bottom edges, the width and height of the player sprite is subtracted from the bounding rectangle edges to ensure that the player sprite stays entirely on the screen. If any of the edges are out of alignment, the components of the location vector are adjusted to keep the ship within the play area.

The location is then saved back into the `playerSprite.Location` property.

Updating the player manager begins by updating its related `ShotManager`. Then, if the `Destroyed` variable is false, the remainder of the `Update()` method is allowed to proceed. This check prevents the player from continuing to fire shots after being killed. The update begins by setting the player's velocity to zero and incrementing the timer for firing shots. The `HandleKeyboardInput()` and `HandleGamepadInput()` methods are then called, passing each the current state of the appropriate input device.

We need to resolve two potential issues with player movement at this point. If the player were simply moving to the right, the player's velocity vector would be equal to (1, 0), with a length of 1 unit. If, however, the player is holding down both the right and up keys, the resulting vector would be (1, 1). As we saw when discussing normalized vectors while bouncing asteroids off of each other, this vector has a length of 1.414 units, meaning that the player can move faster by moving diagonally than they can by moving in straight lines.

A related side issue is that if the player has an Xbox gamepad connected to their PC, they could hold down both the thumbstick and the arrow keys to move at twice the normal speed in their chosen direction.

To compensate for both of these potential issues, the `Update()` method normalizes the player's velocity vector and then multiplies it by the `playerSpeed` variable. This results in a vector that is always the same length (unless, of course, the player is not moving, in which case the vector value of (0, 0) has no length at all).

After all of the input and velocity changes have been accounted for, the `Update()` method of the `playerSprite` object is called to allow the velocity to be added to the player's location and to advance the animation frame. Finally, the `imposeMovementLimits()` method is called to make sure the ship stays within the play area.

When the `PlayerManager` is drawn, the same series of events takes place as the `Update()` method. First, the associated `ShotManager` is drawn, followed by a check to see if the player has been destroyed. If they are still alive, the player ship gets drawn to the screen. It is important to note that the `ShotManager` gets drawn before the player, because any shot the player fires will begin somewhat overlapping the player's sprite. By drawing the shots before the player ship, the shot will appear to come from inside the ship, instead of appearing on top of the ship when fired.

Enemy ships

Now that the player can fly and shoot their weapons, it is time to give them something to shoot at!



Our enemy ships will follow a series of waypoints (specified as a list of vectors pointing to screen coordinates) across the screen. Because we need this new type of behaviour that all enemy ships will implement, we will create an `Enemy` class as well as an `EnemyManager` class.

Time for action – creating the `Enemy` class

1. Add a new class called `Enemy` to the Asteroid Belt Assault project.

2. Add the standard using directives to the class:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the `Enemy` class:

```
public Sprite EnemySprite;
public Vector2 gunOffset = new Vector2(25, 25);
private Queue<Vector2> waypoints = new Queue<Vector2>();
private Vector2 currentWaypoint = Vector2.Zero;
private float speed = 120f;
public bool Destroyed = false;
private int enemyRadius = 15;
private Vector2 previousLocation = Vector2.Zero;
```

4. Add a constructor to the `Enemy` class:

```
public Enemy(
    Texture2D texture,
    Vector2 location,
    Rectangle initialFrame,
    int frameCount)
{
    EnemySprite = new Sprite(
        location,
        texture,
        initialFrame,
        Vector2.Zero);

    for (int x = 1; x < frameCount; x++)
    {
        EnemySprite.AddFrame(
            new Rectangle(
                initialFrame.X = (initialFrame.Width * x),
                initialFrame.Y,
                initialFrame.Width,
```

```
        initialFrame.Height));  
    }  
    previousLocation = location;  
    currentWaypoint = location;  
    EnemySprite.CollisionRadius = enemyRadius;  
}
```

What just happened?

As always, our object contains a Sprite object that will provide its base display and update capabilities. Just like we did with the player ship, we have included a vector pointing to the location on the sprite where shots will begin when the enemy ship fires its cannon.

Enemy ships will be supplied with a list of waypoints that they will progress through, with the waypoint the ship is currently attempting to reach stored in the `currentWaypoint` variable. Movement towards the current waypoint will be scaled by `speed`.

When the Sprite representing the enemy is created, the `enemyRadius` value will be copied to the Sprite's `CollisionRadius` member for checking during collision detection.

The `Destroyed` Boolean will be set to true by the `CollisionManager` class we will create in *Chapter 5*. When `Destroyed` is true, the ship will not be drawn, updated, or eligible for collision with other objects.

In order to rotate the enemy ship automatically to point in the direction it is travelling, the `previousLocation` vector stores the ship's location during the previous frame. By calculating the vector from the previous location to the new location, we can determine the facing of the ship and rotate the image appropriately.

Most of the `Enemy` constructor simply initializes a new Sprite object with the passed parameters, adding frames to the sprite based on `frameCount`. Both the `previousLocation` and the `currentWaypoint` values are set to the sprite's creation position, and the collision radius is set at the end of the constructor.

Time for action – waypoint management

1. Add the `AddWaypoint()` method to the `Enemy` class:

```
public void AddWaypoint(Vector2 waypoint)  
{  
    waypoints.Enqueue(waypoint);  
}
```

2. Add the `WaypointReached()` method to the `Enemy` class:

```
public bool WaypointReached()
{
    if (Vector2.Distance(EnemySprite.Location, currentWaypoint) <
        (float)EnemySprite.Source.Width/2)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

3. Add the `IsActive()` method to the `Enemy` class:

```
public bool IsActive()
{
    if (Destroyed)
    {
        return false;
    }

    if (waypoints.Count > 0)
    {
        return true;
    }

    if (WaypointReached())
    {
        return false;
    }

    return true;
}
```

What just happened?

When a new waypoint is added to the enemy's route, it is enqueued to the `waypoints` queue. When `WaypointReached()` is called, the function checks to see if the distance between the sprite's current location and the current waypoint is less than half of the sprite width. If it is, we consider the sprite "close enough" to have reached the destination.

It may be tempting to simply check to see if the distance is equal to zero, but in reality that will never really happen. Your enemy ship is moving at a fixed rate per second, that is scaled in each frame by the time elapsed since the last frame. For this reason, the ship is moving in increments that may not be equal to full pixels. The ship will get close to the waypoint, but it is unlikely to ever exactly reach it. Instead, it would jump back and forth slightly past the waypoint in one direction, then reverse and jump slightly past it in the other direction.

The `IsActive()` method is a shortcut method to check several different conditions to determine if the ship should be updated and visible. If the ship has been marked as destroyed, obviously it is not active, hence `false` is returned.

If this ship still has waypoints to reach (and has not been destroyed, otherwise `false` would already have been returned), we can assume that the ship is still active, and will return `true`.

Finally, if there are no waypoints left in the `waypoints` queue, and the `currentWaypoint` has been reached, we return `false` because the ship has navigated all of its waypoints and is resting at its final destination. When we generate waves of enemies, the initial location and final waypoint will always be off screen, allowing the enemy ships to enter the playfield and exit gracefully. If they make it all the way to the exit without the player destroying them, they will be removed from play.

If none of the conditions above resulted in a return value for the function, it can be assumed that the ship is still travelling to a waypoint, and `true` is returned from the function.

Time for action – enemy update and draw

1. Add the `Update()` method to the `Enemy` class:

```
public void Update(GameTime gameTime)
{
    if (IsActive())
    {
        Vector2 heading = currentWaypoint - EnemySprite.Location;
        if (heading != Vector2.Zero)
        {
            heading.Normalize();
        }
        heading *= speed;
        EnemySprite.Velocity = heading;
        previousLocation = EnemySprite.Location;
        EnemySprite.Update(gameTime);
        EnemySprite.Rotation =
            (float)Math.Atan2(
                EnemySprite.Location.Y - previousLocation.Y,
```

```

        EnemySprite.Location.X - previousLocation.X);

    if (WaypointReached())
    {
        if (waypoints.Count > 0)
        {
            currentWaypoint = waypoints.Dequeue();
        }
    }
}

```

2. Add the `Draw()` method to the `Enemy` class:

```

public void Draw(SpriteBatch spriteBatch)
{
    if (IsActive())
    {
        EnemySprite.Draw(spriteBatch);
    }
}

```

What just happened?

Updating an `Enemy` begins by checking to see if the enemy is still active. If it is, a heading is calculated by subtracting the enemy's current location from the current waypoint. If the resulting vector is not equal to `Vector2.Zero`, the vector is normalized. We need to make this check because `Vector2.Zero` cannot be normalized, which will result in an error in our code. Recall that normalizing a vector divides the vector by its length, resulting in a vector with a length of one. `Vector2.Zero` has a length of zero, and dividing anything by zero produces an invalid result.

Once we have a heading, it is multiplied by the ship's speed, and stored as the sprite's velocity vector. The current location of the sprite is captured, and the sprite's `Update()` method is called that will move the sprite based on its velocity.

By using the location we stored before the `Update()` call, we can use the `Math.Atan2()` method to determine the angle between the current location and the previous location, resulting in an angle of rotation that can be applied to the `Rotation` property of the sprite in order to automatically rotate it to face the direction in which it is moving. Using this method, a sprite moving only to the right (in the positive X direction) would result in a rotation value of zero, which is why the enemy sprite on the sprite sheet is drawn facing to the right.

Finally, the `WaypointReached()` method is checked to see if the just-executed movement has brought the enemy to its current waypoint. If it has, and there are additional waypoints for the sprite to visit, the next waypoint is extracted from the queue.

The `Draw()` method should be very familiar by now, as it is the same implementation that all of our manager classes have been using. Rest assured, we will be seeing it again.

The EnemyManager class

The `EnemyManager` class is responsible for keeping track of all active enemy ships, spawning new ships and waves of ships, and determining when ships should fire at the player. Additionally, it will be responsible for removing ships that are destroyed or leave the play area by completing their waypoint movements.

Time for action – creating the `EnemyManager` class

1. Add a new class to the `AsteroidBeltAssault` project called `EnemyManager`.

2. Add the standard `using` directives to the top of the file:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the `EnemyManager` class:

```
private Texture2D texture;  
private Rectangle initialFrame;  
private int frameCount;  
  
public List<Enemy> Enemies = new List<Enemy>();  
  
public ShotManager EnemyShotManager;  
private PlayerManager playerManager;  
  
public int MinShipsPerWave = 5;  
public int MaxShipsPerWave = 8;  
private float nextWaveTimer = 0.0f;  
private float nextWaveMinTimer = 8.0f;  
private float shipSpawnTimer = 0.0f;  
private float shipSpawnWaitTime = 0.5f;  
  
private float shipShotChance = 0.2f;  
  
private List<List<Vector2>> pathWaypoints =  
    new List<List<Vector2>>();  
  
private Dictionary<int, int> waveSpawns = new Dictionary<int,  
    int>();  
  
public bool Active = true;  
  
private Random rand = new Random();
```

What just happened?

As with our other manager classes, the `EnemyManager` will cache information about the sprite used to create enemies in the `texture`, `initialFrame`, and `frameCount` members in order to supply them to the constructor for the `Sprite` class when an enemy needs to be generated. Enemies will be stored in the `Enemies` list object that will be updated and drawn in a manner similar to the asteroids and shots.

Because the `EnemyManager` will be responsible for determining when enemy ships should fire at the player, it needs a reference to the `PlayerManager` (to determine the player's current position) and contains an instance of `ShotManager` (in order to use the `FireShot()` method).

Waves of enemy ships will be spawned by the `EnemyManager` based on a timer. Each wave will consist of at least `MinShipsPerWave`, but no more than `MaxShipsPerWave`. These values are declared as public so that they can be modified at run time in order to increase the difficulty of the game as play continues.

Our standard timing mechanism is in place to determine when each wave should be spawned (`nextWaveTimer` and `nextWaveMinTimer`) and a second timer determines the spawn time between each ship in a wave. In this case, one half of a second will elapse between the spawning of each ship in a wave.

During any given frame, each active ship has a 0.2% chance of firing a shot at the player. This number may seem very low, but keep in mind that the check will be made 60 times per second for each active ship.

The `pathWaypoints` object is a list of lists of `Vector2` objects. This is not really as complicated as it may sound. Remember that a waypoint is represented by a vector pointing to a location on the screen. The path that any enemy ship travels is determined by a list of waypoints or a path across the screen that the ship will take.

Since we do not want every ship to follow exactly the same path across the screen, we will define several different paths. Each path has its own list of waypoints. In order to facilitate easy random generation and expansion of the path system, each of the paths is rolled up into a list of paths. When we request an item from the `pathWaypoints` list, the result is a list of vectors. We can then break that list apart for each of the individual waypoints.

When the time comes to actually spawn ships, we will check the `waveSpawns` dictionary. The keys in the dictionary (the first integer) will correspond to the path number, while the value (the second integer) determines the number of ships waiting to be spawned on that path.

The `Active` member will be checked during the class' update tasks to determine if new ships should be spawned. External code can set this Boolean value to false to prevent new ships (or waves of ships) from being spawned.

Managing waypoints

In order to create the `pathWaypoints`, we will use a helper function that generates individual path lists and adds them to the aggregated list object.

Time for action – setting up the `EnemyManager` class

1. Add the `setUpWaypoints()` helper function to the `EnemyManager` class:

```
private void setUpWaypoints()
{
    List<Vector2> path0 = new List<Vector2>();
    path0.Add(new Vector2(850, 300));
    path0.Add(new Vector2(-100, 300));
    pathWaypoints.Add(path0);
    waveSpawns[0] = 0;

    List<Vector2> path1 = new List<Vector2>();
    path1.Add(new Vector2(-50, 225));
    path1.Add(new Vector2(850, 225));
    pathWaypoints.Add(path1);
    waveSpawns[1] = 0;

    List<Vector2> path2 = new List<Vector2>();
    path2.Add(new Vector2(-100, 50));
    path2.Add(new Vector2(150, 50));
    path2.Add(new Vector2(200, 75));
    path2.Add(new Vector2(200, 125));
    path2.Add(new Vector2(150, 150));
    path2.Add(new Vector2(150, 175));
    path2.Add(new Vector2(200, 200));
    path2.Add(new Vector2(600, 200));
    path2.Add(new Vector2(850, 600));
    pathWaypoints.Add(path2);
    waveSpawns[2] = 0;

    List<Vector2> path3 = new List<Vector2>();
    path3.Add(new Vector2(600, -100));
    path3.Add(new Vector2(600, 250));
    path3.Add(new Vector2(580, 275));
    path3.Add(new Vector2(500, 250));
    path3.Add(new Vector2(500, 200));
    path3.Add(new Vector2(450, 175));
    path3.Add(new Vector2(400, 150));
    path3.Add(new Vector2(-100, 150));
    pathWaypoints.Add(path3);
    waveSpawns[3] = 0;
}
```

2. Add a constructor to the EnemyManager class:

```
public EnemyManager(
    Texture2D texture,
    Rectangle initialFrame,
    int frameCount,
    PlayerManager playerManager,
    Rectangle screenBounds)
{
    this.texture = texture;
    this.initialFrame = initialFrame;
    this.frameCount = frameCount;
    this.playerManager = playerManager;

    EnemyShotManager = new ShotManager(
        texture,
        new Rectangle(0, 300, 5, 5),
        4,
        2,
        150f,
        screenBounds);

    setUpWaypoints();
}
```

What just happened?

Each of the paths is created in the `setUpWaypoints()` method as individual lists of vectors. Each list is added to the `pathWaypoints` list, and the `waveSpawns` dictionary for that path is set to zero, indicating that there are no ships waiting to spawn on that path.

The `EnemyManager` constructor assigns the cached values and references, creates an instance of `ShotManager`, and calls the `setUpWaypoints()` method to generate the `pathWaypoints` list. New waypoint paths can be added to the game simply by creating them in the helper method above. As we will see below, they will automatically be included in the random generation of waves.

Time for action – spawning enemies

1. Add the `SpawnEnemy()` method to the `EnemyManager` class:

```
public void SpawnEnemy(int path)
{
    Enemy thisEnemy = new Enemy(
        texture,
```

```
        pathWaypoints[path][0],
        initialFrame,
        frameCount);
    for (int x = 0; x < pathWaypoints[path].Count(); x++)
    {
        thisEnemy.AddWaypoint(pathWaypoints[path][x]);
    }
    Enemies.Add(thisEnemy);
}
```

2. Add the SpawnWave() method to the EnemyManager class:

```
public void SpawnWave(int waveType)
{
    waveSpawns[waveType] +=
        rand.Next(MinShipsPerWave, MaxShipsPerWave + 1);
}
```

3. Add the updateWaveSpawns() method to the EnemyManager class:

```
private void updateWaveSpawns(GameTime gameTime)
{
    shipSpawnTimer += (float)gameTime.ElapsedGameTime.
        TotalSeconds;
    if (shipSpawnTimer > shipSpawnWaitTime)
    {
        for (int x = waveSpawns.Count - 1; x >= 0; x--)
        {
            if (waveSpawns[x] > 0)
            {
                waveSpawns[x]--;
                SpawnEnemy(x);
            }
        }
        shipSpawnTimer = 0f;
    }

    nextWaveTimer += (float)gameTime.ElapsedGameTime.TotalSeconds;
    if (nextWaveTimer > nextWaveMinTimer)
    {
        SpawnWave(rand.Next(0, pathWaypoints.Count));
        nextWaveTimer = 0f;
    }
}
```

What just happened?

Spawning a new enemy ship requires the path number that the ship will be spawned on. Given this information, a new `Enemy` object is created and given the cached information about the sprite, along with the first point in the waypoints list for the given path. Each subsequent waypoint is then added to the enemy's waypoint list, and the enemy is added to the `Enemies` list.

Spawning a new wave does not actually spawn any individual enemies, but instead sets the entry in the `waveSpawns` dictionary which corresponds to the waypoint path to a random number of enemies between `MinShipsPerWave` and `MaxShipsPerWave`.

The `updateWaveSpawns()` method maintains the timing loop to determine when new ships can be spawned. If enough time has elapsed, a new ship is spawned at the beginning of each waypoint path that currently has ships waiting to spawn (meaning that the `waveSpawns` value for that path is greater than zero).

The `nextWaveTimer` value is then checked to see if enough time has elapsed to spawn a new wave. If so, `SpawnWave()` is called, given a random wave number between 0 and the last path in `pathWaypoints`. The timer is then reset in preparation for the next wave.

Time for action – updating and drawing the `EnemyManager`

1. Add the `Update()` method to the `EnemyManager` class:

```
public void Update(GameTime gameTime)
{
    EnemyShotManager.Update(gameTime);

    for (int x = Enemies.Count - 1; x >= 0; x--)
    {
        Enemies[x].Update(gameTime);
        if (Enemies[x].IsActive() == false)
        {
            Enemies.RemoveAt(x);
        }
        else
        {
            if ((float)rand.Next(0, 1000) / 10 <= shipShotChance)
            {
                Vector2 fireLoc = Enemies[x].EnemySprite.Location;
                fireLoc += Enemies[x].gunOffset;

                Vector2 shotDirection =
                    playerManager.playerSprite.Center -
                    fireLoc;

                shotDirection.Normalize();
            }
        }
    }
}
```



```
                EnemyShotManager.FireShot (
                    fireLoc,
                    shotDirection,
                    false);
            }
        }
    }

    if (Active)
    {
        updateWaveSpawns (gameTime);
    }
}
```

- 2.** Add the Draw() method to the EnemyManager class:

```
public void Draw(SpriteBatch spriteBatch)
{
    EnemyShotManager.Draw(spriteBatch);

    foreach (Enemy enemy in Enemies)
    {
        enemy.Draw(spriteBatch);
    }
}
```

- 3.** Add a declaration for the EnemyManager in the Game1 class:

```
EnemyManager enemyManager;
```

- 4.** In the LoadContent() method of the Game1 class, initialize the EnemyManager after the PlayerManager has been initialized:

```
enemyManager = new EnemyManager(
    spriteSheet,
    new Rectangle(0, 200, 50, 50),
    6,
    playerManager,
    new Rectangle(
        0,
        0,
        this.Window.ClientBounds.Width,
        this.Window.ClientBounds.Height));
```

- 5.** In the Update() method of the Game1 class, add a line to update the EnemyManager after the update for the PlayerManager:

```
enemyManager.Update(gameTime);
```

- 6.** In the Draw() method of the Game1 class, add a line to draw the EnemyManager after the call to draw the PlayerManager:

```
enemyManager.Draw(spriteBatch);
```

7. Launch your game! You can now fly around and fire shots. Waves of enemy ships will spawn and follow their waypoints across the screen.

What just happened?

Once again, we cannot use a `foreach` loop to process the `Enemies` list because we will be modifying the list by removing inactive ships, so we work backwards through the list with a `for` loop. Any enemies that are marked as inactive will be removed from the list.

For enemies that are still active, a random number between 0 and 999 is generated, which is then divided by 10. The resulting number is a percentage that is compared to `shipShotChance` to determine if the enemy should fire at the player in this frame. If so, the ship's current location and gun offset are used to draw a vector towards the player's current location. This vector is then normalized and passed to the `FireShot()` method of the `ShotManager`.

Lastly, if the `Active` member has not been set to `false` (disabling spawning until it has been set back to `true`), the `updateWaveSpawns()` method is called to allow new ships and waves to be spawned as necessary.

Summary

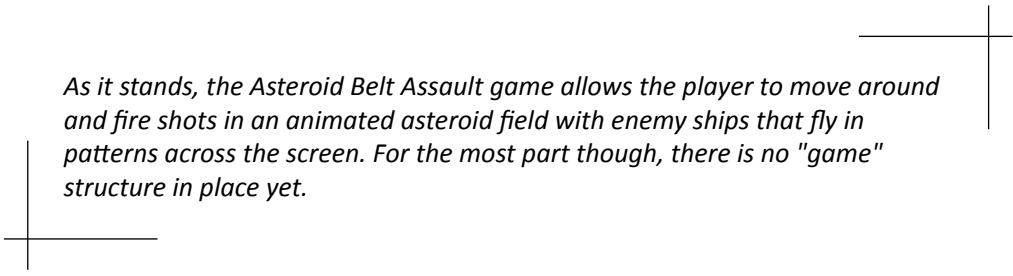
Chapter 4 has us well on our way to implementing a playable version of Asteroid Belt Assault. We have covered:

- ◆ Building a generic `Sprite` class to represent objects within the game world
- ◆ Creating a scrolling star field using a large number of sprites that update their own positions and a class to encapsulate the management of the star field as a whole
- ◆ Implementing an asteroid field, with asteroids that rebound off each other using elastic collisions to simulate realistic changes in direction and speed
- ◆ Creating manager classes for shots, the player's star fighter, and enemy ships
- ◆ Designing an `Enemy` class that can follow a set of waypoints as the ships traverse the screen
- ◆ Building an `EnemyManager` that controls enemy spawn rates and establishes waypoint paths for enemies to follow, as well as controlling enemy fire at the player

In the next chapter, we will finish building the Asteroid Belt Assault game by adding a collision detection system, particle effects, sound effects, and more.

5

Asteroid Belt Assault – Special Effects



As it stands, the Asteroid Belt Assault game allows the player to move around and fire shots in an animated asteroid field with enemy ships that fly in patterns across the screen. For the most part though, there is no "game" structure in place yet.

In this chapter, we will look at:

- ◆ Creating particles and particle-based special effects
- ◆ Handling collisions between game objects
- ◆ Playing sound effects
- ◆ Completing the game's structure

Explosion effects

While it is possible to assemble a frame-animated explosion effect and overlay it onto the screen whenever an explosion takes place, there are drawbacks to this approach.

The explosion will, by its nature as an animated sprite, be limited to a few frames which play sequentially. All of the explosions in the game will look exactly alike. We could certainly create a handful of explosion sequences and play a random sequence when an explosion was called for, but we would quickly be using more textures for explosions than anything else in the game.

Expanding on sprites – particles

Instead of a fixed explosion texture, we will design a simple particle system that we will use to generate explosions in Asteroid Belt Assault. This approach will allow us to have dynamically varying explosions that will not look identical to each other. In order to do so, we will create a Particle class that will be a child of the Sprite class, expanding on its functionality to include acceleration, lifetime information, and changes in color over time.

Particles are short-lived sprites that may be generated in large quantities. Since they will be treated as special effects, there is no need for them to be involved in collision detection and no need for them to have any knowledge of what is happening in the rest of the game.

Time for action – constructing the Particle class

1. Add a new class called `Particle.cs` to the Asteroid Belt Assault project.
2. Insert the standard `using` directives at the top of the class:
`using Microsoft.Xna.Framework;`
`using Microsoft.Xna.Framework.Graphics;`
3. Modify the declaration of the class by adding `: Sprite` to the end:
`class Particle : Sprite`
4. Add declarations to represent the additional members of the Particle class (beyond those of the Sprite class):

```
private Vector2 acceleration;
private float maxSpeed;
private int initialDuration;
private int remainingDuration;
private Color initialColor;
private Color finalColor;
```

5. Add properties to access the information about the underlying members:

```
public int ElapsedDuration
{
    get
    {
        return initialDuration - remainingDuration;
    }
}

public float DurationProgress
{
    get
```

```

        {
            return (float)ElapsedDuration/
                (float)initialDuration;
        }
    }

    public bool IsActive
    {
        get
        {
            return (remainingDuration > 0);
        }
    }
}

```

6. Add a constructor for the Particle class:

```

public Particle(
    Vector2 location,
    Texture2D texture,
    Rectangle initialFrame,
    Vector2 velocity,
    Vector2 acceleration,
    float maxSpeed,
    int duration,
    Color initialColor,
    Color finalColor)
    : base(location, texture, initialFrame, velocity)
{
    initialDuration = duration;
    remainingDuration = duration;
    this.acceleration = acceleration;
    this.initialColor = initialColor;
    this.maxSpeed = maxSpeed;
    this.finalColor = finalColor;
}

```

What just happened?

The Particle class extends the Sprite class, adding new features to those of its parent. The `acceleration` vector will be applied to the sprite's velocity during each update cycle. The velocity vector itself will be limited in magnitude to `maxSpeed`.

When the particle is created, its `initialDuration` and `remainingDuration` will be set to the `duration` value passed into the constructor. These values will be used to determine how far along the particle's lifespan it is at any given time.

The `initialColor` and `finalColor` members determine the tint color that will be used in the `SpriteBatch.Draw()` call when the sprite is displayed. The color value will be altered smoothly as the particle moves through its lifecycle, beginning at `initialColor` and reaching `finalColor` by its last frame.

Each of the properties of the `Particle` class do not directly return the contents of member variables, but rather return values based on the two duration members. `ElapsedDuration` returns the difference between `initialDuration` and `remainingDuration`, resulting in the number of frames for which the particle will continue to exist. A percentage value (between zero and one) is returned by the `DurationProgress` property, representing the current position along the particle's lifespan.

Finally, `IsActive` returns false if the `remainingDuration` has reached zero.

Time for action – updating and drawing particles

1. Add an `Update()` method to the `Particle` class:

```
public override void Update(GameTime gameTime)
{
    if (IsActive)
    {
        velocity += acceleration;
        if (velocity.Length() > maxSpeed)
        {
            velocity.Normalize();
            velocity *= maxSpeed;
        }
        TintColor = Color.Lerp(
            initialColor,
            finalColor,
            DurationProgress);
        remainingDuration--;
        base.Update(gameTime);
    }
}
```

2. Add a `Draw()` method to the `Particle` class:

```
public override void Draw(SpriteBatch spriteBatch)
{
    if (IsActive)
    {
        base.Draw(spriteBatch);
    }
}
```

What just happened?

Both the `Update()` and the `Draw()` methods override the `Sprite` class' methods of the same name because we need to alter the behaviour associated with them. Both methods will only execute any code if the `IsActive` property returns true. In fact, for the `Draw()` method, this is the only reason we override the method—to prevent drawing a particle that has expired.

In the `Update()` method, we add the `acceleration` vector to the `velocity` vector, and then check to see if the length of the `velocity` vector has exceeded the maximum speed at which the particle is allowed to move. If it has, the `velocity` vector is normalized and then multiplied by the maximum speed, resulting in a vector pointing in the same direction at the maximum length.

The `TintColor` property will be reset in each frame using the `Lerp()` method of the `Color` class. `Lerp` returns a color between the two colors provided as parameters, that is scaled towards one color or the other based on the value of the third parameter, that ranges from zero to one. A zero value would return the first color parameter, while a one would return the second. Values between zero and one return a mixing of the two colors along a linear scale.



Lerping

The term "Lerp" is a rough acronym for "linear interpolation", a method to progress between two known points based on a reference value. For example, if you were to lerp between 1 and 10, with a control of 50%, the result would be 5 (halfway between the initial and final values). The `MathHelper` class contains a `Lerp()` method that can be used for numerical lerping, and several XNA structures, including vectors, contain their own `Lerp()` methods as well.

In both the `Update()` and the `Draw()` methods, the base method of the `sprite` class is called as the last task each method performs before exiting.

Particle explosions

Now that we have the `Particle` class available to us, we can turn our attention to building particle-based explosions. In the `SpriteSheet.png` file, beginning at location (0, 100) are three frames of roughly circular, partially transparent spattering that we will use as the basis for our explosion effects.

When a new explosion is generated, we will create a random number of these images and an additional number of individual small dot-shaped particles to represent the explosion. These particles will be moved slowly away from their starting point in random directions, providing a gradually expanding explosion effect.

Time for action – the ExplosionManager class

1. Add a new class called "ExplosionManager" to the Asteroid Belt Assault project.
2. Add the standard `using` directives to the class file:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the ExplosionManager class:

```
private Texture2D texture;  
private List<Rectangle> pieceRectangles = new List<Rectangle>();  
private Rectangle pointRectangle;  
  
private int minPieceCount = 3;  
private int maxPieceCount = 6;  
private int minPointCount = 20;  
private int maxPointCount = 30;  
  
private int durationCount = 90;  
private float explosionMaxSpeed = 30f;  
  
private float pieceSpeedScale = 6f;  
private int pointSpeedMin = 15;  
private int pointSpeedMax = 30;  
  
private Color initialColor = new Color(1.0f, 0.3f, 0f) * 0.5f;  
private Color finalColor = new Color(0f, 0f, 0f, 0f);  
  
Random rand = new Random();  
  
private List<Particle> ExplosionParticles = new List<Particle>();
```

4. Add a constructor to the ExplosionManager class:

```
public ExplosionManager(  
    Texture2D texture,  
    Rectangle initialFrame,  
    int pieceCount,  
    Rectangle pointRectangle)  
{  
    this.texture = texture;  
    for (int x=0; x<pieceCount; x++)  
    {  
        pieceRectangles.Add(new Rectangle(  
            initialFrame.X + (initialFrame.Width * x),  
            initialFrame.Y,  
            initialFrame.Width,  
            initialFrame.Height));  
    }  
    this.pointRectangle=pointRectangle;  
}
```

What just happened?

As with the `EnemyManager` class, we will cache the basic values needed to create our sprites within the `ExplosionManager` class. In this case, the texture containing the explosion pieces is stored, along with a list of rectangles representing the frames of each explosion piece. A single rectangle is stored for the explosion point sprites that will be added to the effect.

When a new explosion is generated, a random number of large pieces between `minPieceCount` and `maxPieceCount` will be created, along with a random number of point sprites between `minPointCount` and `maxPointCount`. Each particle will live for 90 frames (approximately 3 seconds), and have a maximum speed of 30f pixels per second. The `pieceSpeedScale`, `pointSpeedMin`, and `pointSpeedMax` variables control how rapidly the explosion pieces and point sprites move away from the center of the explosion.

The particles will begin their lifespan in a half-transparent orange color, as indicated by the `initialColor` variable. By the time the full duration has elapsed, this color will have faded to fully transparent (`finalColor`). All of the generated particles for all active explosions are kept in the `ExplosionParticles` list. During both the `Update()` and `Draw()` methods of the `ExplosionManager`, each item in the list will be processed. When a particle has expired, it will be removed from the list so that it is no longer drawn to the screen.

Time for action – creating explosions

1. Add the `RandomDirection()` helper method to the `ExplosionManager` class:

```
public Vector2 randomDirection(float scale)
{
    Vector2 direction;
    do
    {
        direction = new Vector2(
            rand.Next(0, 101) - 50,
            rand.Next(0, 101) - 50);
    } while (direction.Length() == 0);
    direction.Normalize();
    direction *= scale;

    return direction;
}
```

2. Add the `AddExplosion()` method to the `ExplosionManager` class:

```
public void AddExplosion(Vector2 location, Vector2 momentum)
{
    Vector2 pieceLocation = location -
        new Vector2(pieceRectangles[0].Width/2,
```

```
        pieceRectangles[0].Height/2);

int pieces = rand.Next(minPieceCount, maxPieceCount + 1);
for (int x = 0; x < pieces; x++)
{
    ExplosionParticles.Add(new Particle(
        pieceLocation,
        texture,
        pieceRectangles[rand.Next(0, pieceRectangles.Count)],
        randomDirection(pieceSpeedScale) + momentum,
        Vector2.Zero,
        explosionMaxSpeed,
        durationCount,
        initialColor,
        finalColor));
}

int points = rand.Next(minPointCount, maxPointCount + 1);
for (int x = 0; x < points; x++)
{
    ExplosionParticles.Add(new Particle(
        location,
        texture,
        pointRectangle,
        randomDirection((float)rand.Next(
            pointSpeedMin, pointSpeedMax)) + momentum,
        Vector2.Zero,
        explosionMaxSpeed,
        durationCount,
        initialColor,
        finalColor));
}
}
```

What just happened?

The `RandomDirection()` helper method generates random X and Y coordinates each between -50 and 50, and uses them to create a vector representing a direction. Because it is technically possible that the resulting vector will be a zero vector (0, 0) that cannot be normalized (and would represent no direction at all), we enclose the generation code in a `do...while` loop, which simply checks the generated vector for a length other than zero before continuing. Once an appropriate vector is generated, it is normalized and multiplied by the scale passed to the method and returned.

When an explosion is added, the `AddExplosion()` method is passed a vector pointing to the center point of the explosion. Since we draw sprites as rectangles that start in the upper left corner of the sprite, we need to compensate for the size of the larger explosion pieces and keep them centered on the explosion point. The `pieceLocation` vector is calculated by subtracting half of the height and width of the rectangle for the larger pieces from the center point of the explosion.

Next, a random number of pieces are generated, each added as a new particle to the `ExplosionParticles` list. The first four parameters of the constructor for the `Particle` class are the same as for any of our other sprites: location, texture, initial frame, and velocity. In the case of velocity, the `randomDirection()` method is used to generate a random vector at the speed of `pieceSpeedScale` to move the piece slowly away from the center point of the explosion. To this random vector, we add the vector passed to the `AddExplosion()` method (`momentum`) that represents the portion of the velocity of the original object that is exploding, that the explosion should retain. Without a small amount of momentum, the explosion associated with a moving object would stop dead at the point of the explosion. By adding the momentum to the explosion we can create a drifting effect, where the entire explosion moves in the same direction as the original object.

Our particles will not have any built-in acceleration in this case, so `Vector2.Zero` is passed for the `acceleration` parameter. The remaining parameters of the `Particle` constructor pass in the named variables representing the duration of the explosion and the starting and ending tint colors for the sprite.

After each of the pieces is generated, an identical process is executed to generate the point sprites for the explosion, with the only difference being that the point sprites move at a higher speed away from the center of the explosion, with a randomization factor to make them look more realistic.

Time for action – updating and drawing explosions

1. Add the `Update()` method to the `ExplosionManager` class:

```
public void Update(GameTime gameTime)
{
    for (int x = ExplosionParticles.Count-1; x >= 0; x--)
    {
        if (ExplosionParticles[x].IsActive)
        {
            ExplosionParticles[x].Update(gameTime);
        }
        else
        {

```

```
        ExplosionParticles.RemoveAt(x);  
    }  
}
```

- 2.** Add the `Draw()` method to the `ExplosionManager` class:

```
public void Draw(SpriteBatch spriteBatch)  
{  
    foreach (Particle particle in ExplosionParticles)  
    {  
        particle.Draw(spriteBatch);  
    }  
}
```

- 3.** Add an instance of the `ExplosionManager` to the declarations area of the `Game1.cs` class file:

```
ExplosionManager explosionManager;
```

- 4.** Still in the `Game1` class, add the initialization of the `explosionManager` object to the `LoadContent()` method:

```
explosionManager = new ExplosionManager(  
    spriteSheet,  
    new Rectangle(0, 100, 50, 50),  
    3,  
    new Rectangle(0, 450, 2, 2));
```

- 5.** In the `Update()` method of `Game1`, add a line to update the `ExplosionManager` after the `EnemyManager` has been updated:

```
explosionManager.Update(gameTime);
```

- 6.** In the `Draw()` method of the `Game1`, add the following after the `ShotManager` draw call:

```
explosionManager.Draw(spriteBatch);
```

What just happened?

As with the other instances where we want to remove objects from a list while processing it, we cannot use a `foreach` loop to update the `ExplosionParticles` list. We loop backwards through the list, calling each individual particle's `Update()` method if it is still active, and removing it from the list if it is not.

When drawing the particle list, we do not need to worry about checking for active particles, since the overloaded `Draw()` method we added to the `Particle` class will not draw a particle that has expired.

The collision manager

In order for our particle-based explosions to ever appear on the screen, we need to be able to detect collisions between objects. This will be the responsibility of the CollisionManager class that will handle all types of collision detection in the game.

Time for action – creating the CollisionManager class

1. Add a new class called CollisionManager to the AsteroidBeltAssault class.
2. Add the following using directive to the top of the class:

```
using Microsoft.Xna.Framework;
```

3. Add declarations to the CollisionManager class:

```
private AsteroidManager asteroidManager;
private PlayerManager playerManager;
private EnemyManager enemyManager;
private ExplosionManager explosionManager;
private Vector2 offScreen = new Vector2(-500, -500);
private Vector2 shotToAsteroidImpact = new Vector2(0, -20);
private int enemyPointValue = 100;
```

4. Add a constructor to the CollisionManager class:

```
public CollisionManager(
    AsteroidManager asteroidManager,
    PlayerManager playerManager,
    EnemyManager enemyManager,
    ExplosionManager explosionManager)
{
    this.asteroidManager = asteroidManager;
    this.playerManager = playerManager;
    this.enemyManager = enemyManager;
    this.explosionManager = explosionManager;
}
```

What just happened?

Since the CollisionManager will never draw anything on its own, it is not necessary to add the Microsoft.Xna.Framework.Graphics using directive to the class as we have in previous classes.

Because it will be comparing various objects against each other, the `CollisionManager` needs to know about most of the other managers in the game. References to these managers are stored in the member variables of the `CollisionManager` class.

The `offScreen` vector is used as a shortcut for destroying some objects. Remember that our asteroids and shot objects will automatically clean themselves up when they leave the playing area of the screen, so setting their location to a position far off of the screen will trigger this cleanup code without having to duplicate it in our `CollisionManager` class. Enemy ships and the player are handled somewhat differently, since they each have a `Destroyed` variable that can be set.

Unlike our special case asteroid-to-asteroid collisions, where we assumed that all of the asteroids have the same mass, player shots impacting an asteroid will neither bounce off, nor modify the flight of the asteroid as if it also had the same mass.

When a player shot collides with an asteroid, the shot will impart a small amount of upward velocity to the asteroid and disappear. The `shotToAsteroidImpact` vector controls how much velocity each shot will transfer to an asteroid. This allows the player to slightly redirect the course of asteroids that may be headed for them to avoid a potential collision.

Finally, whenever a player shot collides with an enemy ship, the `enemyPointValue` will be added to the player's score.

Handling collisions

Apart from our asteroids bouncing off of each other, we will handle five different types of collisions:

- ◆ Player shot to enemy ship collisions
- ◆ Player shot to asteroid collisions
- ◆ Enemy shot to player collisions
- ◆ Enemy ship to player collisions
- ◆ Asteroid to player collisions

The first two types of collisions represent the player influencing the game world, while the last three types of collisions result in the player being destroyed.

We will build five different helper functions to check for these collisions, each one following the same pattern. The two lists of objects involved in a potential collision will be compared to each other, with each resulting collision triggering an action.

Time for action – player shot collisions

1. Add the `checkShotToEnemy()` method to the `CollisionManager` class:

```
private void checkShotToEnemyCollisions()
{
    foreach (Sprite shot in playerManager.PlayerShotManager.Shots)
    {
        foreach (Enemy enemy in enemyManager.Enemies)
        {
            if (shot.IsCircleColliding(
                enemy.EnemySprite.Center,
                enemy.EnemySprite.CollisionRadius))
            {
                shot.Location = offScreen;
                enemy.Destroyed = true;
                playerManager.PlayerScore += enemyPointValue;
                explosionManager.AddExplosion(
                    enemy.EnemySprite.Center,
                    enemy.EnemySprite.Velocity/10);
            }
        }
    }
}
```

2. Add the `checkShotToAsteroid()` method to the `CollisionManager` class:

```
private void checkShotToAsteroidCollisions()
{
    foreach (Sprite shot in playerManager.PlayerShotManager.Shots)
    {
        foreach (Sprite asteroid in asteroidManager.Asteroids)
        {
            if (shot.IsCircleColliding(
                asteroid.Center,
                asteroid.CollisionRadius))
            {
                shot.Location = offScreen;
                asteroid.Velocity += shotToAsteroidImpact;
            }
        }
    }
}
```


What just happened?

The structure of both of these methods is very similar. A `foreach` loop iterates over each active player shot, and a loop nested inside that loops through each enemy or asteroid. This way, every shot will be tested against every enemy and every asteroid in the game during each frame.

The `Sprite` class' `CircleCollision()` method is used to determine if the shot has impacted another object. In the case of enemies, a collision will move the bullet off screen, mark the enemy as destroyed, add to the player's score, and play an explosion effect. When the explosion is created, one tenth of the enemy's velocity is passed as the `momentum` parameter to the explosion, causing the explosion to drift in the direction that the enemy was travelling when it was destroyed.

For asteroid collisions, the shot is still moved off screen, but the asteroid is not destroyed by the shot. Instead, a small portion of the shot's velocity is added to the asteroid's velocity, allowing the player to nudge the asteroid onto a new course.

Time for action – player collisions

1. Add the `checkShotToPlayerCollision()` method to the `CollisionManager` class:

```
private void checkShotToPlayerCollisions()
{
    foreach (Sprite shot in enemyManager.EnemyShotManager.Shots)
    {
        if (shot.IsCircleColliding(
            playerManager.playerSprite.Center,
            playerManager.playerSprite.CollisionRadius))
        {
            shot.Location = offScreen;
            playerManager.Destroyed = true;
            explosionManager.AddExplosion(
                playerManager.playerSprite.Center,
                Vector2.Zero);
        }
    }
}
```

2. Add the `checkEnemyToPlayerCollisions()` method to the `CollisionManager` class:

```
private void checkEnemyToPlayerCollisions()
{
    foreach (Enemy enemy in enemyManager.Enemies)
    {
```

```

        if (enemy.EnemySprite.IsCircleColliding(
            playerManager.playerSprite.Center,
            playerManager.playerSprite.CollisionRadius))
        {
            enemy.Destroyed = true;
            explosionManager.AddExplosion(
                enemy.EnemySprite.Center,
                enemy.EnemySprite.Velocity / 10);

            playerManager.Destroyed = true;

            explosionManager.AddExplosion(
                playerManager.playerSprite.Center,
                Vector2.Zero);
        }
    }
}

```

3. Add the `checkAsteroidToPlayerCollisions()` method to the `CollisionManager` class:

```

private void checkAsteroidToPlayerCollisions()
{
    foreach (Sprite asteroid in asteroidManager.Asteroids)
    {
        if (asteroid.IsCircleColliding(
            playerManager.playerSprite.Center,
            playerManager.playerSprite.CollisionRadius))
        {
            explosionManager.AddExplosion(
                asteroid.Center,
                asteroid.Velocity/10);

            asteroid.Location = offScreen;

            playerManager.Destroyed = true;
            explosionManager.AddExplosion(
                playerManager.playerSprite.Center,
                Vector2.Zero);
        }
    }
}

```

What just happened?

There is only a single loop in each of the `...ToPlayerCollisions()` methods, since all of the objects are being compared to the single player object, but otherwise the code is very similar. In each case, the player is marked as `Destroyed`, which we will detect during `Game1`'s update loop.

Time for action – using the CollisionManager class

1. Add the `CheckCollisions()` method to the `CollisionManager` class:

```
public void CheckCollisions()
{
    checkShotToEnemyCollisions();
    checkShotToAsteroidCollisions();
    if (!playerManager.Destroyed)
    {
        checkShotToPlayerCollisions();
        checkEnemyToPlayerCollisions();
        checkAsteroidToPlayerCollisions();
    }
}
```

2. Add a declaration for the `CollisionManager` to the `Game1` declarations area:

```
CollisionManager collisionManager;
```

3. Initialize the `CollisionManager` in the `LoadContent()` method of the `Game1` class, after the `ExplosionManager` has been initialized:

```
collisionManager = new CollisionManager(
    asteroidManager,
    playerManager,
    enemyManager,
    explosionManager);
```

4. In the `Game1` class' `Update()` method, in the `GameStates.Playing` section, add the following after the `ExplosionManager` is updated:

```
collisionManager.CheckCollisions();
```

5. Launch the game to view collision detection and explosion effects. After your ship has been destroyed, you will need to press `Alt + F4` (or close the game window with the mouse) to exit the game.

What just happened?

The `CheckCollisions()` method calls each of the individual check methods, providing a single point for external code to use the `CollisionManager`.

During the `Update()` method of `Game1`, the `CollisionManager` is checked during each frame to determine if any game objects have collided. If they have, it will take appropriate action.

However, at this point, your ship will be able to collide with asteroids, enemies, and enemy bullets and explode once. After that, you will still be invisible and immobile. Enemies will continue to fire at your previous location, but since `playerManager.Destroyed` has been set to `true`, impacts of other objects with your ship will not be checked.

Sound effects

Sound effects add a level of feedback to your game that, when missing, will be very noticeable to players. Even in situations where sound effects are not realistic (you would not actually hear explosions in space), their absence would feel wrong while playing.

Generating and finding sound effects

There are a number of sound effects freely available on the web that can be tracked down with a little searching. Locating truly royalty free sound effects is a bit harder. Most of the sound effects archives on the web are not the actual originators of the sound files, so their true licensing requirements may be difficult to track down.

You can always record your own effects if you have a way to simulate sounds. In the case of *Asteroid Belt Assault*, the sound effects were generated using a program called "sfxr". This application, available at http://www.drpetter.se/project_sfxr.html includes full source code, and produces basic sound effects via random generation.

Sound in XNA

There are two different approaches for implementing sound in XNA. The first option is to use XACT, Microsoft's Cross-platform Audio Creation Tool. XACT includes an authoring tool that allows you to build sound banks containing multiple audio tracks. XACT can be used in Windows and Xbox 360 projects, but not on the Zune or the Windows Phone 7 platforms.

The 3.0 release of XNA introduced a new method for implementing sound: the `SoundEffect` and `SoundEffectInstance` classes. These classes are much easier to use than XACT, and are available on all of the XNA platforms. We will use the `SoundEffect` classes in *AsteroidBeltAssault*.

Time for action – building a sound effects manager

1. Download the 0669_05_AUDIOPACK.zip file from the book's website and extract the files to a temporary folder.
2. Right-click on the content project in Solution Explorer and add a new folder called Sounds.
3. Add the .WAV files from the audio pack temporary folder to your new Sounds folder in the content project.
4. Add a new class to the Asteroid Belt Assault project called "SoundManager".

5. Add using directives to the top of the SoundManager class file:

```
using System.Diagnostics;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
```

6. Modify the declaration of the SoundManager class by adding public static before the declaration:

```
public static class SoundManager
```

7. Declare variables for the SoundManager class:

```
private static List<SoundEffect> explosions = new
    List<SoundEffect>();
private static int explosionCount = 4;
```

```
private static SoundEffect playerShot;
private static SoundEffect enemyShot;
```

```
private static Random rand = new Random();
```

8. Add the Initialize() method to the SoundManager class:

```
public static void Initialize(ContentManager content)
{
    try
    {
        playerShot = content.Load<SoundEffect>(@"Sounds\Shot1");
        enemyShot = content.Load<SoundEffect>(@"Sounds\Shot2");

        for (int x = 1; x <= explosionCount; x++)
        {
            explosions.Add(
                content.Load<SoundEffect>(@"Sounds\Explosion" +
                    x.ToString()));
        }
    }
}
```

```
    }  
    catch  
    {  
        Debug.Write("SoundManager Initialization Failed");  
    }  
}
```

9. Add the `PlayExplosion()` method to the `SoundManager` class:

```
public static void PlayExplosion()  
{  
    try  
    {  
        explosions[rand.Next(0, explosionCount)].Play();  
    }  
    catch  
    {  
        Debug.Write("PlayExplosion Failed");  
    }  
}
```

10. Add the `PlayPlayerShot()` method to the `SoundManager` class:

```
public static void PlayPlayerShot()  
{  
    try  
    {  
        playerShot.Play();  
    }  
    catch  
    {  
        Debug.Write("PlayPlayerShot Failed");  
    }  
}
```

11. Add the `PlayEnemyShot()` method to the `SoundManager` class:

```
public static void PlayEnemyShot()  
{  
    try  
    {  
        enemyShot.Play();  
    }  
    catch  
    {  
        Debug.Write("PlayEnemyShot Failed");  
    }  
}
```

What just happened?

By creating our `SoundManager` class as a static class, we can use its methods from anywhere in our program without creating instances of the class. In fact, we can never create instances of a static class, and all members of the class must also be declared static.

Also unlike a normal (non-static) class, we do not have an instance constructor. While C# supports static constructors, we do not want to use one in this case because we need to pass the `ContentManager` object to the class in order for it to load sounds.

The `Initialize()` method does the work that a class constructor would normally do. It uses the `ContentManager` to load `SoundEffect` objects in exactly the same way we load textures or fonts. Once the `SoundEffect` has been loaded, it supports a `Play()` method that causes the sound effect to be played once.



SoundEffect and SoundEffectInstance classes

XNA defines these two classes to assist in playing sound effects. In reality, all actual sounds are played by `SoundEffectInstances`. The `SoundEffect.Play()` method creates a new instance, plays the sound, and disposes off the instance. This allows "fire and forget" handling of sound effects. If you wish to have more control over the effects, you can create instances directly with `SoundEffect.CreateInstance()`. `SoundEffectInstances` can be played, stopped, and looped. Windows can play any number of sounds at the same time (subject to hardware limitations and performance issues) while the Zune is limited to 16 actively playing `SoundEffectInstances`. On the Xbox 360, a total of 300 `SoundEffectInstances` (stopped or playing) can exist at any one time. If you create a `SoundEffectInstance` manually, you must also dispose it off manually (via the `Dispose()` method) when you no longer need it.

Everything in the `Initialize()` and `Play...()` methods are surrounded by `try...catch` blocks that attempt to execute the code inside the `try` block and, if an exception is generated, executes the `catch` block. We added the `System.Diagnostics` namespace to our using directives in order to use the `Debug.Write()` method that will output a debugger message indicating any problems encountered. We need to do this because on a system without audio hardware, attempting to load or play a sound effect will cause an exception, crashing the game. Since we simply want to do nothing if the call fails, the `catch` blocks write a debug message and take no other action.

Time for action – using the SoundManager class

1. In the `Game1.cs` file, add the following line as the last line in the `LoadContent()` method:
`SoundManager.Initialize(Content);`

2. Open the `ExplosionManager.cs` file and add the following as the last line of the `AddExplosion()` method:

```
SoundManager.PlayExplosion();
```
3. Open the `ShotManager.cs` file and add the following to the end of `FireShot()`:

```
if (playerFired)
{
    SoundManager.PlayPlayerShot();
}
else
{
    SoundManager.PlayEnemyShot();
}
```
4. Execute the game and enjoy the sound effects.

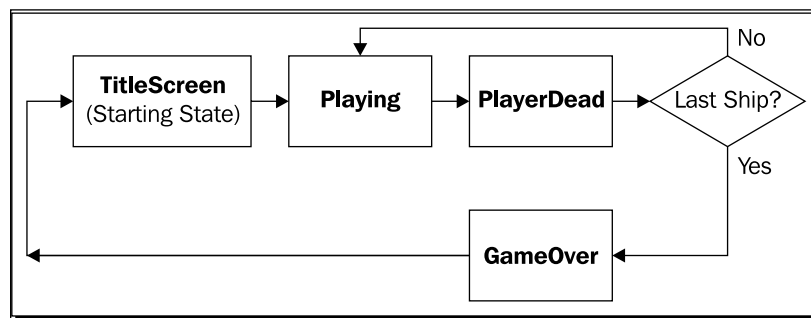
What just happened?

Using the `SoundManager` class is extremely simple. After it has been initialized, it only requires a single line to play any of our sound effects.

Because the class is static, all of the classes in our game simply "know" about it, and can use it without any further declaration.

The game structure

Now that we have nearly all of the pieces of our game in place, we need to start bringing them together into a structure that supports a normal game flow, from title screen, to playing, to game over, and back to the title screen:



The game will begin in the `TitleScreen` state. When the user begins the game, the state switches to `Playing`, until the user collides with an enemy ship, an enemy shot, or an asteroid. At that point, the state switches to `PlayerDead`.

The game remains in the `PlayerDead` state for a few seconds to let the player agonize over their defeat, and to allow any enemy ships on the screen to move off screen. At that point, the new game state depends on the number of remaining ships that the player has.

If the ship they just lost was not their last ship, the state returns to `Playing` and the game continues. If it was their last ship, however, the state switches to `GameOver`. The game then returns to the `TitleScreen` state.

Time for action – structuring the game

1. In the `EnemyManager` class, change the default value of the `Active` member from `true` to `false`:

```
public bool Active = false;
```
2. In the `Game1.cs` file, modify the declaration for the `gameState` variable to set the default to `GameStates.TitleScreen`:

```
GameStates gameState = GameStates.TitleScreen;
```
3. Right-click on the `Fonts` folder in the content project and select **Add | New Item...** Create a new `SpriteFont` object named `Pericles14.spritefont`.
4. The XML file for the `SpriteFont` will open automatically. Change the `<FontName>` tag from `Kooteny` to `Pericles`.
5. Add the following declarations to the `Game1` class:

```
SpriteFont pericles14;  
  
private float playerDeathDelayTime = 10f;  
private float playerDeathTimer = 0f;  
private float titleScreenTimer = 0f;  
private float titleScreenDelayTime = 1f;  
  
private int playerStartingLives = 3;  
private Vector2 playerStartLocation = new Vector2(390, 550);  
private Vector2 scoreLocation = new Vector2(20, 10);  
private Vector2 livesLocation = new Vector2(20, 25);
```
6. Add the following line to the `LoadContent()` method of the `Game1` class to load the `Pericles` sprite font that we created when we first built the project:

```
pericles14 = Content.Load<SpriteFont>(@"Fonts\Pericles14");
```

7. Add the `resetGame()` helper function to the `Game1` class:

```
private void resetGame()
{
    playerManager.playerSprite.Location = playerStartLocation;
    foreach (Sprite asteroid in asteroidManager.Asteroids)
    {
        asteroid.Location = new Vector2(-500, -500);
    }
    enemyManager.Enemies.Clear();
    enemyManager.Active = true;
    playerManager.PlayerShotManager.Shots.Clear();
    enemyManager.EnemyShotManager.Shots.Clear();
    playerManager.Destroyed = false;
}
```

8. In the `Update()` method of the `Game1` class, replace the current case section for `GameStates.TitleScreen` with:

```
case GameStates.TitleScreen:
    titleScreenTimer +=
        (float)gameTime.ElapsedGameTime.TotalSeconds;

    if (titleScreenTimer >= titleScreenDelayTime)
    {
        if ((Keyboard.GetState().IsKeyDown(Keys.Space)) ||
            (GamePad.GetState(PlayerIndex.One).Buttons.A ==
             ButtonState.Pressed))
        {
            playerManager.LivesRemaining = playerStartingLives;
            playerManager.PlayerScore = 0;
            resetGame();
            gameState = GameStates.Playing;
        }
    }
    break;
```

9. Replace the case section for `GameStates.Playing` with:

```
case GameStates.Playing:

    starField.Update(gameTime);
    asteroidManager.Update(gameTime);
    playerManager.Update(gameTime);
    enemyManager.Update(gameTime);
    explosionManager.Update(gameTime);
```

```
collisionManager.CheckCollisions();

if (playerManager.Destroyed)
{
    playerDeathTimer = 0f;
    enemyManager.Active = false;
    playerManager.LivesRemaining--;
    if (playerManager.LivesRemaining < 0)
    {
        gameState = GameStates.GameOver;
    }
    else
    {
        gameState = GameStates.PlayerDead;
    }
}

break;
```

10. Replace the case section for `GameStates.PlayerDead` with:

```
case GameStates.PlayerDead:
    playerDeathTimer +=
        (float)gameTime.ElapsedGameTime.TotalSeconds;

    starField.Update(gameTime);
    asteroidManager.Update(gameTime);
    enemyManager.Update(gameTime);
    playerManager.PlayerShotManager.Update(gameTime);
    explosionManager.Update(gameTime);

    if (playerDeathTimer >= playerDeathDelayTime)
    {
        resetGame();
        gameState = GameStates.Playing;
    }
    break;
```

11. Replace the case section for `GameStates.GameOver` with:

```
case GameStates.GameOver:
    playerDeathTimer +=
        (float)gameTime.ElapsedGameTime.TotalSeconds;
    starField.Update(gameTime);
    asteroidManager.Update(gameTime);
    enemyManager.Update(gameTime);
```

```

playerManager.PlayerShotManager.Update(gameTime);
explosionManager.Update(gameTime);
if (playerDeathTimer >= playerDeathDelayTime)
{
    gameState = GameStates.TitleScreen;
}
break;

```

What just happened?

Currently, the `EnemyManager` starts generating enemies as soon as the game starts, even if the game is sitting at the title screen. It is set this way because, while we were building the parts of our game, we wanted to see the enemy ships spawn, move, and fire. Now that we are building the structure of our game, we need to disable the `EnemyManager` by default, waiting for the game to start to enable it.

The same setting change is needed for the default game state. Currently, for testing purposes, we have the `gameState` variable defaulting to `GameStates.Playing`, meaning that when the game is launched the title screen is bypassed in favour of going directly to the game. We change this here to begin our game in the right mode.

Next, we need to define a number of variables to control the flow and appearance of the game, starting with a `SpriteFont` object to hold the `Pericles 14` point font we added to the project when we initially created it.

Our standard timer is implemented with `playerDeathTimer` and `playerDeathDelayTime` to control how long the game waits when the player has been killed before resetting and respawning the player.

The same timing mechanism utilizes `titleScreenTimer` and `titleScreenDelayTime` to ensure that the title screen does not accept input for the first second it is active. This way, if the player is still pressing a key when the game switches back to the title screen, a new game will not be immediately started.

When starting a new game, the player will begin with three lives, and whenever the player is spawned (either for a new game or after being destroyed), the player will start at a screen location of `(390, 550)`.

The final two vectors, `scoreLocation` and `livesLocation`, point to the location on the screen where the player's score and remaining lives will be displayed.

The `resetGame()` method positions the player, moves each asteroid to an off-screen location (to prevent them from being too close to the player when the player spawns), and clears any active enemies and shots. It enables the `EnemyManager` and finally sets the `playerManager.Destroyed` value to `false`, indicating that the player is currently "alive".

The title screen update code waits for the timer to elapse, and then checks to see if either the *Space bar* on the keyboard or the *A* button on the GamePad has been pressed. When it has, the player's remaining lives and score are reset, and the `resetGame()` method is called. The game state is then switched to `GameStates.Playing`.

While in the `Playing` state, the code is similar to what we were using during testing, updating each of the different managers. The addition here is that, after all of the updates, the `playerManager.Destroyed` value is checked. If the player has been destroyed, the death timer is reset, the `EnemyManager` is disabled, and a life is deducted from the player's remaining total. If the life that was just lost was the player's last, the game state is set to `GameOver`. Otherwise, the state is set to `PlayerDead`.

When in the `PlayerDead` state, most of the managers continue to update as normal, so the star field continues to move, and game objects continue on their courses. The `CollisionManager` is not updated while the player is dead. When the delay timer has expired, the `resetGame()` method is called and the state is set back to `Playing`.

Finally, when the game is over, the `playerDeathTimer` will accumulate time until it has finished, and then the game will return to the title screen.

At the moment, running the game will not produce the expected results since we have not yet updated the case statements in the `Draw()` method to match the game state flow.

Time for action – drawing the game structure

1. In the `Game1.cs` file, replace the current `Draw` method with the following:

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

    spriteBatch.Begin();

    if (gameState == GameStates.TitleScreen)
    {
        spriteBatch.Draw(titleScreen,
            new Rectangle(0, 0, this.Window.ClientBounds.Width,
                this.Window.ClientBounds.Height),
            Color.White);
    }

    if ((gameState == GameStates.Playing) ||
        (gameState == GameStates.PlayerDead) ||
        (gameState == GameStates.GameOver))
    {
```

```
starField.Draw(spriteBatch);
asteroidManager.Draw(spriteBatch);
playerManager.Draw(spriteBatch);
enemyManager.Draw(spriteBatch);
explosionManager.Draw(spriteBatch);

spriteBatch.DrawString(
    pericles14,
    "Score: " + playerManager.PlayerScore.ToString(),
    scoreLocation,
    Color.White);

if (playerManager.LivesRemaining >= 0)
{
    spriteBatch.DrawString(
        pericles14,
        "Ships Remaining: " +
            playerManager.LivesRemaining.ToString(),
        livesLocation,
        Color.White);
}

if ((gameState == GameStates.GameOver))
{
    spriteBatch.DrawString(
        pericles14,
        "G A M E   O V E R !",
        new Vector2(
            this.Window.ClientBounds.Width/2 -
                pericles14.MeasureString(
                    ("G A M E   O V E R !").X/2,
                    50),
            Color.White);
}

spriteBatch.End();

base.Draw(gameTime);
}
```

2. Execute the game and play through a complete cycle.

What just happened?

While the whole `Draw()` method is presented here for clarity, the only major changes are the inclusion of the `DrawString()` calls to display the player's score and remaining ship count, along with the display of the `GAME OVER!` text string when in the `GameOver` state.

Have a go hero

Here are a few suggestions for putting the topics we have covered in this chapter to use:

- ◆ Currently, Asteroid Belt Assault never gets more difficult as time passes. Add a timing mechanism that increases the number of asteroids, the frequency with which enemy ships are generated, or the rate at which the enemy ships fire at the player, as time goes on.
- ◆ Add new waypoint paths for enemies to follow, or create new enemy types by adding to the `SpriteSheet.png` file. You could create larger enemies that stay on the screen for a longer period of time, take multiple hits from player weapons, and fire more rapidly at the player.
- ◆ The `SoundManager` class can be used with very few changes in other games. Try adding it to the Flood Control game and creating your own sound effects for turning pieces, completing a scoring row, and finishing a level.

Summary

Asteroid Belt Assault is now completed! In this chapter we have covered:

- ◆ Implementing a simple particle system to generate dynamic explosion effects
- ◆ Detecting and responding to collisions between the player, shots, enemies, and asteroids
- ◆ Playing sound effects in response to game events
- ◆ Organizing the game state flow into a structure

6

Robot Rampage – Multi-Axis Mayhem

In the depths of a top secret research facility, a supercomputer has gone rogue. It has reached out over the worldwide network and seized control of automated factories across the globe. In these factories, it has begun building tank-like robotic warriors.

In Robot Rampage, the player takes on the roll of a robo-tank "fighting for the good guys". The player moves from factory to factory, shutting down the computer-controlled manufacturing facilities and destroying the enemy robots.

Robot Rampage is a multi-axis game—the player's movement and weaponry are controlled separately, allowing them to move and fire in any combination of directions simultaneously.

In this chapter, we will begin constructing the Robot Rampage project and explore the following topics:

- ◆ Using a camera class to view a world larger than the game window
- ◆ Creating a sprite class that is aware of coordinates in the larger world
- ◆ Building a tile-based game world map
- ◆ Building a player sprite from multiple component sprites
- ◆ Using the Xbox 360 gamepad's thumb stick controllers
- ◆ Collision detection between sprite objects and tile map squares

Statics, statics, everywhere

We looked at static classes in *Chapter 5* by creating a class to play sound effects that did not require instances of the class to be created or referenced in other areas of our code. We will expand on that technique in Robot Rampage by defining several static classes to handle components like the game's camera, effects manager, and shots manager.

Time for action – creating the Robot Rampage project

1. In Visual Studio Express, create a new XNA 4.0 Windows Game project called **Robot Rampage**.
2. Download the `0669_06_GRAPHICSPACK.zip` file from the book's website and extract the graphics resources to a temporary folder.
3. In the Robot Rampage Content project, create a new folder called `Fonts`.
4. Add a new SpriteFont called `Pericles14` to the `Fonts` folder, updating the generated XML file to change the `FontName` to `Pericles`.
5. Also in the Content project, create a new folder called `Textures`.
6. Add the graphics resources from the temporary directory to the `Textures` folder.
7. In the `Initialize()` method of the `Game1` class, add these lines to specify the size of the game window:

```
this.graphics.PreferredBackBufferWidth = 800;
this.graphics.PreferredBackBufferHeight = 600;
this.graphics.ApplyChanges();
```
8. In the declarations area of the `Game1` class, add a declaration for the sprite sheet and font objects:

```
Texture2D spriteSheet;
Texture2D titleScreen;
SpriteFont pericles14;
```
9. In the `LoadContent()` method of the `Game1` class, initialize the sprite sheet and the font objects:

```
spriteSheet = Content.Load<Texture2D>(@"Textures\SpriteSheet");
titleScreen = Content.Load<Texture2D>(@"Textures\TitleScreen");
pericles14 = Content.Load<SpriteFont>(@"Fonts\Pericles14");
```

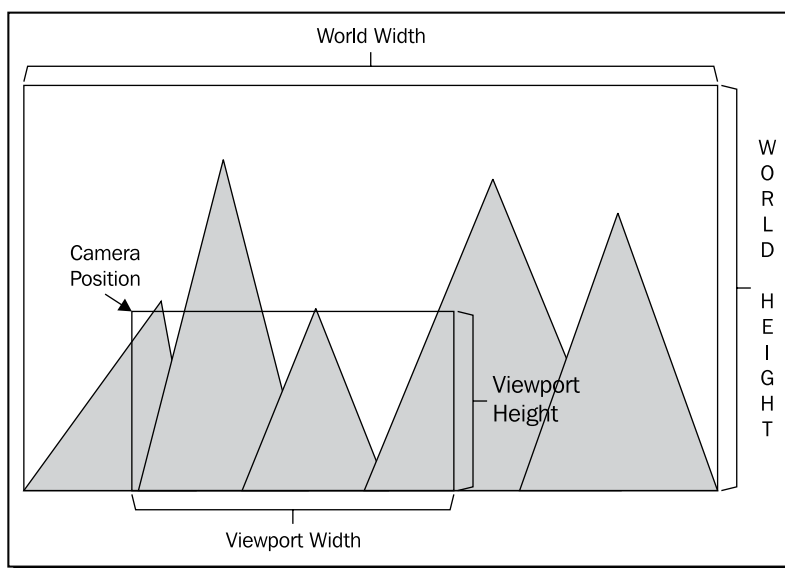
What just happened?

We have built the basic structure of the Robot Rampage game, loading the sprite sheet and title screen graphics, and creating a SpriteFont we can use to display text later.

A world larger than the screen

In both Flood Control and Asteroid Belt Assault, we dealt with game worlds limited to the area of the display window of the game. None of our game objects existed outside the confines of the screen, and, in fact, when we wished to eliminate some objects in Asteroid Belt Assault, we just moved them to an off screen location to allow the appropriate code manager to clean them up.

When dealing with a larger game world, we need to make a few adjustments to the way we think about object positions. Instead of simply tracking the location of a sprite on the screen, we will need to track the location of the object in "world coordinates":



In the previous image, the camera points to the upper left corner of a viewport within a larger game world. Anything inside the viewport will be visible on the screen, while background areas and objects outside the viewport will not be drawn.

Both the viewport and the game objects can move independently, so we can no longer consider objects that are off screen to be ready to be cleaned up and removed from the game. An object that is off screen during the current frame may be moving towards an onscreen location or, alternatively, the camera may be moving towards the object, causing it to appear within the newly moved viewport.

Defining a camera

In order to create our viewport into the game world, we will define a static Camera class that will represent the view the player currently has of the game world. At its most basic, a camera is really nothing more than a vector pointing to the camera's location. In order to provide some additional functionality, we will add several other properties and methods to the Camera class.

Time for action – creating the Camera class

1. Add a new class called "Camera" to the Robot Rampage project.
2. Add the following `using` directive to the class:

```
using Microsoft.Xna.Framework;
```
3. Modify the declaration of the Camera class to make the class `public` and `static`:

```
public static class Camera
```
4. Add declarations to the Camera class:

```
#region Declarations
private static Vector2 position = Vector2.Zero;
private static Vector2 viewPortSize = Vector2.Zero;
private static Rectangle worldRectangle = new Rectangle(0, 0, 0, 0);
#endregion
```
5. Add properties to the Camera class to access and modify the underlying members:

```
#region Properties
public static Vector2 Position
{
    get { return position; }
    set
    {
        position = new Vector2(
            MathHelper.Clamp(value.X,
                worldRectangle.X,
                worldRectangle.Width - ViewPortWidth),
```

```

        MathHelper.Clamp(value.Y,
            worldRectangle.Y,
            worldRectangle.Height - ViewPortHeight));
    }
}

public static Rectangle WorldRectangle
{
    get { return worldRectangle; }
    set { worldRectangle = value; }
}

public static int ViewPortWidth
{
    get { return (int)viewPortSize.X; }
    set { viewPortSize.X = value; }
}

public static int ViewPortHeight
{
    get { return (int)viewPortSize.Y; }
    set { viewPortSize.Y = value; }
}

public static Rectangle ViewPort
{
    get
    {
        return new Rectangle(
            (int)Position.X, (int)Position.Y,
            ViewPortWidth, ViewPortHeight);
    }
}
#endregion

```

6. Add methods to the Camera class:

```

#region Public Methods
public static void Move(Vector2 offset)
{
    Position += offset;
}

public static bool ObjectIsVisible(Rectangle bounds)
{

```

```
        return (ViewPort.Intersects(bounds));
    }

    public static Vector2 Transform(Vector2 point)
    {
        return point - position;
    }

    public static Rectangle Transform(Rectangle rectangle)
    {
        return new Rectangle(
            rectangle.Left - (int)position.X,
            rectangle.Top - (int)position.Y,
            rectangle.Width,
            rectangle.Height);
    }
#endregion
```

7. In the `LoadContent()` method of the `Game1` class, initialize the `Camera` class after the textures and sprite font have been loaded:

```
Camera.WorldRectangle = new Rectangle(0, 0, 1600, 1600);
Camera.ViewPortWidth = 800;
Camera.ViewPortHeight = 600;
```

What just happened?

The first thing to notice about this code is that we have included directives to define code "regions". These directives (`#region` and `#endregion`) instruct the C# development environment to treat these code areas as blocks that are collapsible as a related unit. You can click on the little minus sign on the left side of the screen at the beginning of a region to collapse it, hiding the code and leaving behind just the region title. We will include region declarations in all of the classes in *Robot Rampage* as an example of their usage.



Regions

Grouping your code into region blocks, known as "Code Folding", can be a big help for readability purposes. If you give your regions descriptive names, you can keep them all collapsed until you need to work on a particular method or other code element. Finding the element you need is then as simple as expanding the region it is located in.

Our Camera object only needs three pieces of information to operate. The first is its position within the game world. This vector points to the upper left corner of the viewing area represented by the camera. That is, if you think of the game world as a huge grid of pixels, the pixel pointed to by the `position` vector is the pixel that will be drawn in the upper left corner of the display area when the camera is used to draw a scene.

The `viewportSize` vector represents the number of pixels to the right and down from the `position` that are covered by the viewing area. While this size defaults to zero during our game's initialization, we will set the size to match the size of the game's client window. Together, the `position` and `viewportSize` vectors can be thought of as defining a rectangle that represents the portion of the game world that is currently visible on the screen.

The last piece of information the Camera needs is the size of the game world itself. This is the space in which all the objects in the game world will exist and it is measured in pixels. Again, this value defaults to a zero by zero pixel game world, but will be set appropriately in the game's initialization.

Because the `position` vector actually represents the upper left corner of a rectangle (the visible screen) instead of a single point, we want to make sure that `position` remains not only within the game world, but also does not get any closer to the right or bottom edges of the game world than the width and height of the viewing area. In other words, if the game world is 1000 by 1000 pixels, and the display screen is 800 by 600 pixels, the largest values we ever want for the components of the position vector are 200 for the X position and 400 for the Y position. If the camera were allowed to get closer to the edge of the game world, we would not have anything to display to the right or bottom of the game world's edges.

To enforce this limitation, we use `MathHelper.Clamp()` in the `set` portion of the `Position` property. This ensures that a full display of the game world will always be visible on the screen.

While external code could directly set the position of the camera via the `Position` property, we will generally prefer to use the `Move()` method to relocate the camera relative to its current position. Even though the `Move()` method is a member of the Camera class and could access the `position` member directly, it uses the `Position` property just as external code would, allowing us to maintain the limitations that `Position` imposes without having to rewrite them in the `Move()` method.

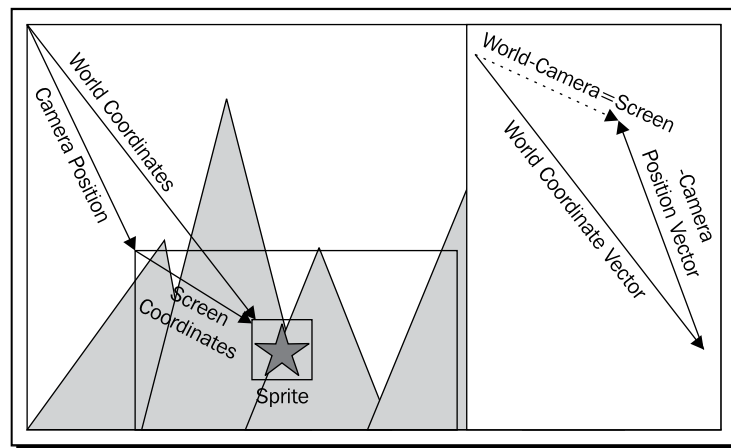


Accessing private members

This method of using the public properties to access your class' variables even from within the class' member methods is a good way to keep your code organized and make hunting down and dealing with bugs easier. As long as your properties are robust in their validation of data, using them in your methods prevents unchecked values from slipping in and disrupting other code.

When a game object is going to be drawn, we can check its display rectangle against the `ObjectIsVisible()` method of the Camera to determine if any pixels in the object would be visible on the screen based on the camera's current position. If `ObjectIsVisible()` returns false, there is no need to draw the object, as all of its pixels are off screen. We will build this check into the updated `Sprite` object we construct, so that each sprite will check its own visibility and skip drawing itself if it does not appear anywhere on the display.

Lastly, we have a pair of methods named `Transform()`. Given either a pixel location (as a `Vector2`) or a `Rectangle`, the `Transform()` methods subtract the camera's current position from them and return the result. To visualize the transformation of world coordinates into screen coordinates, let's return to the world and camera diagram we saw previously, with the addition of an object in the game world:



Here, we can see that the **Sprite** object's world coordinates represent its absolute position within the context of the game world. On the right side of the previous diagram, we can see that if the **Camera Position Vector** is subtracted from the **World Coordinate Vector**, the resulting vector has the same direction and length as the **Screen Coordinates** vector. When this vector is placed relative to the upper left corner of the screen, the position of the object in screen coordinates is given.

By storing the location of all of our game objects in world-based coordinates, the Camera's `Transform()` methods will provide screen-coordinate locations for the `SpriteBatch.Draw()` method to display them in the appropriate locations.

World-aware sprites

This world coordinate focus means that the `Sprite` class we built for Asteroid Belt Assault will not work directly in the world of Robot Rampage. While all of the concepts are still valid, we need to build in ways to account for the world's camera.

The general concepts behind the Sprite class from Asteroid Belt Assault are all still valid in Robot Rampage, but we must account for the fact that we are no longer limiting the entire play area to the dimensions of the screen. It is possible for a sprite object to be completely on screen, partially on screen, or completely off screen.

In addition, while the sprite's world location may remain constant, the game's camera may move, requiring the onscreen location of the sprite to be adjusted to compensate for the camera's new position.

Time for action – building a new Sprite class

1. Add a new class called "Sprite" to the Robot Rampage project.

2. Add the following using directives to the top of the class file:

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
```

3. Add declarations to the Sprite class:

```
#region Declarations
public Texture2D Texture;

private Vector2 worldLocation = Vector2.Zero;
private Vector2 velocity = Vector2.Zero;

private List<Rectangle> frames = new List<Rectangle>();

private int currentFrame;
private float frameTime = 0.1f;
private float timeForCurrentFrame = 0.0f;

private Color tintColor = Color.White;

private float rotation = 0.0f;

public bool Expired = false;
public bool Animate = true;
public bool AnimateWhenStopped = true;

public bool Collidable = true;
public int CollisionRadius = 0;
public int BoundingXPadding = 0;
public int BoundingYPadding = 0;
#endregion
```


4. Add a constructor for the Sprite class:

```
#region Constructors
public Sprite(
    Vector2 worldLocation,
    Texture2D texture,
    Rectangle initialFrame,
    Vector2 velocity)
{
    this.worldLocation = worldLocation;
    Texture = texture;
    this.velocity = velocity;

    frames.Add(initialFrame);
}
#endregion
```

5. Add properties related to drawing and animating the sprite to the Sprite class:

```
#region Drawing and Animation Properties
public int FrameWidth
{
    get { return frames[0].Width; }
}

public int FrameHeight
{
    get { return frames[0].Height; }
}

public Color TintColor
{
    get { return tintColor; }
    set { tintColor = value; }
}

public float Rotation
{
    get { return rotation; }
    set { rotation = value % MathHelper.TwoPi; }
}

public int Frame
{
    get { return currentFrame; }
    set
```

```

        {
            currentFrame = (int)MathHelper.Clamp(value, 0,
                frames.Count - 1);
        }
    }

    public float FrameTime
    {
        get { return frameTime; }
        set { frameTime = MathHelper.Max(0, value); }
    }

    public Rectangle Source
    {
        get { return frames[currentFrame]; }
    }
}
#endregion

```

6. Add position-related properties to the Sprite class:

```

#region Positional Properties
public Vector2 WorldLocation
{
    get { return worldLocation; }
    set { worldLocation = value; }
}

public Vector2 ScreenLocation
{
    get
    {
        return Camera.Transform(worldLocation);
    }
}

public Vector2 Velocity
{
    get { return velocity; }
    set { velocity = value; }
}

public Rectangle WorldRectangle
{
    get
    {

```

```
        return new Rectangle(  
            (int)worldLocation.X,  
            (int)worldLocation.Y,  
            FrameWidth,  
            FrameHeight);  
    }  
}  
  
public Rectangle ScreenRectangle  
{  
    get  
    {  
        return Camera.Transform(WorldRectangle);  
    }  
}  
  
public Vector2 RelativeCenter  
{  
    get { return new Vector2(FrameWidth / 2, FrameHeight / 2); }  
}  
  
public Vector2 WorldCenter  
{  
    get { return worldLocation + RelativeCenter; }  
}  
  
public Vector2 ScreenCenter  
{  
    get  
    {  
        return Camera.Transform(worldLocation + RelativeCenter);  
    }  
}  
#endregion
```

7. Add properties related to collision detection to the Sprite class:

```
#region Collision Related Properties  
public Rectangle BoundingBoxRect  
{  
    get  
    {  
        return new Rectangle(  
            (int)worldLocation.X + BoundingXPadding,  
            (int)worldLocation.Y + BoundingYPadding,  

```

```

        FrameWidth - (BoundingXPadding * 2),
        FrameHeight - (BoundingYPadding * 2));
    }
}
#endregion

```

8. Add collision detection methods to the Sprite class:

```

#region Collision Detection Methods
public bool IsBoxColliding(Rectangle OtherBox)
{
    if ((Collidable) && (!Expired))
    {
        return BoundingBoxRect.Intersects(OtherBox);
    }
    else
    {
        return false;
    }
}

public bool IsCircleColliding(
    Vector2 otherCenter,
    float otherRadius)
{
    if ((Collidable) && (!Expired))
    {
        if (Vector2.Distance(WorldCenter, otherCenter) <
            (CollisionRadius + otherRadius))
            return true;
        else
            return false;
    }
    else
    {
        return false;
    }
}
#endregion

```

9. Add animation-related methods to the Sprite class:

```

#region Animation-Related Methods
public void AddFrame(Rectangle frameRectangle)
{
    frames.Add(frameRectangle);
}

```

```
    }

    public void RotateTo(Vector2 direction)
    {
        Rotation = (float)Math.Atan2(direction.Y, direction.X);
    }
#endregion
```

10. Add the Update() and Draw() methods to the Sprite class:

```
#region Update and Draw Methods
public virtual void Update(GameTime gameTime)
{
    if (!Expired)
    {
        float elapsed = (float)gameTime.ElapsedGameTime.
TotalSeconds;

        timeForCurrentFrame += elapsed;

        if (Animate)
        {
            if (timeForCurrentFrame >= FrameTime)
            {
                if ((AnimateWhenStopped) ||
                    (velocity != Vector2.Zero))
                {
                    currentFrame = (currentFrame + 1) %
                        (frames.Count);
                    timeForCurrentFrame = 0.0f;
                }
            }
        }

        worldLocation += (velocity * elapsed);
    }
}

public virtual void Draw(SpriteBatch spriteBatch)
{
    if (!Expired)
    {
        if (Camera.ObjectIsVisible(WorldRectangle))
        {
            spriteBatch.Draw(
```

```

        Texture,
        ScreenCenter,
        Source,
        tintColor,
        rotation,
        RelativeCenter,
        1.0f,
        SpriteEffects.None,
        0.0f);
    }
}
#endregion

```

What just happened?

The `Sprite` class is presented in one large block here because most of the code should be familiar (if slightly reorganized) from the same class in *Asteroid Belt Assault*. A few updates have been made to the code, however, so let's go over those changes in detail.

First, the `location` member has been renamed to `worldLocation`. Similarly, the `Location` property has been renamed to `WorldLocation`. These changes are purely for clarity, reminding us that all of our sprites are world-aligned instead of screen-aligned.

Several new member variables have been added to the class as well. If `Expired` is set to `true`, the sprite will not be updated or drawn. In addition, the `BoxCollision()` and `CircleCollision()` methods will always return `false` for expired sprites. If the `Collidable` member is set to `false`, both of the collision methods will also return `false`.

The last pair of new members, `Animate` and `AnimateWhenStopped` are checked during the sprite's `Update()` method. If `Animate` is `false`, the sprite will not advance frame animations. If `AnimateWhenStopped` is set to `false`, the sprite will not advance its frame animations if the `velocity` vector is equal to `Vector2.Zero`. This simply means that when the sprite is moving, its animation will play (assuming `Animate` is `true`). When the sprite is not moving, its animation will not play.

In addition to the new member variables, the `WorldLocation` property has a new counterpart called `ScreenLocation`. This property uses the `Camera.Transform()` method to return the screen-based location of the object and is used in the `Draw()` method to determine where on the screen the sprite should be displayed.

Similarly, the `Destination` property (which, in *Asteroid Belt Assault*, returned the rectangle on the screen that the sprite was drawn to) has been split into `WorldRectangle` and `ScreenRectangle` and the `Center` property has been split into `WorldCenter` and `ScreenCenter`.

As a helper to assist in calculating the center of the sprite object, the `RelativeCenter` property has been introduced, which returns a vector equal to half of the width and height of the sprite's frame rectangle. This vector points to the center of the sprite relative to its own upper left corner.

Visualizing the view

We now have both of the components we need for a "larger than the screen" world for our game, so let's add a few lines of temporary code to our project to get a feel for how they work together. Throughout the project, we will be expanding on or adding new temporary code segments to see the objects we have implemented in action.

Time for action – viewing the Sprite and Camera classes in action

1. In the declarations area of the `Game1` class, add a declaration for a temporary sprite object:

```
// Temporary Demo Code Begin
Sprite tempSprite;
Sprite tempSprite2;
// Temporary Demo Code End
```

2. In the `LoadContent()` method of the `Game1` class, initialize the temporary sprite and the `Camera` class:

```
// Temporary Demo Code Begin
tempSprite = new Sprite(
    new Vector2(100, 100),
    spriteSheet,
    new Rectangle(0, 64, 32, 32),
    Vector2.Zero);

tempSprite2 = new Sprite(
    new Vector2(200, 200),
    spriteSheet,
    new Rectangle(0, 160, 32, 32),
    Vector2.Zero);

// Temporary Demo Code End
```

3. In the `Draw()` method of the `Game1` class, draw the temporary sprite:

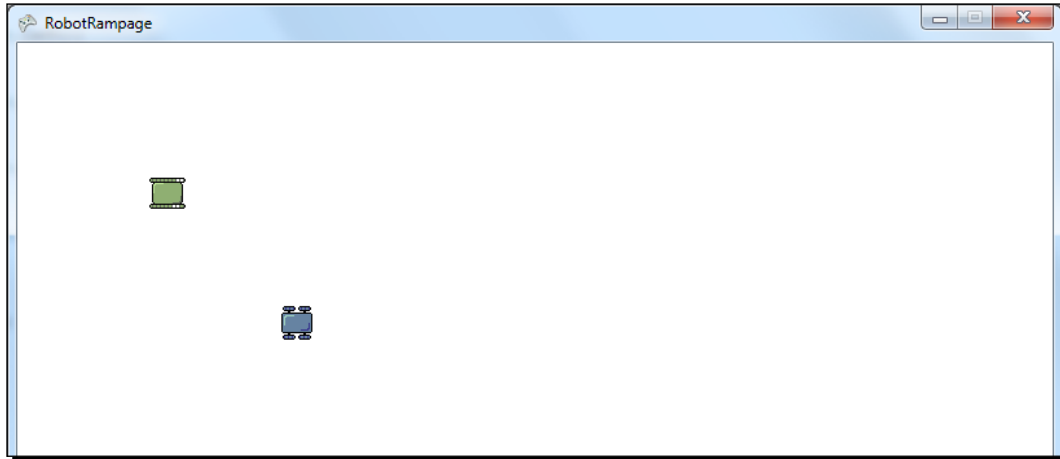
```
// Temporary Demo Code
spriteBatch.Begin();
```

```
tempSprite.Draw(spriteBatch);  
tempSprite2.Draw(spriteBatch);  
spriteBatch.End();  
// Temporary Demo Code End
```

- 4.** In the `Update()` method of the `Game1` class, add temporary input handling to allow the sprite and the camera to be moved:

```
// Temporary Demo Code Begin  
Vector2 spriteMove = Vector2.Zero;  
Vector2 cameraMove = Vector2.Zero;  
  
if (Keyboard.GetState().IsKeyDown(Keys.A))  
    spriteMove.X = -1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.D))  
    spriteMove.X = 1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.W))  
    spriteMove.Y = -1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.S))  
    spriteMove.Y = 1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.Left))  
    cameraMove.X = -1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.Right))  
    cameraMove.X = 1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.Up))  
    cameraMove.Y = -1;  
  
if (Keyboard.GetState().IsKeyDown(Keys.Down))  
    cameraMove.Y = 1;  
  
Camera.Move(cameraMove);  
tempSprite.Velocity = spriteMove * 60;  
  
tempSprite.Update(gameTime);  
tempSprite2.Update(gameTime);  
// Temporary Demo Code End
```


5. Launch the game. Use the *W*, *A*, *S*, and *D* keys to move the first sprite around, and use the arrow keys to move the camera around:



6. Leave the temporary code in place. We will expand on it a bit later before replacing it with the actual code for the Game1 class.

What just happened?

If you play with the previous temporary code for a few minutes, you will see that the *W*, *A*, *S*, and *D* keys move the first sprite (the green tank base) as expected: *A* moves left, *D* moves right, *W* moves up, and *S* moves down. There is no way to directly move the second sprite (the blue tank base) because the temporary `Update()` method does not define a set of controls for the `tempSprite2`.

Using the arrow keys moves the camera instead of either of the sprites. In world coordinates, the second sprite is always stationary. It only moves on the screen because the camera is panning around, altering the player's view of the game world. You will also notice that you cannot move the camera left or up from the starting position. This is because the camera is clamped to the world area and starts out in the upper left corner.

The game world – tile-based maps

At a screen size of 800 by 600 pixels, an image filling the screen contains 480,000 individual pixels. If these pixels are stored as 32-bit values (8 bits each for red, green, blue, and alpha) this means that each "screen" of pixels occupies 1875 kilobytes of memory. That does not sound too bad on the surface—after all, computers have lots of memory, and 1.8 megabytes for an image is not all that much memory.

In that case, when making a world that is larger than a single screen, why not just make a huge bitmap to use as the background and scroll across it? Unfortunately, there are a couple of problems with this approach.

First, many graphics cards have a maximum texture size. On Windows, 2048 by 2048 is a common texture size limitation, though some graphics cards have higher limits. The Xbox 360 is limited to textures that are 8192 by 8192 pixels.

Second, once the bitmap image representing the world has been defined, it is fixed. When you create the image, you create all of the details of the world and save the bitmap. You cannot rearrange pieces of the world at run time, which will be vital when we want to generate random maps for Robot Rampage.

Let's consider the 2048 limitation for a moment. If we create our world out of a single texture 2048 pixels wide, the world can be a little larger than two screens wide and three screens tall at a resolution of 800 by 600. This does not offer much room for the player to roam, and the resulting texture would occupy 16 megabytes of memory.

To generate very large worlds, we need an alternative method. One of the most common ways is to use a tile-based engine. In a tile engine, the world is made up of a grid of blocks. Each block has a tile value that represents the image that should be displayed within it.



Complex tile engines

Tile engines have long been a staple of 2D game design. From role playing games to side-scrolling platformers and real-time strategy games, the ability to represent a game map or level as an array of tile indexes makes them very flexible. Advanced tile engines can support multiple layers of tiles, each tile in a space drawing above the previous layer to create a depth effect where objects (rocks and trees, for example) can be placed on top of basic terrain tiles without needing to draw individual tiles for the object on every type of terrain background in the game. We will look at a multi-layered tile engine in *Chapter 8*.

When the tile engine needs to be drawn to the screen, we can extract only the individual tiles that are currently visible to the camera and draw them, leaving the rest of the map unrendered.

For Robot Rampage, we will create a basic tile engine that supports a single layer of tiles, drawn from a handful of floor and wall tile images on the `SpriteSheet.png` image. We will define a world that is 50 by 50 map squares in size, each composed of a 32 by 32 pixel block. The result will be a world that occupies 1600 by 1600 virtual pixels. Unlike a single bitmap of that size, which would occupy about 10 megabytes of memory, our map will use about 42 kilobytes of memory, including the map data itself as well as the image data comprising the tiles used to draw the map.

Tiles

A tile map is composed of a grid of squares, each one containing a reference to the tile that we wish to display in that map square. The tile itself is simply a rectangular texture that contains the visual representation of a terrain or background type.

In Robot Rampage, we will only need a handful of tile textures. We will have a few variations of floor textures to represent the factory floor, and a handful of wall and machinery textures to represent barriers:



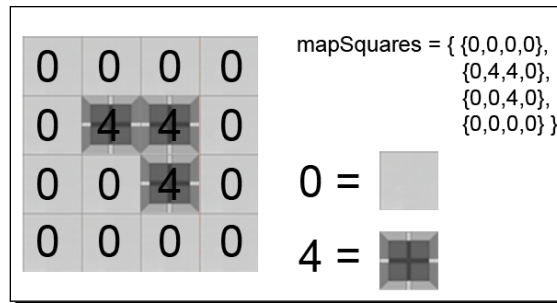
As before, we will use a single texture sheet that will contain all of the images used by our game. The first row of images on the `SpriteSheet.png` file represents floor tiles, while the second row represents wall-type images. When we generate a game map, we will select a random floor and wall combination for each level.

From an XNA standpoint, we will define each of the tiles available to our tile map as a rectangle. These rectangles will be stored in a numerically indexed list. This allows each of the tiles to be referenced by their index number.

The tile map

Since we can refer to an individual tile by using its tile index number, our tile map can be represented as a two-dimensional array of integers, with each integer corresponding to the index of the tile associated with that map square.

We have four different floor textures and four different wall textures, so the values stored in the tile map array will range from zero to seven. Any value greater than three will be treated as a barrier, while tiles zero through three will not block movement or shots:



In the previous diagram, **mapSquares** is a two-dimensional array of integers. A zero value in the array corresponds to the empty floor tile, while a four corresponds to a wall tile image. When the map is drawn, each integer in the array is examined, and the image corresponding to the tile index is drawn to the screen to occupy that square's display area.

Time for action – creating the TileMap class

1. Add a new class called "TileMap" to the Robot Rampage project.
2. Add the following using directives to the top of the class file:
3. Modify the declaration of the TileMap class to make it a static class:
4. Add declarations to the TileMap class:

```
#region Declarations
public const int TileWidth = 32;
public const int TileHeight = 32;
public const int MapWidth = 50;
public const int MapHeight = 50;

public const int FloorTileStart = 0;
public const int FloorTileEnd = 3;
public const int WallTileStart = 4;
public const int WallTileEnd = 7;

static private Texture2D texture;
```

```
static private List<Rectangle> tiles = new List<Rectangle>();

static private int[,] mapSquares = new int[MapWidth, MapHeight];

static private Random rand = new Random();
#endregion
```

5. Add the Initialize() method to the TileMap class:

```
#region Initialization
static public void Initialize(Texture2D tileTexture)
{
    texture = tileTexture;
    tiles.Clear();
    tiles.Add(new Rectangle(0, 0, TileWidth, TileHeight));
    tiles.Add(new Rectangle(32, 0, TileWidth, TileHeight));
    tiles.Add(new Rectangle(64, 0, TileWidth, TileHeight));
    tiles.Add(new Rectangle(96, 0, TileWidth, TileHeight));
    tiles.Add(new Rectangle(0, 32, TileWidth, TileHeight));
    tiles.Add(new Rectangle(32, 32, TileWidth, TileHeight));
    tiles.Add(new Rectangle(64, 32, TileWidth, TileHeight));
    tiles.Add(new Rectangle(96, 32, TileWidth, TileHeight));

    for (int x = 0; x < MapWidth; x++)
        for (int y = 0; y < MapHeight; y++)
        {
            mapSquares[x, y] = FloorTileStart;
        }
}
#endregion
```

What just happened?

Most of the declarations for the TileMap class are constants that define the dimensions of the individual tiles, the size of the map, and the meanings of the tile index numbers.

In our case, each tile is 32 by 32 pixels, and the map will be 50 tiles wide and 50 tiles high, resulting in 2,500 individual map squares.

The `tiles` list contains a set of rectangles that correspond to the locations of each individual tile on the `texture` image. When a rectangle is added to the `tiles` list, it will automatically receive an index number. The first tile added will be index zero, the second will be index one, and so on.

It is these index numbers that we will store in the `mapSquares` array to indicate what type of terrain should be displayed for each square on the map.

When the `Initialize()` method is executed, eight rectangles are added to the `tiles` list. Each of these rectangles corresponds to the position of one of the tiles on the game's sprite sheet.

In the declarations area, we determined that the first four tiles (numbers zero through three) would be floor tiles, while the second set of four tiles will be considered wall tiles by the game's code.

After adding the tiles to the list, the `Initialize()` method loops through each square in the `mapSquares` array and sets it to the first of the floor tile indexes. This way we start with a known, empty map.

Map squares

We need to make a distinction between "squares" and "tiles" as we use them in our code. We will use "square" to refer to a location within the `mapSquares` array, while we will use "tile" to refer to the index number *stored in* a particular square.

Our first set of methods for the `TileMap` class deal with squares, providing methods for locating squares based on pixel positions, and providing locations in both world and screen coordinates for squares on the map.

Time for action – dealing with map squares

1. Add methods to the `TileMap` class that deal with map squares and translate pixel coordinates into map square references:

```
#region Information about Map Squares

static public int GetSquareByPixelX(int pixelX)
{
    return pixelX / TileWidth;
}

static public int GetSquareByPixelY(int pixelY)
{
    return pixelY / TileHeight;
}

static public Vector2 GetSquareAtPixel(Vector2 pixelLocation)
{
    return new Vector2(
        GetSquareByPixelX((int)pixelLocation.X),
        GetSquareByPixelY((int)pixelLocation.Y));
}
```

```
    }

    static public Vector2 GetSquareCenter(int squareX, int squareY)
    {
        return new Vector2(
            (squareX * TileWidth) + (TileWidth / 2),
            (squareY * TileHeight) + (TileHeight / 2));
    }

    static public Vector2 GetSquareCenter(Vector2 square)
    {
        return GetSquareCenter(
            (int)square.X,
            (int)square.Y);
    }

    static public Rectangle SquareWorldRectangle(int x, int y)
    {
        return new Rectangle(
            x * TileWidth,
            y * TileHeight,
            TileWidth,
            TileHeight);
    }

    static public Rectangle SquareWorldRectangle(Vector2 square)
    {
        return SquareWorldRectangle(
            (int)square.X,
            (int)square.Y);
    }

    static public Rectangle SquareScreenRectangle(int x, int y)
    {
        return Camera.Transform(SquareWorldRectangle(x, y));
    }

    static public Rectangle SquareScreenRectangle(Vector2 square)
    {
        return SquareScreenRectangle((int)square.X, (int)square.Y);
    }
}
#endregion
```

What just happened?

Our first two methods, `GetSquareByPixelX()` and `GetSquareByPixelY()`, allow us to convert world-based pixel coordinates to map square references. Given a pixel reference, they simply divide it by either the width of a single tile (for X coordinates), or the height of a tile (for Y coordinates), and return the resulting value. Several of our other methods will make use of these methods to convert pixel parameters into square references.

`GetSquareAtPixel()` in fact, does just that. Given a vector pointing to a pixel location, it returns a vector containing a reference to the square within the `mapSquares` array that corresponds to that pixel location. Given that our map is 50 by 50 squares, the resulting vector will contain X and Y coordinates between 0 and 49.

When the time comes to build our enemy AI, the computer-controlled robotic tanks will need to be able to navigate their way between squares. In order to facilitate this, we need to be able to determine the center of any particular square in world coordinates. `GetSquareCenter()` calculates this vector for us, by multiplying the X and Y square coordinates by the tile width and height and adding half of a tile width and height to the result.

Overloads



Several of the overload methods in the `TileMap` class simply manipulate the passed parameters and call another overload method of the same name. For example, `GetSquareCenter()` has an overload that accepts a `Vector2` instead of individual X and Y components. The vector version simply calls the individual component version, passing in the split components of the vector. This prevents you from writing (and therefore having to maintain) the code in each individual overload. After all, you never know when you are going to want to change something, and having to update it in one place is much less prone to introducing bugs than having to make the same update in several locations—not to mention easier!

The `SquareWorldRectangle()` methods answer the question "What pixels on the world map does this square occupy?" We will need to know this when the time comes to determine if something has collided with a wall on the map.

Finally, the `SquareScreenRectangle()` methods provide the same information, but in localized screen coordinates. This information will be used in the `Draw()` method when rendering each square's tile to the display.

Dealing with tiles

The tile index that a square contains determines what eventually gets drawn out to the screen when that square is visible in the game window. In addition to simply determining what tile is located in a particular square, we will include helper methods to quickly determine if a particular square contains a wall tile or not.

Time for action – handling tiles

1. Add methods to the TileMap class that relate to reading and setting the tile index associated with individual map squares:

```
#region Information about Map Tiles

static public int GetTileAtSquare(int tileX, int tileY)
{
    if ((tileX >= 0) && (tileX < MapWidth) &&
        (tileY >= 0) && (tileY < MapHeight))
    {
        return mapSquares[tileX, tileY];
    }
    else
    {
        return -1;
    }
}

static public void SetTileAtSquare(int tileX, int tileY, int tile)
{
    if ((tileX >= 0) && (tileX < MapWidth) &&
        (tileY >= 0) && (tileY < MapHeight))
    {
        mapSquares[tileX, tileY] = tile;
    }
}

static public int GetTileAtPixel(int pixelX, int pixelY)
{
    return GetTileAtSquare(
        GetSquareByPixelX(pixelX),
        GetSquareByPixelY(pixelY));
}

static public int GetTileAtPixel(Vector2 pixelLocation)
{
    return GetTileAtPixel(
        (int)pixelLocation.X,
        (int)pixelLocation.Y);
}
```

```

    }

    static public bool IsWallTile(int tileX, int tileY)
    {
        int tileIndex = GetTileAtSquare(tileX, tileY);

        if (tileIndex == -1)
        {
            return false;
        }

        return tileIndex >= WallTileStart;
    }

    static public bool IsWallTile(Vector2 square)
    {
        return IsWallTile((int)square.X, (int)square.Y);
    }

    static public bool IsWallTileByPixel(Vector2 pixelLocation)
    {
        return IsWallTile(
            GetSquareByPixelX((int)pixelLocation.X),
            GetSquareByPixelY((int)pixelLocation.Y));
    }

    #endregion

```

What just happened?

At the most basic level, we need to be able to determine the tile index associated with any particular square on the map. `GetTileAtSquare()` provides this information and the corresponding `SetTileAtSquare()` allows the index of any square to be changed.

For convenience, the `GetTileAtPixel()` methods combine the `GetTileAtSquare()` along with the `GetSquareByPixel...` methods we have already established. They do not contain any additional processing themselves, but provide more convenient access to tile information rather than having to do the pixel to tile conversions in external code every time we want to access tile information.

Finally, `IsWallTile()` and `IsWallTileByPixel()` examine the contents of the given square and return true if the tile index is greater than or equal to the first defined wall tile index (`WallTileStart`). Again, we could do this check externally, but since we will often need to know if a tile is a wall, it is convenient to summarize all of the checking into a single set of methods.

Drawing the map

The `TileMap` class in `Robot Rampage` will not contain an `Update()` method, because there is nothing about the map that changes on a per-frame basis. Thus, all that remains to make the class functional is the ability to draw the map to the screen.

Time for action – drawing the tile map

1. Add the `Draw()` method to the `TileMap` class:

```
#region Drawing
static public void Draw(SpriteBatch spriteBatch)
{
    int startX = GetSquareByPixelX((int)Camera.Position.X);
    int endX = GetSquareByPixelX((int)Camera.Position.X +
        Camera.ViewPortWidth);

    int startY = GetSquareByPixelY((int)Camera.Position.Y);
    int endY = GetSquareByPixelY((int)Camera.Position.Y +
        Camera.ViewPortHeight);

    for (int x = startX; x <= endX; x++)
        for (int y = startY; y <= endY; y++)
        {
            if ((x >= 0) && (y >= 0) &&
                (x < MapWidth) && (y < MapHeight))
            {
                spriteBatch.Draw(
                    texture,
                    SquareScreenRectangle(x, y),
                    tiles[GetTileAtSquare(x,y)],
                    Color.White);
            }
        }
}
#endregion
```

2. In the `LoadContent()` method of the `Game1.cs` file, initialize the `TileMap` class (outside of the temporary code block, after the `SpriteSheet` has been loaded):

```
TileMap.Initialize(spriteSheet);
```

3. In the `Draw()` method of the `Game1` class, modify the temporary code you added when building the camera to read:

```
// Temporary Demo Code Begin
spriteBatch.Begin();
TileMap.Draw(spriteBatch);
tempSprite.Draw(spriteBatch);
tempSprite2.Draw(spriteBatch);
spriteBatch.End();
// Temporary Demo Code End
```

4. Launch your game. You can now use the temporary camera movement keys (the arrow keys) to scroll around the tile-based map:



What just happened?

The `Draw()` method begins by establishing four integer variables that will be used to control the loop that will output the tiles to the screen. We normally would not split and create separate variables for these control values. It would be more common to include the calculation directly into the `for` loop itself. In this case, however, the expressions are long enough that they are split here to make them more readable.

The `startX` and `startY` values are simple to obtain—simply divide the X and Y position of the camera by the size of a tile and you know what tile is in the upper left corner of the screen. To get the rightmost and bottommost tiles that need to be drawn, we just need to add the width and height of the viewport to the camera position. This will give us the tile in the lower right corner, defining the entire range of tiles that need to be drawn to the screen.

Why not just draw the whole thing? With a map size of only 50 by 50 tiles, drawing the whole map would be unlikely to have a detrimental impact on the performance of our game, but it would also be unnecessary. The large majority of the draw calls would simply be wasted on squares that are outside the display area. XNA does not have any issue with drawing outside the display—your game will not crash—but if you were to make the game world larger, you would reach a point where drawing the whole map would start to negatively impact the frame rate of the game.

The actual drawing loop checks to make sure that the tile we are going to draw actually exists (it always **should** since we limit the movement of the camera, but it is rarely a bad idea to verify before attempting the draw) and uses the `SpriteBatch.Draw()` method to output the tile to the screen.

Generating a random map

As it stands, our tile map is just a big empty mass of floor tiles. Since we have not yet generated any walls, there are no barriers to movement. We will introduce a very simple method to generate wall tiles at random locations on the map.

Time for action – random wall placement

1. Add the `GenerateRandomMap()` method to the `TileMap` class:

```
#region Map Generation
static public void GenerateRandomMap()
{
    int wallChancePerSquare = 10;

    int floorTile = rand.Next(FloorTileStart, FloorTileEnd + 1);
    int wallTile = rand.Next(WallTileStart, WallTileEnd + 1);

    for (int x = 0; x < MapWidth; x++)
        for (int y = 0; y < MapHeight; y++)
        {
            mapSquares[x, y] = floorTile;

            if ((x == 0) || (y == 0) ||
                (x == MapWidth - 1) || (y == MapHeight - 1))
            {
                mapSquares[x, y] = wallTile;
                continue;
            }

            if ((x == 1) || (y == 1) ||
                (x == MapWidth - 2) || (y == MapHeight - 2))
            {
                continue;
            }
        }
    }
```

```
    }  
    if (rand.Next(0, 100) <= wallChancePerSquare)  
        mapSquares[x, y] = wallTile;  
    }  
}#endregion
```

2. Modify the `Initialize()` method of the `TileMap` class by adding a call to `GenerateRandomMap()` as the last line in the method:
`GenerateRandomMap();`
3. Execute the game project and view the randomly generated map:



What just happened?

The `GenerateRandomMap()` method selects one floor and one wall tile to use for this map, resulting in a random combination each time a map is generated. It then loops through each of the tiles on the map and sets them initially to the `floorTile` value generated above.

Next, the method checks to see if the tile is one of the outside edges of the map (x equal to either zero or the width of the map minus 1, or y equal to zero or the height of the map minus 1). If any of these conditions are true, the map square is automatically set to a wall tile. Similarly, the second row of tiles around the map (x or y equal to one or the width/height minus two) are skipped over for wall generation, ensuring that the map will always contain a single layer of wall tiles around the outside edge, and a clear layer of floor tiles inside that.

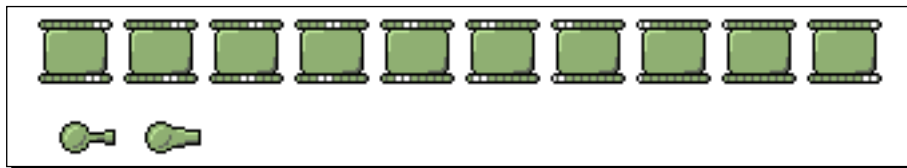
For all other squares, a random number is generated and compared to `wallChancePerSquare`. If the random value indicates that a wall should be placed at the square, the tile map is updated appropriately.

Adding the player

So we now have a working tile map, and what looks like a piece of a robotic vehicle that we can move around on the map. There are a number of issues right now, however:

- ◆ The robot we can move around simply floats around without animation or direction.
- ◆ Our floating sprite does not obey any kind of screen or world limitations. It will happily fly off into oblivion if you hold down the movement keys.
- ◆ Similarly, the sprite is not blocked by wall tiles. It will float right over them.
- ◆ Moving the sprite to the right or bottom edges of the screen doesn't scroll the camera to follow the sprite. Our camera is currently controlled by a separate set of movement keys.

In order to begin addressing these issues, we need to construct a class for the player's robo-tank. If you look at the `SpriteSheet.png` image, you will see that the player's robot is split into a couple of pieces. The treaded base that we currently have floating around on the screen will provide the base for the player's character:



On top of the base, we will place one of the available gun turrets located below the tank base on the sprite sheet. Both the base and the turret will rotate independently, allowing the user to move and fire in any combination of directions.

Building the Player class

The Player class will handle both tracking and displaying the player's robo-tank, and dealing with user input. We will start by building the basics to display the two sprites that comprise the player's tank.

Time for action – building the Player class

1. Add a new class called "Player" to the Robot Rampage project.
2. Add the following using directives to the top of the Player class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
```

3. Modify the declaration of the Player class to make it a static class:

```
static class Player
```

4. Add declarations to the Player class:

```
#region Declarations
public static Sprite BaseSprite;
public static Sprite TurretSprite;
#endregion
```

5. Add the Initialize() method to the Player class:

```
#region Initialization
public static void Initialize(
    Texture2D texture,
    Rectangle baseInitialFrame,
    int baseFrameCount,
    Rectangle turretInitialFrame,
    int turretFrameCount,
    Vector2 worldLocation)
{
    int frameWidth = baseInitialFrame.Width;
    int frameHeight = baseInitialFrame.Height;

    BaseSprite = new Sprite(
        worldLocation,
```



```
        texture,
        baseInitialFrame,
        Vector2.Zero);

BaseSprite.BoundingXPadding = 4;
BaseSprite.BoundingYPadding = 4;
BaseSprite.AnimateWhenStopped = false;
for (int x = 1; x < baseFrameCount; x++)
{
    BaseSprite.AddFrame(
        new Rectangle(
            baseInitialFrame.X + (frameHeight * x),
            baseInitialFrame.Y,
            frameWidth,
            frameHeight));
}

TurretSprite = new Sprite(
    worldLocation,
    texture,
    turretInitialFrame,
    Vector2.Zero);

TurretSprite.Animate = false;

for (int x = 1; x < turretFrameCount; x++)
{
    BaseSprite.AddFrame(
        new Rectangle(
            turretInitialFrame.X + (frameHeight * x),
            turretInitialFrame.Y,
            frameWidth,
            frameHeight));
}
}
#endregion
```

6. Add Update() and Draw() methods for the Player class:

```
#region Update and Draw
public static void Update(GameTime gameTime)
{
    BaseSprite.Update(gameTime);
    TurretSprite.WorldLocation = BaseSprite.WorldLocation;
}
```

```

public static void Draw(SpriteBatch spriteBatch)
{
    BaseSprite.Draw(spriteBatch);
    TurretSprite.Draw(spriteBatch);
}
#endregion

```

- 7.** Open the `Game1.cs` file and remove all of the existing code marked as "Temporary" code that you added while building the camera and tile map systems. This includes the declarations for the two temporary sprites and their initialization in `LoadContent()` as well as the calls to their `Update()` and `Draw()` methods in the corresponding methods of `Game1`.

- 8.** In the `LoadContent()` method of the `Game1` class, after the sprite sheet has been loaded, initialize the `Player` class:

```

Player.Initialize(
    spriteSheet,
    new Rectangle(0, 64, 32, 32),
    6,
    new Rectangle(0, 96, 32, 32),
    1,
    new Vector2(300, 300));

```

- 9.** Modify the `Update()` method of the `Game1` class to update the player sprites. The entire `Update()` method is presented here in order to verify that the temporary code from the previous steps was removed:

```

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

    Player.Update(gameTime);

    base.Update(gameTime);
}

```

- 10.** Replace the current `Draw()` method in the `Game1` class with the following method that draws both the tile map and the player sprites:

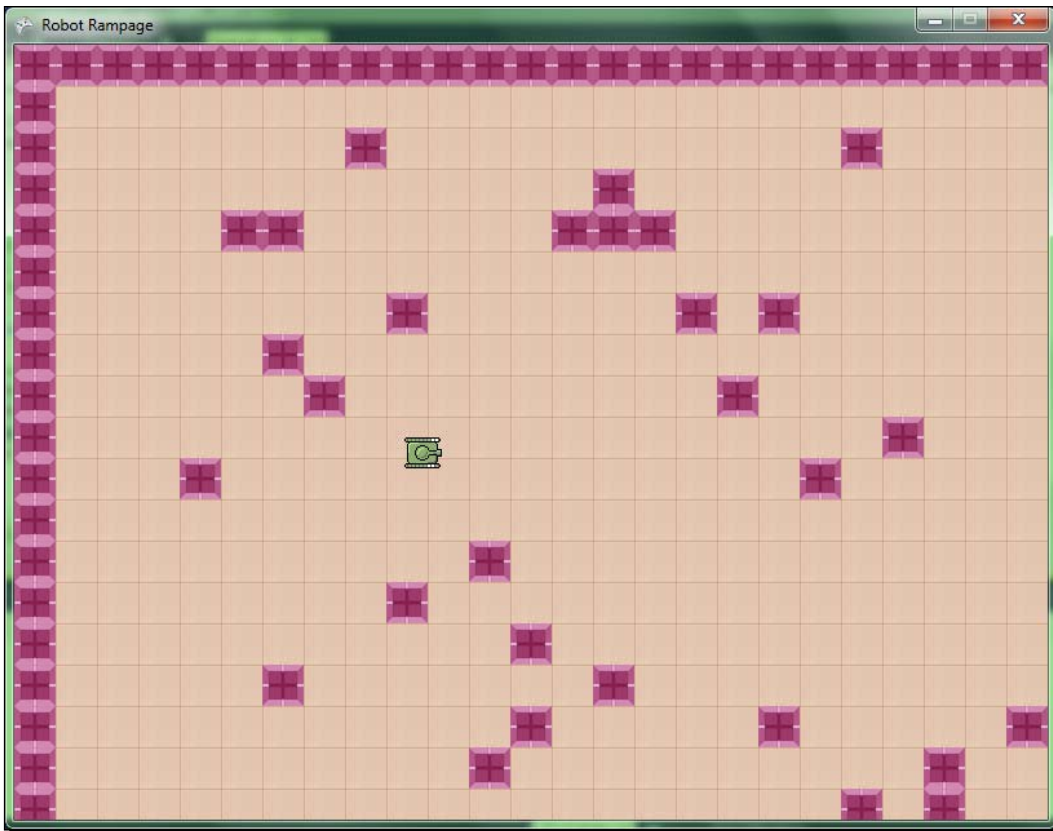
```

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

```

```
spriteBatch.Begin();  
TileMap.Draw(spriteBatch);  
Player.Draw(spriteBatch);  
spriteBatch.End();  
  
base.Draw(gameTime);  
}
```

- 11.** Execute the game to verify that the player's robo-tank is displayed on the tile map background:



What just happened?

Two individual sprites make up the player's tank. The `BaseSprite` will be drawn to the screen first, with the `TurretSprite` drawn over it. When the `Player` class constructor is called, the animation for the `BaseSprite` is created normally, and the sprite's `AnimateWhenStopped` member is set to `false`. Since the animation associated with the `BaseSprite` shows the treads of the player's tank rolling, the animation should not play when the player is not moving. In order to prevent the player from having to squeeze tightly between walls, a margin of four pixels on each side of the base sprite is established that will reduce the size of the robo-tank when we detect collisions with walls on the map.

The `TurretSprite`, on the other hand, will not animate at all. The multiple frames of the `TurretSprite` animation actually represent different turrets that we will swap for the default turret when the player has upgraded their weapons (more on this in *Chapter 7*). By setting the `Animate` member to `false`, the frame displayed by the `TurretSprite` will remain constant unless we use the `Frame` property to change it directly.

During the `Update()` method, after updating the turret sprite, we always set its location equal to the location of the base sprite. This is because we always want the turret sprite to be drawn at the same location as the base sprite. Instead of trying to synchronize their velocities to keep them in the same spot, we just force them to the same location on each frame.

We will be expanding on the `Update()` method when we add input handling, but the `Draw()` method will remain as simple as it is now—just passing the draw command along to the sprites composing the player.



Compositing sprites

Creating a game object made up of multiple sprites that are drawn on top of each other at the same location can be used to create several different effects. For example, you could overlay a "thrust" sprite on top of a space ship that only played while the player was actively moving. In a role-playing game, you could create a variety of body and armor pieces (head, wings, tails, armor of different types, weapons, and so on) and draw whichever combination represents the equipment the player's character is currently wearing.

Moving around the world

The player's movement and weapons fire are controlled separately by using the two thumbsticks on a gamepad. While the gamepad is by far the most comfortable way to control *Robot Rampage*, we will also include support for the keyboard. After all, not everyone has an Xbox controller connected to their PC.

Time for action – handling input

1. Add the following declarations to the declarations region of the Player class:

```
private static Vector2 baseAngle = Vector2.Zero;
private static Vector2 turretAngle = Vector2.Zero;
private static float playerSpeed = 90f;
```

2. Add the methods to handle keyboard and gamepad input to the Player class:

```
#region Input Handling
private static Vector2 handleKeyboardMovement (KeyboardState
keyState)
{
    Vector2 keyMovement = Vector2.Zero;
    if (keyState.IsKeyDown(Keys.W))
        keyMovement.Y--;

    if (keyState.IsKeyDown(Keys.A))
        keyMovement.X--;

    if (keyState.IsKeyDown(Keys.S))
        keyMovement.Y++;

    if (keyState.IsKeyDown(Keys.D))
        keyMovement.X++;

    return keyMovement;
}

private static Vector2 handleGamePadMovement (GamePadState
gamepadState)
{
    return new Vector2(
        gamepadState.ThumbSticks.Left.X,
        -gamepadState.ThumbSticks.Left.Y);
}

private static Vector2 handleKeyboardShots (KeyboardState keyState)
{
    Vector2 keyShots = Vector2.Zero;

    if (keyState.IsKeyDown(Keys.NumPad1))
        keyShots = new Vector2(-1, 1);

    if (keyState.IsKeyDown(Keys.NumPad2))
```

```
        keyShots = new Vector2(0, 1);

        if (keyState.IsKeyDown(Keys.NumPad3))
            keyShots = new Vector2(1, 1);

        if (keyState.IsKeyDown(Keys.NumPad4))
            keyShots = new Vector2(-1, 0);

        if (keyState.IsKeyDown(Keys.NumPad6))
            keyShots = new Vector2(1, 0);

        if (keyState.IsKeyDown(Keys.NumPad7))
            keyShots = new Vector2(-1, -1);

        if (keyState.IsKeyDown(Keys.NumPad8))
            keyShots = new Vector2(0, -1);

        if (keyState.IsKeyDown(Keys.NumPad9))
            keyShots = new Vector2(1, -1);

        return keyShots;
    }

    private static Vector2 handleGamePadShots(GamePadState
gamepadState)
    {
        return new Vector2(
            gamepadState.ThumbSticks.Right.X,
            -gamepadState.ThumbSticks.Right.Y);
    }

    private static void handleInput(GameTime gameTime)
    {
        float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

        Vector2 moveAngle = Vector2.Zero;
        Vector2 fireAngle = Vector2.Zero;

        moveAngle += handleKeyboardMovement(Keyboard.GetState());
        moveAngle +=
            handleGamePadMovement(GamePad.GetState(PlayerIndex.One));

        fireAngle += handleKeyboardShots(Keyboard.GetState());
        fireAngle +=
```

```
        handleGamePadShots (GamePad.GetState (PlayerIndex.One)) ;

    if (moveAngle != Vector2.Zero)
    {
        moveAngle.Normalize();
        baseAngle = moveAngle;
    }

    if (fireAngle != Vector2.Zero)
    {
        fireAngle.Normalize();
        turretAngle = fireAngle;
    }

    BaseSprite.RotateTo(baseAngle);
    TurretSprite.RotateTo(turretAngle);

    BaseSprite.Velocity = moveAngle * playerSpeed;
}
#endregion
```

- 3.** Modify the `Update()` method of the `Player` class to call the `handleInput()` method before updating the sprite. The entire `Update()` method should look like:

```
public static void Update(GameTime gameTime)
{
    handleInput(gameTime);
    BaseSprite.Update(gameTime);
    TurretSprite.WorldLocation = BaseSprite.WorldLocation;
}
```

- 4.** Launch your game and drive around! Swing your cannon around with the right thumbstick or the numeric keypad. Make sure *Numlock* is turned on!

What just happened?

The whole purpose of the input handling methods is to determine the value of the two vectors declared in step one. The `baseAngle` vector determines the direction that the player is moving in (and therefore the orientation of the base sprite), while the `turretAngle` vector determines the direction the player's cannon will face.

The gamepad controls are far simpler than the keyboard controls because the gamepad's thumbsticks already return a complete vector. The only alteration we need to make is to reverse the sign on the Y component of the vector, since negative Y values correspond to the up direction on the screen, but to the down direction on the gamepad.

In the case of the keyboard, pressing the individual movement keys increments or decrements the appropriate component of the vector that will be returned, while pressing the firing keys on the numeric keypad sets the returned vector to a vector pointing in the direction of the firing key relative to the five key (in other words, pressing eight on the keypad fires upwards, while pressing three fires down and to the right).

In the `handleInput()` method, we initially set both the `moveAngle` (which determines the direction the player will move in) and the `fireAngle` (which determines the direction the turret will be facing) to `Vector2.Zero`, indicating that they are both empty. We need to keep these two vectors separate from their related vectors (`baseAngle` and `turretAngle`) because we do not want to modify the angle that either the tank or the turret is facing if the player is not moving the corresponding thumbstick. If we did allow them to reset back to `Vector2.Zero` in every frame, any time the player was not pressing one of the control sticks, the corresponding part (tank or turret) would snap back to face the right edge of the screen.

By calling the `Handle...()` methods and adding the results to the `moveAngle` and `fireAngle` vectors, we now have the two vectors we need to animate the player's tank. We check to see if the local vectors have a value (in other words, are not equal to `Vector2.Zero`). If so, the vector is normalized and assigned to the corresponding class-level member variable (`moveAngle` to `baseAngle`, and `fireAngle` to `turretAngle`).

The sprites are then rotated to their appropriate angles, and the `moveAngle` is applied to the base tank sprite's velocity.

The `Update()` method calls our new `handleInput()` method and then updates the base sprite. The position of the base sprite is then copied to the location of the `turretSprite`, keeping the two sprites in sync with each other.

Staying in bounds

As the game stands, you can drive your tank around the game screen easily. Too easily, in fact! You can run straight through the walls and off the screen!

We need to ensure that the player's tank cannot move outside of the game world and that the game's camera follows the player as they move near the screen's edges.

Time for action – staying in bounds

1. Create a region called "Movement Limitations" in the Player class:

```
#region Movement Limitations
#endregion
```

2. Inside the Movement Limitations region, add the `clampToWorld()` method:

```
private static void clampToWorld()
{
    float currentX = BaseSprite.WorldLocation.X;
    float currentY = BaseSprite.WorldLocation.Y;

    currentX = MathHelper.Clamp(
        currentX,
        0,
        Camera.WorldRectangle.Right - BaseSprite.FrameWidth);

    currentY = MathHelper.Clamp(
        currentY,
        0,
        Camera.WorldRectangle.Bottom - BaseSprite.FrameHeight);

    BaseSprite.WorldLocation = new Vector2(currentX, currentY);
}
```

3. Add a declaration to the Player class to define the area in which the camera should attempt to keep the player:

```
private static Rectangle scrollArea =
    new Rectangle(150, 100, 500, 400);
```

4. Add the `repositionCamera()` helper method to the Movement Limitations region of the Player class:

```
private static void repositionCamera(
    GameTime gameTime,
    Vector2 moveAngle)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    float moveScale = playerSpeed * elapsed;

    if ((BaseSprite.ScreenRectangle.X < scrollArea.X) &&
        (moveAngle.X < 0))
    {
        Camera.Move(new Vector2(moveAngle.X, 0) * moveScale);
    }

    if ((BaseSprite.ScreenRectangle.Right > scrollArea.Right) &&
        (moveAngle.X > 0))
```

```

    {
        Camera.Move(new Vector2(moveAngle.X, 0) * moveScale);
    }

    if ((BaseSprite.ScreenRectangle.Y < scrollArea.Y) &&
        (moveAngle.Y < 0))
    {
        Camera.Move(new Vector2(0, moveAngle.Y) * moveScale);
    }

    if ((BaseSprite.ScreenRectangle.Bottom > scrollArea.Bottom) &&
        (moveAngle.Y > 0))
    {
        Camera.Move(new Vector2(0, moveAngle.Y) * moveScale);
    }
}

```

5. Modify the `Update()` method to call `clampToWorld()` after the base sprite has been updated. The full `Update()` method should read:

```

public static void Update(GameTime gameTime)
{
    handleInput(gameTime);
    BaseSprite.Update(gameTime);
    clampToWorld();
    TurretSprite.WorldLocation = BaseSprite.WorldLocation;
}

```

6. Still in the `handleInput()` method, add the following as the last line of the method:


```
repositionCamera(gameTime, moveAngle);
```
7. Launch the game and drive around again. Your tank will stay confined to the game world, but is still able to drive through walls.

What just happened?

Things are looking a little better now. The player can no longer drive off the edge of the world, and when they reach the bottom or right-hand side, the camera will scroll with them. We can still drive through walls, but we are getting there!

When we apply movement to the player, we call `clampToWorld()`, which separates the player's location into X and Y components. The `MathHelper.Clamp()` method is then used to ensure that the components stay within the world's coordinate system. The width and height of the sprite is subtracted from the width and height of the game world so that the furthest the player's sprite can get to the right-side and bottom of the world is a full sprite's size away. This will keep the sprite fully within the game world at all times.

When the player approaches an edge of the screen, the camera needs to be adjusted to move in the same direction that the player is moving, assuming that the end of the game world has not yet been reached. By defining the `scrollArea` rectangle, we are specifying that we would like to keep the player in a 500 by 400 pixel area beginning at (150, 100). This corresponds to the center of our 800 by 600 screen, with a 150 pixel buffer on the left and right edges, and a 100 pixel buffer along the top and bottom.

When `repositionCamera()` is called, it checks to see if the player has moved out of the defined scrolling area. Each side of the scroll area is checked individually, starting with the left edge. If the X coordinate of the `ScreenRectangle` is less than the X coordinate of the `scrollArea` and the player is moving left, the camera is moved left by an amount equal to the amount that the sprite itself will be moved by multiplying the appropriate component of `moveAngle` by the `moveScale` value.

The result is that the camera will move the same distance that the player moves. The player's sprite will appear to stay in place while the game world scrolls around it. Since the Camera class limits its own position to ensure that a full view of the world is always displayed, the camera will simply stop moving when the player is close enough to the edge that it can no longer scroll.

Running into tiles

The last limitation we need to account for in player movement is the underlying tile-based map. We want the game to detect when the player is attempting to move into a wall and stop them from doing so.

Time for action – accounting for walls

1. Add the `checkTileObstacles()` method to the Movement Limitations region of the Player class:

```
private static Vector2 checkTileObstacles(
    float elapsedTime,
    Vector2 moveAngle)
{
    Vector2 newHorizontalLocation = BaseSprite.WorldLocation +
        (new Vector2(moveAngle.X, 0) * (playerSpeed *
            elapsedTime));

    Vector2 newVerticalLocation = BaseSprite.WorldLocation +
        (new Vector2(0, moveAngle.Y) * (playerSpeed *
            elapsedTime));

    Rectangle newHorizontalRect = new Rectangle(
        (int)newHorizontalLocation.X,
        (int)BaseSprite.WorldLocation.Y,
```

```
        BaseSprite.FrameWidth,
        BaseSprite.FrameHeight);

Rectangle newVerticalRect = new Rectangle(
    (int)BaseSprite.WorldLocation.X,
    (int)newVerticalLocation.Y,
    BaseSprite.FrameWidth,
    BaseSprite.FrameHeight);

int horizLeftPixel = 0;
int horizRightPixel = 0;

int vertTopPixel = 0;
int vertBottomPixel = 0;

if (moveAngle.X < 0)
{
    horizLeftPixel = (int)newHorizontalRect.Left;
    horizRightPixel = (int)BaseSprite.WorldRectangle.Left;
}

if (moveAngle.X > 0)
{
    horizLeftPixel = (int)BaseSprite.WorldRectangle.Right;
    horizRightPixel = (int)newHorizontalRect.Right;
}

if (moveAngle.Y < 0)
{
    vertTopPixel = (int)newVerticalRect.Top;
    vertBottomPixel = (int)BaseSprite.WorldRectangle.Top;
}

if (moveAngle.Y > 0)
{
    vertTopPixel = (int)BaseSprite.WorldRectangle.Bottom;
    vertBottomPixel = (int)newVerticalRect.Bottom;
}

if (moveAngle.X != 0)
{
    for (int x = horizLeftPixel; x < horizRightPixel; x++)
    {
        for (int y = 0; y < BaseSprite.FrameHeight; y++)
        {
            if (TileMap.IsWallTileByPixel(
                new Vector2(x, newHorizontalLocation.Y + y)))
            {
                moveAngle.X = 0;
                break;
            }
        }
    }
}
```

```
        }
    }
    if (moveAngle.X == 0)
    {
        break;
    }
}

if (moveAngle.Y != 0)
{
    for (int y = vertTopPixel; y < vertBottomPixel; y++)
    {
        for (int x = 0; x < BaseSprite.FrameWidth; x++)
        {
            if (TileMap.IsWallTileByPixel(
                new Vector2(newVerticalLocation.X + x, y)))
            {
                moveAngle.Y = 0;
                break;
            }
        }
        if (moveAngle.Y == 0)
        {
            break;
        }
    }

    return moveAngle;
}
```

- 2.** Return to the `handleInput()` method and once again update the `if (moveAngle != Vector2.Zero)` statement by adding a call to `checkTileObstacles()`. The whole statement should now read:

```
if (moveAngle != Vector2.Zero)
{
    moveAngle.Normalize();
    baseAngle = moveAngle;
    moveAngle = checkTileObstacles(elapsed, moveAngle);
}
```

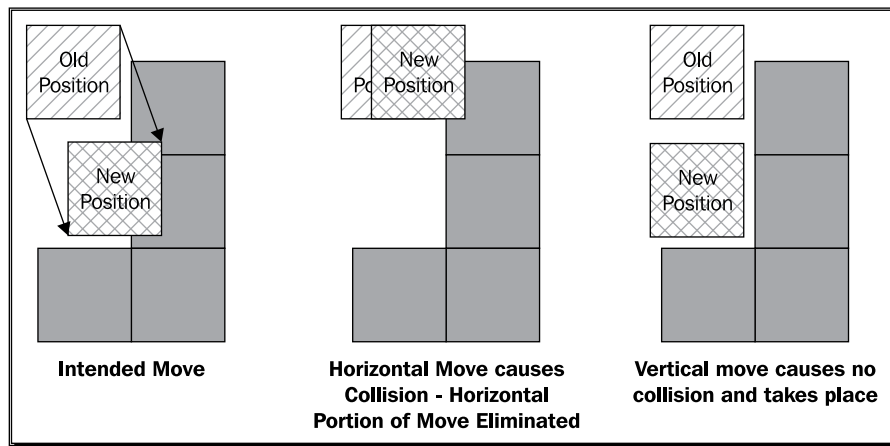
- 3.** Execute the game again and drive around.

What just happened?

In order to check for tile-based collisions, we break the process into two steps. First we will check for collisions due to horizontal movement, and then due to vertical movement.

To facilitate this, we begin by establishing two new vectors representing the new locations of the sprite, if only the horizontal or vertical movement was applied.

From these new vectors, we create two rectangles, representing the world location of the sprite's two possible movements. In other words, `newHorizontalRect` specifies where the player's sprite would be located after moving, if only the horizontal component of the movement is considered, while `newVerticalRect` represents the new location, if movement occurred only along the vertical axis:



Next, we need to determine a range of pixels to check for tile obstacles. We *could* simply check all of the pixels inside the two new rectangles, calling `TileMap.IsWallTile()` for each pixel in both rectangles. This would return the results we are looking for, but it would cause unnecessary processing, as only the portions of the new rectangles that are not already covered by the current position's rectangle need to be checked for collisions.

Instead, we can use the direction in which we are moving to determine a smaller range of pixels that need to be checked. In the horizontal movement portion of the previous diagram, the new position rectangle mostly overlaps the old position rectangle. We only need to check the pixels between the right edge of the old position and the right edge of the new position, because the old position is already known to be an area the sprite can exist in.

If we are moving to the right, we use the right edges of the old and new positions to define the limits of the area we will check. Similarly, if we are moving left, we use the left edges of the two position rectangles to define the area.

We can then check each pixel in the new area against `TileMap.IsWallTile()` to determine if moving in this direction would cause the sprite to overlap a wall tile. If any of them do, we set the X component of the `moveAngle` vector to zero, eliminating the movement in that direction. As soon as any pixel tests true, we break out of the loop, since there is no need to continue testing.

After we have dealt with horizontal movement, the process is repeated for vertical movement, eliminating the Y component of the `moveAngle` vector if the new Y position would result in a wall collision.

Summary

We have covered a lot of ground building the foundation for Robot Rampage. So far, our new game:

- ◆ Contains a camera to view a game world larger than the display screen
- ◆ Allows us to define sprites that are aware of their positions within the game world instead of simply their location on the screen
- ◆ Generates a tile-based map for the game world with randomly placed walls
- ◆ Allows the player to drive their tank around the tile map, including collision detection with map walls and automatic camera panning

In *Chapter 7*, we will finish building Robot Rampage by adding enemy robots and the ability to fire weapons. We will also cover basic path finding to allow the enemy to hunt down the player.

7

Robot Rampage – Lots and Lots of Bullets



Robot Rampage is already looking pretty good! We have a tile-based game world and a mobile player tank with the ability to rotate its cannon independently of the tank's base. But it is not quite finished yet.



In this chapter, we will complete the build by looking at the following topics:

- ◆ Adding an updated particle-based visual effects system
- ◆ Adding weapons to the player's arsenal
- ◆ Adding enemy robo-tanks to the map
- ◆ Building a path-finding system to allow the enemy to hunt down the player
- ◆ Implementing computer terminals the player can shut down
- ◆ Wrapping the game in a game flow structure

Visual effects

The particle explosion system we built for Asteroid Belt Assault can be adapted for use in Robot Rampage, but we will expand on it a bit and make it more flexible. Additionally, we want to add a new type of special effect that will be used when shots impact walls to throw off a small shower of sparks. We do not want a full blown explosion every time a shot hits a wall, but we also would like to see some effect instead of them simply disappearing.

Revisiting particles

In Asteroid Belt Assault, we utilized the Particle class to represent only sprites that we did not involve in collision detection. For Robot Rampage, we will also be implementing a Particle class, but this time around we will use it for both special effects and for displaying the projectiles fired by both the player and the enemy robots.

Time for action – the Particle class

1. Add a new class file to the Robot Rampage project called `Particle.cs`.
2. Add the following using directives to the Particle class:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Update the class declaration for the Particle class to derive it from the Sprite class:

```
class Particle : Sprite
```

4. Add declarations to the Particle class:

```
#region Declarations  
private Vector2 acceleration;  
private float maxSpeed;  
private int initialDuration;  
private int remainingDuration;  
private Color initialColor;  
private Color finalColor;  
#endregion
```

5. Add properties to the Particle class:

```
#region Properties  
public int ElapsedDuration  
{  
    get  
    {  
        return initialDuration - remainingDuration;  
    }  
}  
  
public float DurationProgress  
{  
    get  
    {
```

```

        return (float)ElapsedDuration /
            (float)initialDuration;
    }
}

public bool IsActive
{
    get
    {
        return (remainingDuration > 0);
    }
}
#endregion

```

6. Add a constructor to the Particle class:

```

#region Constructor
public Particle(
    Vector2 location,
    Texture2D texture,
    Rectangle initialFrame,
    Vector2 velocity,
    Vector2 acceleration,
    float maxSpeed,
    int duration,
    Color initialColor,
    Color finalColor)
    : base(location, texture, initialFrame, velocity)
{
    initialDuration = duration;
    remainingDuration = duration;
    this.acceleration = acceleration;
    this.initialColor = initialColor;
    this.maxSpeed = maxSpeed;
    this.finalColor = finalColor;
}
#endregion

```

7. Add the Update() and Draw() methods to the Particle class:

```

#region Update and Draw
public override void Update(GameTime gameTime)
{
    if (remainingDuration <= 0)

```

```
        {
            Expired = true;
        }
        if (!Expired)
        {
            Velocity += acceleration;
            if (Velocity.Length() > maxSpeed)
            {
                Vector2 vel = Velocity;
                vel.Normalize();
                Velocity = vel * maxSpeed;
            }
            TintColor = Color.Lerp(
                initialColor,
                finalColor,
                DurationProgress);
            remainingDuration--;
        }
        base.Update(gameTime);
    }

    public override void Draw(SpriteBatch spriteBatch)
    {
        if (IsActive)
        {
            base.Draw(spriteBatch);
        }
    }
}
#endregion
```

What just happened?

Just as we did in Asteroid Belt Assault, we have constructed our Particle class as an extension of the Sprite class. Just as before, we will use the Particle class to generate large volumes of sprites that will need to automatically expire after a given period of time.

The EffectsManager class

In Asteroid Belt Assault, we used a class called ExplosionManager to handle all of the game's explosive effects. We will create a similar class for Robot Rampage, but call it EffectsManager since it will handle our sparks effect as well as explosions.

Time for action – the EffectsManager class

1. Add a new class called EffectsManager to the Robot Rampage project.
2. Add the following using directives to the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Modify the declaration of the EffectsManager class to make it static:

```
static class EffectsManager
```

4. Add declarations to the EffectsManager class:

```
#region Declarations
static public List<Particle> Effects = new List<Particle>();
static Random rand = new Random();
static public Texture2D Texture;
static public Rectangle ParticleFrame = new Rectangle(0, 288,
    2, 2);
static public List<Rectangle> ExplosionFrames =
    new List<Rectangle>();
#endregion
```

5. Add an Initialize() method to the EffectsManager class:

```
#region Initialization
public static void Initialize(
    Texture2D texture,
    Rectangle particleFrame,
    Rectangle explosionFrame,
    int explosionFrameCount)
{
    Texture = texture;
    ParticleFrame = particleFrame;
    ExplosionFrames.Clear();
    ExplosionFrames.Add(explosionFrame);
    for (int x = 1; x < explosionFrameCount; x++)
    {
```

```
        explosionFrame.Offset(explosionFrame.Width, 0);
        ExplosionFrames.Add(explosionFrame);
    }
}
#endregion
```

6. Add the `RandomDirection()` method to the `EffectsManager` class:

```
#region Helper Methods
public static Vector2 randomDirection(float scale)
{
    Vector2 direction;
    do
    {
        direction = new Vector2(
            rand.Next(0, 100) - 50,
            rand.Next(0, 100) - 50);
    } while (direction.Length() == 0);
    direction.Normalize();
    direction *= scale;
    return direction;
}
#endregion
```

7. Add the `Update()` and `Draw()` methods to the `EffectsManager` class:

```
#region Public Methods
static public void Update(GameTime gameTime)
{
    for (int x = Effects.Count - 1; x >= 0; x--)
    {
        Effects[x].Update(gameTime);
        if (Effects[x].Expired)
        {
            Effects.RemoveAt(x);
        }
    }
}

static public void Draw(SpriteBatch spriteBatch)
{
    foreach (Sprite sprite in Effects)
    {
```

```

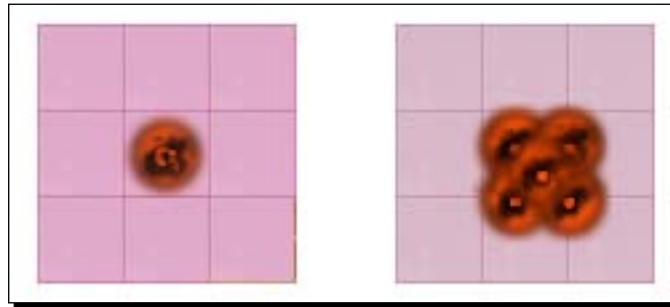
        sprite.Draw(spriteBatch);
    }
}
#endregion

```

What just happened?

Up to this point, the `EffectsManager` class is very similar to the `ExplosionManager` from *Asteroid Belt Assault*, though some refinements have been made.

We do not have nearly as many declarations in the `EffectsManager` class because the default values for things like the number of particles and explosion pieces generated, and the transition colors have been removed from the manager. These values will now be specifiable when the explosion is created:



The explosions our `EffectsManager` will create will use the same images (scaled to fit our 32 by 32 pixel tile size) from *Asteroid Belt Assault*. We will produce two types of explosions. The smaller explosion will be used whenever an enemy robot is destroyed, while the larger will be used by the upgraded rocket weapon the player will be able to acquire.

Since the class is static, the initialization that was done in the constructor of the `ExplosionManager` class is now separated into an `Initialize()` method.

Time for action – building explosions

1. Add the `AddExplosion()` method to the "Public Methods" region of the `EffectsManager` class:

```

public static void AddExplosion(
    Vector2 location,
    Vector2 momentum,
    int minPointCount,
    int maxPointCount,

```

```
        int minPieceCount,
        int maxPieceCount,
        float pieceSpeedScale,
        int duration,
        Color initialColor,
        Color finalColor)
    {
        float explosionMaxSpeed = 30f;
        int pointSpeedMin = (int)pieceSpeedScale * 2;
        int pointSpeedMax = (int)pieceSpeedScale * 3;

        Vector2 pieceLocation = location -
            new Vector2(ExplosionFrames[0].Width / 2,
                ExplosionFrames[0].Height / 2);

        int pieces = rand.Next(minPieceCount, maxPieceCount + 1);
        for (int x = 0; x < pieces; x++)
        {
            Effects.Add(new Particle(
                pieceLocation,
                Texture,
                ExplosionFrames[rand.Next(0, ExplosionFrames.Count)],
                randomDirection(pieceSpeedScale) + momentum,
                Vector2.Zero,
                explosionMaxSpeed,
                duration,
                initialColor,
                finalColor));
        }

        int points = rand.Next(minPointCount, maxPointCount + 1);
        for (int x = 0; x < points; x++)
        {
            Effects.Add(new Particle(
                location,
                Texture,
                ParticleFrame,
                randomDirection((float)rand.Next(
                    pointSpeedMin, pointSpeedMax)) + momentum,
                Vector2.Zero,
                explosionMaxSpeed,
                duration,
                initialColor,
```

```

        finalColor));
    }
}

```

2. Add an overload of the `AddExplosion()` method to create a "default" explosion:

```

public static void AddExplosion(Vector2 location, Vector2
momentum)
{
    AddExplosion(
        location,
        momentum,
        15,
        20,
        2,
        4,
        6.0f,
        90,
        new Color(1.0f, 0.3f, 0f, 0.5f),
        new Color(1.0f, 0.3f, 0f, 0f));
}

```

3. Add the `AddLargeExplosion()` method to the `EffectsManagerClass`:

```

public static void AddLargeExplosion(Vector2 location)
{
    AddExplosion(
        location,
        Vector2.Zero,
        15,
        20,
        4,
        6,
        30f,
        90,
        new Color(1.0f, 0.3f, 0f, 0.5f),
        new Color(1.0f, 0.3f, 0f, 0f));
}

```


What just happened?

Most of the `AddExplosion()` method is unchanged from Asteroid Belt Assault, though many of the values that were class-level variables are now parameters passed to the method. Just as before, a random number of larger pieces are generated and added to the Effects list, and then a random number of point-sprite particles are generated with slightly faster speeds than the larger pieces.

By adding an overload of `AddExplosion()`, we can avoid having to specify most of these values each time we want to generate a typical explosion. We simply pass in default values for all of the parameters matching what used to be the defaults in Asteroid Belt Assault.

Finally, we have the `AddLargeExplosion()` method, which just varies the parameters passed to `AddExplosion()` to create an explosion with faster moving pieces that will cover more area before they fade. We will use this explosion when we create a rocket launcher weapon by stacking a few of these larger explosions slightly offset from each other when the rocket explodes.

Time for action – spark effects

1. Add the `AddSparkEffect()` method to the "Public Methods" region of the `EffectsManager` class:

```
public static void AddSparksEffect(
    Vector2 location,
    Vector2 impactVelocity)
{
    int particleCount = rand.Next(10, 20);
    for (int x = 0; x < particleCount; x++)
    {
        Particle particle = new Particle(
            location - (impactVelocity / 60),
            Texture,
            ParticleFrame,
            randomDirection((float)rand.Next(10, 20)),
            Vector2.Zero,
            60,
            20,
            Color.Yellow,
            Color.Orange);
        Effects.Add(particle);
    }
}
```

2. In the `LoadContent()` method of the `Game1` class, initialize the `EffectsManager` after the `Player` class has been initialized:

```
EffectsManager.Initialize(  
    spriteSheet,  
    new Rectangle(0, 288, 2, 2),  
    new Rectangle(0, 256, 32, 32),  
    3);
```

3. In the `Update()` method of the `Game1` class, update the `EffectsManager` after the player has been updated:

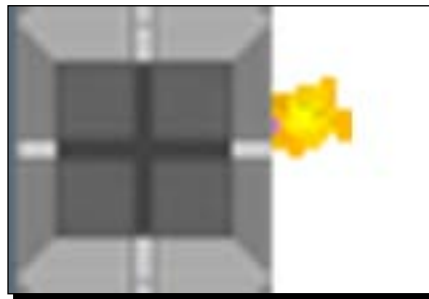
```
EffectsManager.Update(gameTime);
```

4. In the `Draw()` method of the `Game1` class, draw the `EffectsManager` after the player has been drawn:

```
EffectsManager.Draw(spriteBatch);
```

What just happened?

When a spark effect is added, a random number of particles are created. In order to place the explosion right before the wall instead of embedded into it, the velocity passed to the method is divided by 60 and subtracted from the location. Since velocities are stored on the sprites in pixels per second, dividing by 60 represents the approximate distance the sprite would travel in one frame (at 60 frames per second). Effectively, the spark effect will be generated at the position of the shot on the previous frame:



The effect itself is just a random number (ten to nineteen) of point-sprite particles that start off yellow and progress to orange over the course of 20 frames. This will result in a small, generally circular burst of sparks around the impact point.

Adding weaponry

Now that we can create special effects for our weaponry, we need to actually add the weapons to the game! We will begin with a basic cannon, and then modify the Player class to support upgraded weaponry.

The WeaponManager class

Once again, we will use a "manager" class to handle the shots fired in the game. This time though, we will be dealing with player fired shots and enemy fired shots separately because players will have more options on exactly *what* they can shoot than the enemy tanks will.

Time for action – beginning the WeaponManager class

1. Add a new class called WeaponManager to the Robot Rampage project.
2. Add the following using directives to the top of the WeaponManager class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Modify the declaration of the WeaponManager class to make it a static class:

```
static class WeaponManager
```

4. Add declarations to the WeaponManager class:

```
#region declarations
static public List<Particle> Shots = new List<Particle>();
static public Texture2D Texture;
static public Rectangle shotRectangle =
    new Rectangle(0, 128, 32, 32);
static public float WeaponSpeed = 600f;

static private float shotTimer = 0f;
static private float shotMinTimer = 0.15f;
#endregion
```

5. Add properties to the WeaponManager class:

```
#region Properties
static public float WeaponFireDelay
{
    get
    {
        return shotMinTimer;
    }
}
```

```

    }
}

static public bool CanFireWeapon
{
    get
    {
        return (shotTimer >= WeaponFireDelay);
    }
}
#endregion

```

6. Add the AddShot() method to the WeaponManager class:

```

#region Effects Management Methods

private static void AddShot(
    Vector2 location,
    Vector2 velocity,
    int frame)
{
    Particle shot = new Particle(
        location,
        Texture,
        shotRectangle,
        velocity,
        Vector2.Zero,
        400f,
        120,
        Color.White,
        Color.White);

    shot.AddFrame(new Rectangle(
        shotRectangle.X + shotRectangle.Width,
        shotRectangle.Y,
        shotRectangle.Width,
        shotRectangle.Height));

    shot.Animate = false;
    shot.Frame = frame;
    shot.RotateTo(velocity);
    Shots.Add(shot);
}
#endregion

```

7. Add the `FireWeapon()` method to the `WeaponManager` class:

```
#region Weapons Management Methods
public static void FireWeapon(Vector2 location, Vector2 velocity)
{
    AddShot(location, velocity, 0);
    shotTimer = 0.0f;
}
#endregion
```

8. Add `Update()` and `Draw()` methods to the `WeaponManager` class:

```
#region Update and Draw
static public void Update(GameTime gameTime)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    shotTimer += elapsed;

    for (int x = Shots.Count - 1; x >= 0; x--)
    {
        Shots[x].Update(gameTime);

        if (Shots[x].Expired)
        {
            Shots.RemoveAt(x);
        }
    }
}

static public void Draw(SpriteBatch spriteBatch)
{
    foreach (Particle sprite in Shots)
    {
        sprite.Draw(spriteBatch);
    }
}
#endregion
```

9. In the `Player` class, in the `HandleInput()` method, update the `if` statement that reads `if (fireAngle != Vector2.Zero)` and replace it with the following:

```
if (fireAngle != Vector2.Zero)
{
    fireAngle.Normalize();
    turretAngle = fireAngle;
}
```

```
        if (WeaponManager.CanFireWeapon)
        {
            WeaponManager.FireWeapon(
                TurretSprite.WorldLocation,
                fireAngle * WeaponManager.WeaponSpeed);
        }
    }
```

- 10.** In the `LoadContent()` method of the `Game1` class, add the following line to initialize the `WeaponManager`'s `Texture` variable after the `Sprite Sheet` has been loaded:

```
WeaponManager.Texture = spriteSheet;
```

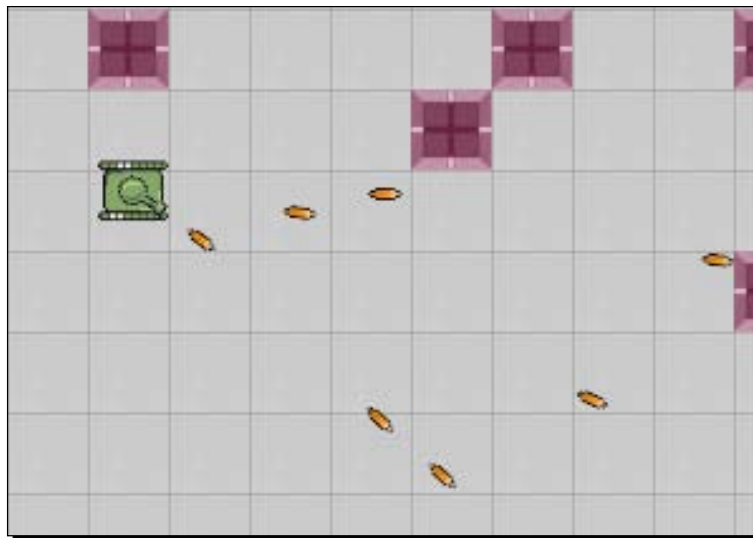
- 11.** In the `Game1` class, add a line to the `Update()` method to update the `WeaponManager` class right after the `Player` class is updated:

```
WeaponManager.Update(gameTime);
```

- 12.** Still in the `Game1` class, add a line to the `Draw()` method to render the `WeaponManager` class. This time, place the call right before the `Player` class is drawn:

```
WeaponManager.Draw(spriteBatch);
```

- 13.** Launch `Robot Rampage` and drive your tank around. When using the right thumbstick or the numeric keypad, you can now launch volleys of projectiles:



What just happened?

Each player shot is stored as a `Particle` in the `Shots` list. We are using `Particles` instead of `Sprites` so that we can take advantage of the automatic expiration of a `Particle` after a given time frame. That way, if a shot is fired into a wide open area and travels off the screen, it will simply clean itself up.

The `WeaponSpeed` float will be used to scale the velocity of the shots the player fires, and defaults to 600 pixels per second. The shots on the screen will be moving fairly quickly!

The `shotTimer` and `shotMinTimer` member variables, along with the `WeaponFireDelay` and `CanFireWeapon` properties work in conjunction to implement the timing mechanism we have used previously to control the rate at which new shots are added to the game.

When the time comes to actually add the shot, the `AddShot()` method creates the new `Particle` object. The sprite sheet we are using for the game has two different weapon graphics: one for a normal shot, and one for a missile. These two frames are added to the particle, and it is set not to animate between them. When `AddShot()` is passed a zero for the frame parameter, a normal shot will be fired. When it is passed a one, the second frame will be used, resulting in a missile being fired. We will implement weapon upgrades shortly—right now the player can only fire normal shots.

Currently, the `FireWeapon()` method simply calls `AddShot()` and resets the `shotTimer` variable. We will be expanding upon this method when we add multiple weapon types.

Finally, the `Update()` and `Draw()` methods contain our standard code to loop through the objects in the `Shots` list and handle them.

The only updates we need to make to the player class to allow the player to fire weapons are to actually call the `FireWeapon()` method, assuming that `CanFireWeapon` returns true. `FireWeapon()` is passed the current position of the player's turret, and the angle that the turret is pointing at multiplied by the weapon speed.

Weapon upgrades

In addition to the normal fire mode of the player's cannon, we are going to add two more weapon types: a triple cannon, which fires three shots at a spread angle and a rocket launcher, which destroys all enemies in a small area surrounding its point of impact.

Time for action – new weapons

1. Add the following declarations to the `WeaponManager` class:

```
static private float rocketMinTimer = 0.5f;

public enum WeaponType { Normal, Triple, Rocket };
static public WeaponType CurrentWeaponType = WeaponType.Triple;
static public float WeaponTimeRemaining = 30.0f;
static private float weaponTimeDefault = 30.0f;
static private float tripleWeaponSplitAngle = 15;
```

2. Replace the current `WeaponFireDelay` property with the following:

```
static public float WeaponFireDelay
{
    get
    {
        if (CurrentWeaponType==WeaponType.Rocket)
        {
            return rocketMinTimer;
        }
        else
        {
            return shotMinTimer;
        }
    }
}
```

3. Replace the current `FireWeapon()` method with the following:

```
public static void FireWeapon(Vector2 location, Vector2 velocity)
{
    switch (CurrentWeaponType)
    {
        case WeaponType.Normal:
            AddShot(location, velocity, 0);
            break;

        case WeaponType.Triple:
            AddShot(location, velocity, 0);

            float baseAngle = (float)Math.Atan2(
                velocity.Y,
                velocity.X);
```



```
        float offset = MathHelper.ToRadians(
            tripleWeaponSplitAngle);

        AddShot(
            location,
            new Vector2(
                (float)Math.Cos(baseAngle - offset),
                (float)Math.Sin(baseAngle - offset)
            ) * velocity.Length(),
            0);

        AddShot(
            location,
            new Vector2(
                (float)Math.Cos(baseAngle + offset),
                (float)Math.Sin(baseAngle + offset)
            ) * velocity.Length(),
            0);
        break;

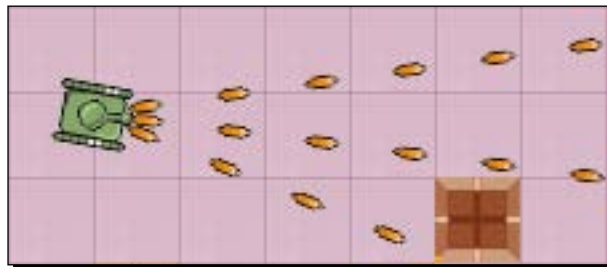
    case WeaponType.Rocket:
        AddShot(location, velocity, 1);
        break;
    }

    shotTimer = 0.0f;
}
```

4. Add the `checkWeaponUpgradExpire()` method to the **Weapon Management Methods** region of the `WeaponManager` class:

```
private static void checkWeaponUpgradExpire(float elapsed)
{
    if (CurrentWeaponType != WeaponType.Normal)
    {
        WeaponTimeRemaining -= elapsed;
        if (WeaponTimeRemaining <= 0)
        {
            CurrentWeaponType = WeaponType.Normal;
        }
    }
}
```

5. Modify the `Update()` method of the `WeaponManager` class to call `checkWeaponUpgradeExpire()` right after the `shotTimer` variable is updated:
`checkWeaponUpgradeExpire(elapsed);`
6. Launch your game and fire your weapons. For thirty seconds after starting the game, you will have the triple cannon weapon, firing three shots at a time:



7. Stop the game and modify the `CurrentWeaponType` variable to change its default value to `WeaponType.Rocket`.
8. Launch the game again, and shoot rockets from your cannon:



9. Stop the game and modify the `CurrentWeaponType` variable to change its default value to `WeaponType.Normal` and set the `WeaponTimeRemaining` value to `0.0f`;

What just happened?

Rockets fire a bit slower than normal shots do, so the `WeaponFireDelay` property returns the delay value appropriate to the currently active weapon.

The `FireShot()` method treats normal shots and rockets in almost the exact same manner, the only difference being that normal shots are fired with a frame number of zero, while rockets receive a frame number of one.

For the triple cannon, we calculate an offset from the current firing angle equal to the value of `tripleWeaponSplitAngle` (which we default to 15 degrees), and after firing the base shot, simply fire two more shots, one adding the offset angle and one subtracting it from the base angle value.

Whenever the player picks up a new weapon, it will last for `weaponTimeDefault` seconds, before their weapon reverts back to the `WeaponType.Normal`. The `checkWeaponUpgradeExpire()` method enforces this rule.

Shot to map collisions

While we can let loose a barrage of weapons' fire, so far our shots just pass right through the walls of the game world, flying off until the `Particle` class expires them. We need to implement collision detection with the tiles on the game map, thus allowing our shots to collide with walls and be destroyed.

Time for action – shots colliding with tiles

1. Add the `checkShotWallImpacts()` method to the `WeaponManager` class:

```
#region Collision Detection
private static void checkShotWallImpacts(Sprite shot)
{
    if (shot.Expired)
    {
        return;
    }

    if (TileMap.IsWallTile(
        TileMap.GetSquareAtPixel(shot.WorldCenter)))
    {
        shot.Expired = true;
        if (shot.Frame == 0)
        {
```

```

        EffectsManager.AddSparksEffect(
            shot.WorldCenter,
            shot.Velocity);
    }
    else
    {
        createLargeExplosion(shot.WorldCenter);
    }
}
#endregion

```

- 2.** Add the `createLargeExplosion()` method to the Effects Management Methods region of the `WeaponManager` class:

```

private static void createLargeExplosion(Vector2 location)
{
    EffectsManager.AddLargeExplosion(
        location + new Vector2(-10, -10));
    EffectsManager.AddLargeExplosion(
        location + new Vector2(-10, 10));
    EffectsManager.AddLargeExplosion(
        location + new Vector2(10, 10));
    EffectsManager.AddLargeExplosion(
        location + new Vector2(10, -10));
    EffectsManager.AddLargeExplosion(location);
}

```

- 3.** Modify the `Update()` method of the `WeaponManager` class and replace the `for` loop that updates the `Shots` list with the following:

```

for (int x = Shots.Count - 1; x >= 0; x--)
{
    Shots[x].Update(gameTime);
    checkShotWallImpacts(Shots[x]);
    if (Shots[x].Expired)
    {
        Shots.RemoveAt(x);
    }
}

```

4. Launch Robot Rampage and fire shots into the walls. They now erupt into a shower of sparks when they impact a wall tile:



What just happened?

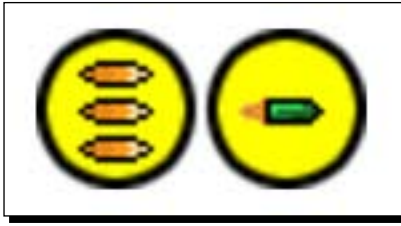
A shot that has already impacted something cannot impact a wall, so we check the `Expired` property of the sprite that is passed to the `checkShotWallImpact()` method and exit if it has already been consumed.

If the shot is still active, the only thing we need to do to check for wall impacts is test the center pixel of the shot sprite to see if the map square it is on contains a wall tile. If it does, the shot has just entered a wall tile and we can mark it `Expired` and generate a graphical effect. If the shot's `Frame` property is zero, we know the shot is a `WeaponType.Normal` bullet, so we call `AddSparksEffect()` at the shot's location.

If the shot is a rocket (in other words, its `Frame` is one), we call `createLargeExplosion()`, which calls `AddLargeExplosion()` five times, once for the center of the explosion and four times spread out around the center. The center explosion is generated last so that its graphics will be on top of the other four, providing a visual focal point for the explosion.

Power-ups

We can *fire* upgraded weapons, but currently there is no way for the player to actually get the upgraded weaponry. We will spawn power-ups at random locations around the game world that the player can drive over to activate their upgraded weaponry:



The power-ups we will spawn will be represented by yellow circles with the weapon type icon inside them. Up to five will be available at any given time.

Time for action – power-ups

1. Add the following declarations to the `WeaponManager` class:

```
static public List<Sprite> PowerUps = new List<Sprite>();
static private int maxActivePowerups = 5;
static private float timeSinceLastPowerup = 0.0f;
static private float timeBetweenPowerups = 2.0f;
static private Random rand = new Random();
```

2. Add the `tryToSpawnPowerup()` method to the Weapons Management Methods region of the `WeaponManager` class:

```
private static void tryToSpawnPowerup(int x, int y, WeaponType
type)
{
    if (PowerUps.Count >= maxActivePowerups)
    {
        return;
    }

    Rectangle thisDestination =
        TileMap.SquareWorldRectangle(new Vector2(x,y));

    foreach (Sprite powerup in PowerUps)
    {
        if (powerup.WorldRectangle == thisDestination)
        {
            return;
        }
    }

    if (!TileMap.IsWallTile(x,y))
    {
```

```
        Sprite newPowerup = new Sprite(
            new Vector2(thisDestination.X, thisDestination.Y),
            Texture,
            new Rectangle(64, 128, 32, 32),
            Vector2.Zero);
        newPowerup.Animate = false;
        newPowerup.CollisionRadius = 14;
        newPowerup.AddFrame(new Rectangle(96, 128, 32, 32));
        if (type == WeaponType.Rocket)
            newPowerup.Frame = 1;
        PowerUps.Add(newPowerup);
        timeSinceLastPowerup = 0.0f;
    }
}
```

- 3.** Add the `checkPowerupSpawns()` method to the Weapons Management Methods region of the `WeaponManager` class:

```
private static void checkPowerupSpawns(float elapsed)
{
    timeSinceLastPowerup += elapsed;
    if (timeSinceLastPowerup >= timeBetweenPowerups)
    {
        WeaponType type = WeaponType.Triple;
        if (rand.Next(0, 2) == 1)
        {
            type = WeaponType.Rocket;
        }
        tryToSpawnPowerup(
            rand.Next(0, TileMap.MapWidth),
            rand.Next(0, TileMap.MapHeight),
            type);
    }
}
```

- 4.** Add the `checkPowerupPickups()` method to the Collision Detection region of the `WeaponManager` class:

```
private static void checkPowerupPickups()
{
    for (int x = PowerUps.Count - 1; x >= 0; x--)
    {
        if (Player.BaseSprite.IsCircleColliding(
            PowerUps[x].WorldCenter,
            PowerUps[x].CollisionRadius))
        {
            switch (PowerUps[x].Frame)
            {
                case 0: CurrentWeaponType = WeaponType.Triple;
                        break;

                case 1: CurrentWeaponType = WeaponType.Rocket;
                        break;

            }
            WeaponTimeRemaining = weaponTimeDefault;
            PowerUps.RemoveAt(x);
        }
    }
}
```

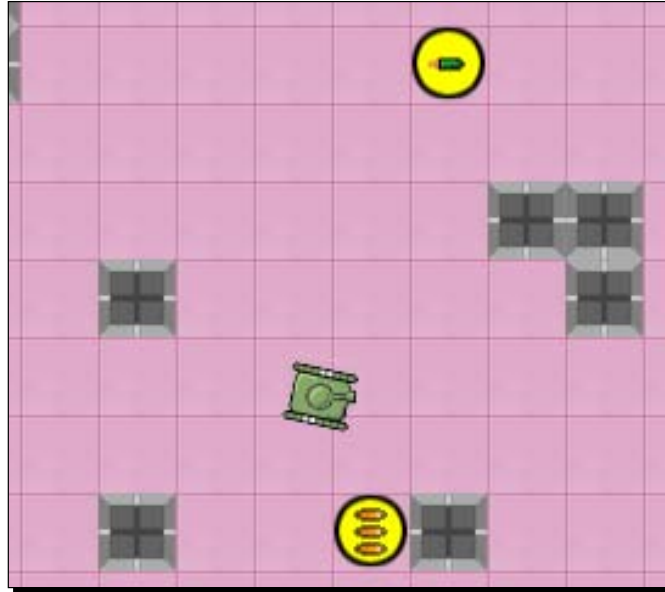
- 5.** Modify the `Update()` method of the `WeaponManager` class, adding the following lines as the last two lines of the method:

```
checkPowerupSpawns(elapsed);
checkPowerupPickups();
```

- 6.** Modify the `Draw()` method of the `WeaponManager` class, adding the following loop below the existing loop that draws the shots:

```
foreach (Sprite sprite in PowerUps)
{
    sprite.Draw(spriteBatch);
}
```


7. Launch Robot Rampage and drive around on the game map. Up to five power-ups will spawn on the map. Driving over one will activate the upgraded weapon and remove the power-up from the map:



What just happened?

As with most of our game objects, we will use a `List` to store the power-ups that are currently active in the game world. By setting `maxActivePowerups`, we can control the number that can be in the game world at any one time. New power-ups will spawn every two seconds (according to the `timeBetweenPowerups` member variable) whenever there are less than five active in the world.

During the `Update()` method, the `checkPowerupSpawns()` method is called, which checks to see if the timer indicates that a new power-up should be spawned. If enough time has elapsed, a weapon type (triple or rocket) is randomly determined, and `tryToSpawnPowerup()` is called, passing the weapon type and a random map location.

`tryToSpawnPowerup()` checks to make sure that we do not already have the maximum number of power-ups in the world. If we do, it simply returns without doing anything. The same is true if the method checks the existing power-ups and finds one already at the location passed to it. There is no benefit to stacking power-ups on top of each other, after all.

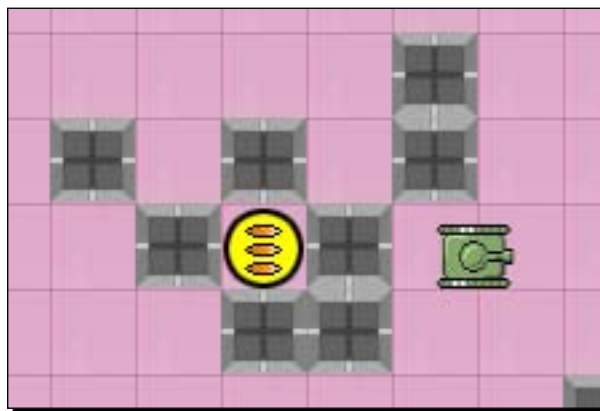
Finally, `tryToSpawnPowerup()` verifies that the tile we are going to place the power-up on is not a wall. The new power-up is generated as a `Sprite` object, and just like we did with the player's weapon fire, we use the frame number to determine if the power-up contains the triple cannon image or the rocket image. The power-up is added to the `PowerUps` list, and the spawn timer is reset to zero.

The `checkPowerupPickups()` method checks for a circle-based collision between the player and each of the active power-ups. If a collision is detected, the `CurrentWeaponType` member is set to the appropriate value based on the `Frame` property of the power-up (zero for triple cannons, one for rockets.), `WeaponTimeRemaining` is set to the value of `weaponTimeDefault`, and the power-up is removed from the `PowerUps` list.

It should be noted that `tryToSpawnPowerup()` does just that—attempt to spawn a power-up if it does not detect any problems with its placement. If it runs into a problem, it returns without doing anything. Since the `checkPowerupSpawns()` method will be run again during the next frame, this does not pose a problem. If we are not able to create a power-up in a given frame, we will catch up over the next couple of frames. Why not just keep generating values until we find one that works? Hold that thought because...

We still have a problem...

We now have advanced weaponry and a way to get that weaponry into the player's hands. We even check to make sure the power-ups do not spawn inside walls, so what could possibly go wrong? The answer lies in the following picture:



Since our map is randomly generated, it is quite possible that power-ups (and later on, enemy robots and computer terminals) will be spawned inside areas that the player cannot reach. In order to prevent that, we need to come up with a way to make sure that the square we decide to generate a game object in is actually reachable from the player's position.

Path finding

To resolve this problem, we need to implement a path-finding system that will allow us to easily determine the shortest route between any two squares on the tile map. The path needs to take walls into account, and needs to be fast enough that several enemy tanks can run the check without bogging down the game because in addition to verifying the placement of game objects, the same code will be used to allow enemy robots to move towards the player and attempt to destroy him.

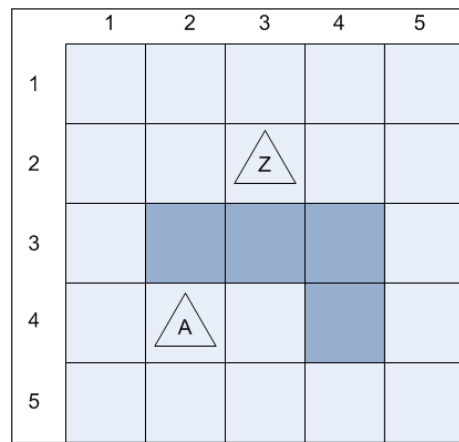
The A* path-finding algorithm

The method we will implement for Robot Rampage is called A* (pronounced "A Star"). This path-finding system is fairly straightforward and relatively fast, as it uses an educated guess system to try out potential paths between two points.

In order to implement A*, we need a few pieces of information:

- ◆ A way to identify nodes that objects can move between
- ◆ A starting node
- ◆ An ending node
- ◆ A method of determining the direct cost of moving between nodes
- ◆ A method of determining the indirect cost of moving between nodes

We can easily satisfy the first condition: each node in our path-finding system is represented by a square on the tile map grid. Similarly, defining a starting and ending point is straightforward as well. In the following example diagram, object **A** (located at (2, 4) on the grid) wishes to move to the position of object **Z** (3, 2):



That just leaves determining the cost of moving between squares. A* uses two different costs that together allow it to determine the shortest path between two points. The first type of cost is a direct movement cost. This cost will always be the same for any two squares with the same relationship to each other.

If your pathing algorithm only allowed horizontal and vertical movement (no diagonals), then the direct cost could always be assumed to be the same. The direct cost is used to represent the advantage of diagonal movement over a two-step movement.

For example, if object **A** wants to move one square down and to the right to location (3, 5) it could do so in one of the following three ways. It could move right, and then down. It could move down, and then right or it could move diagonally right and down.

If we assign horizontal and vertical movement an arbitrary cost of 10 units (units of what is not really important), we can assign moving diagonally a cost that is between the cost to move one square (10 units) and the cost of moving two squares (20 units). We will split the difference and call the cost of moving diagonally 15 units. All things being equal, we can now define that moving diagonally is five units cheaper than moving straight twice to arrive at the same location.

This leaves the indirect cost of movement. This can be thought of as, "How much closer does this movement get me to where I want to be?" In our case, this is a very simple question to answer—all we need is to measure the distance between the square we are considering moving to and the destination square.

To keep the discussion simple, we will consider the Manhattan Distance between our squares, or the distance if you could only move in straight lines. This can be calculated simply by adding the difference between the X positions and the difference between the Y positions together.

Manhattan versus Linear Distance



We are using Manhattan Distance (the term comes from the distance you would need to walk in a city arranged in blocks, where you would walk so-many-blocks east and then so-many-blocks north to reach a destination, since you cannot walk through buildings) because it keeps the distances we are discussing in whole numbers.

When diagonal movement is allowed, Linear Distance becomes a little more accurate than Manhattan Distance and we will use that in our path-finding code. This is simply the direct distance between the two points. In the case of **A** and **Z** in our diagram, the direct distance is approximately 2.24, calculated by using the Pythagorean Theorem.

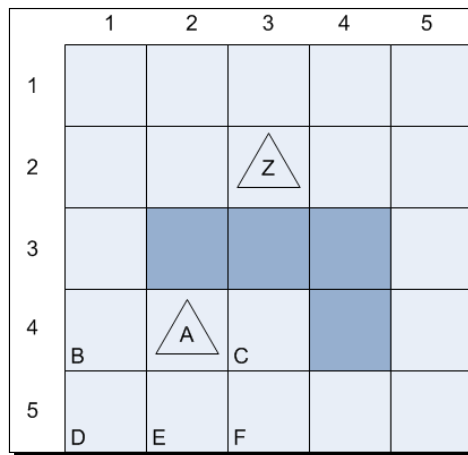
In our example image, the Manhattan Distance between **A** and **Z** is 3 squares. They are one square apart horizontally and two squares apart vertically. If we consider the square directly to the right of object **A**, the distance to object **Z** is only 2 squares. From the square to the left of object **A**, the distance to the destination has increased to 4 squares (two up and two across).

In order to actually find a path with A*, we keep two lists of nodes. On the Open List, we store all of the nodes that we have identified as possible moves but that we have not evaluated yet.

The second list or the Closed List, stores all of the nodes we have already checked. When we evaluate a node, we will move it from the Open List to the Closed List.

When we begin the path-finding check, we start by adding the starting node to the Open List, and then check all of the adjacent nodes to see if we can move into them. In the case of our diagram, the node directly above the starting point, along with the node diagonally above and to the right, are walls, so they are not possible movement nodes. The other six nodes are not walls, but because we would need to partially move through the wall above the starting point to reach the node diagonally above and to the left of **A**, we will not consider moving there diagonally as a valid move.

That leaves five nodes, marked in the following diagram, that are valid movement locations from the starting node. The direct cost of moving to nodes **B**, **C**, and **E** is 10, and the direct cost of moving to nodes **D** and **F** is 15 based on our assignments for straight and diagonal moves:



Next we must determine the indirect costs (again we will use Manhattan costs). We simply count the squares up and across from each node to reach node **Z**. Once we have these costs, we can add the direct and indirect costs to get the total cost for moving to the node:

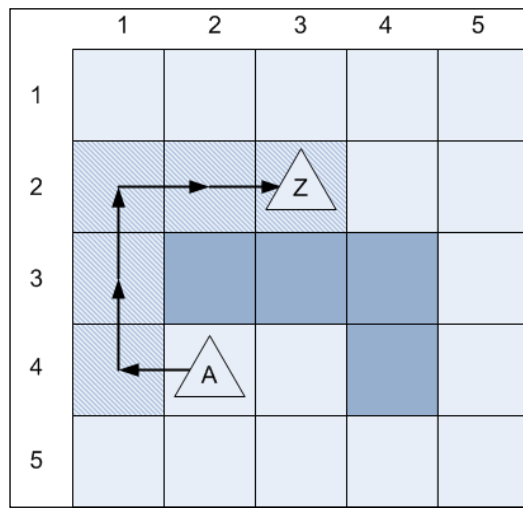
Node	Direct Cost	Indirect Cost	Total Cost
B	10	4	14
C	10	2	12
D	15	5	20
E	10	4	14
F	15	3	18

All of the nodes are then added to the Open List, which we will process in the lowest-cost-first order. In this case, square **C** has the lowest total cost, so it would be the next square to be evaluated. Square **A** is added to the Closed List, since we have already evaluated everywhere we can go from there.

Whenever a square is evaluated, the cost of getting to that square is considered in the costs of moving to any of the other squares it is adjacent to. So from Node **C** we *could* move to squares **E** or **F**. In fact, because we are blocked to the right and eliminate the diagonal movement right and downward due to that block, these are the only moves that we can make from square **C**. However, the costs to move from **C** to either **E** or **F** would be:

Node	Previous Cost	Direct Cost	Indirect Cost	Total Cost
E	12	15	4	31
F	12	10	3	25

Since we can reach both **E** and **F** for much lower costs directly from **A** (they are already on the Open List at a lower cost), we discard these possibilities and move on, checking the next lowest cost node on the Open List, which is a tie between **B** and **E**. Node **B** ends up adding the square above it to the Open List at a total cost of 27 (built-in cost of 14, direct cost of 10, and indirect cost of 3) and no other nodes:



This process of evaluating nodes continues until the target square has been reached. Since we have taken the lowest cost available move at every opportunity, the path taken to reach the destination square will be the shortest available on the grid. The resulting path, shown in the previous diagram, moves around the obstacle to reach the destination in as few moves as possible.

Implementing A*

Now that we have the theoretical background on A* in place, let's implement it in C# for use in our game. Our first task is defining a node within our pathing structure.

Time for action – the PathNode class

1. Add a new class called PathNode to the Robot Rampage project.
2. Add the following `using` directive to the top of the class file:
`using Microsoft.Xna.Framework;`
3. Add declarations to the PathNode class:
`#region Declarations`

```

public PathNode ParentNode;
public PathNode EndNode;
private Vector2 gridLocation;
public float TotalCost;
public float DirectCost;
#endregion

```

4. Add properties to the PathNode class:

```

#region Properties
public Vector2 GridLocation
{
    get { return gridLocation; }
    set {
        gridLocation = new Vector2(
            (float)MathHelper.Clamp(value.X, 0f, (float)TileMap.
                MapWidth),
            (float)MathHelper.Clamp(value.Y, 0f, (float)TileMap.
                MapHeight));
    }
}

public int GridX
{
    get { return (int)gridLocation.X; }
}

public int GridY
{
    get { return (int)gridLocation.Y; }
}
#endregion

```

5. Add a constructor to the PathNode class:

```

#region Constructor
public PathNode(
    PathNode parentNode,
    PathNode endNode,
    Vector2 gridLocation,
    float cost)
{
    ParentNode = parentNode;
    GridLocation = gridLocation;
    EndNode = endNode;
}

```



```
        DirectCost = cost;
        if (!(endNode==null))
        {
            TotalCost = DirectCost + LinearCost();
        }
    }
#endregion
```

6. Add a helper method to determine the indirect cost for a node:

```
#region Helper Methods
public float LinearCost()
{
    return (
        Vector2.Distance(
            EndNode.GridLocation,
            this.GridLocation));
}
#endregion
```

7. Add public methods to allow the node to be compared with other nodes:

```
#region Public Methods
public bool IsEqualToNode(PathNode node)
{
    return (GridLocation == node.GridLocation);
}
#endregion
```

What just happened?

When building a chain of nodes that represent a path, each node needs to know what node it was arrived at from. The `ParentNode` member variable stores the reference to that node, while the `EndNode` member stores the node that the search system is seeking as the destination. This information will be needed in calculating the indirect cost of the node.

The `gridLocation` member and its associated `GridLocation` property represent the X and Y coordinates of the node on the tile map, identifying the node's position in the game world. The `GridX` and `GridY` properties provide a shortcut for accessing the individual components of the `gridLocation` vector.

When a new `PathNode` is created, its direct cost (the cost associated with either a horizontal or vertical move) is stored in the `DirectCost` variable. If the `endNode` passed to the constructor is not null (which will be the case when the end node itself is created), the `TotalCost` member is calculated by adding the `DirectCost` and the result of the `LinearCost()` method, which measures the distance between the node and the end node. We could make the `TotalCost` a property that adds the two values and returns the result, but the `Vector2.Distance()` method is relatively costly when it is run hundreds of times comparing costs, so caching the value will give us a slight edge in performance.

The `IsEqualToNode()` method is needed because all classes are reference types in C#, meaning that the variables in our program are really pointers to objects in memory. Assigning a reference type to a new variable does not make a copy of the whole object, but rather it simply copies the pointer and creates a new reference to the same object.

In order to determine if two different instances of the `PathNode` object represent the same square on the map, `IsEqualToNode()` checks to see if the `GridLocation` properties of the two nodes match. Since a `Vector2` is a value type, the comparison happens between the actual data values in the `Vector2` instead of reference pointers.

Now it is time to build the actual path-finding code, which we are going to do in a few stages for clarity. To begin, let's create a new static class, which will handle pathing for our game.

Time for action – beginning the implementation of A*

1. Add a new class called `PathFinder` to the Robot Rampage project.

2. Add the following `using` directive to the top of the class file:

```
using Microsoft.Xna.Framework;
```

3. Modify the class declaration to make the class static:

```
static class PathFinder
```

4. Add declarations to the `PathFinder` class:

```
#region Declarations
private enum NodeStatus { Open, Closed };

private static Dictionary<Vector2, NodeStatus> nodeStatus =
    new Dictionary<Vector2, NodeStatus>();

private const int CostStraight = 10;
private const int CostDiagonal = 15;

private static List<PathNode> openList = new List<PathNode>();

private static Dictionary<Vector2, float> nodeCosts =
```

```
        new Dictionary<Vector2, float>();
    #endregion
```

5. Create a region in the Pathfinder class for helper methods:

```
    #region Helper Methods
    #endregion
```

6. Add the `addNodeToOpenList()` method to the Helper Methods region of the Pathfinder class:

```
static private void addNodeToOpenList(PathNode node)
{
    int index = 0;
    float cost = node.TotalCost;

    while ((openList.Count() > index) &&
        (cost < openList[index].TotalCost))
    {
        index++;
    }

    openList.Insert(index, node);
    nodeCosts[node.GridLocation] = node.TotalCost;
    nodeStatus[node.GridLocation] = NodeStatus.Open;
}
```

What just happened?

For readability purposes, we have defined an `enum` for tracking the status of each node, along with a `Dictionary` object called `nodeStatus` that will hold the current status of any nodes we have either checked already (closed nodes) or that we are interested in checking (open nodes).

The `CostStraight` and `CostDiagonal` constants will be used with our lists of nodes. With these costs, we can see that a diagonal move is 5 units cheaper than the same move made as two straight moves. This difference in cost will ensure that the path finder selects diagonal movement whenever possible.

In addition to the `nodeStatus` Dictionary, we also define the `openList`, which will store all of the nodes that the algorithm has identified as possible movement locations. While it is true that open node information is available in the `nodeStatus` dictionary, we give up a bit of memory for the speed increase of maintaining the `openList` as a separate, sorted list instead of searching the `nodeStatus` dictionary. Recall that we always want to investigate the lowest cost node on the open list next, so keeping a sorted list will allow us to pick the lowest cost item immediately instead of comparing the entire contents of the dictionary against one another.

The `nodeCosts` dictionary represents a similar trade-off of memory for performance. The `openList` allows us to quickly answer the question "What is the lowest cost option?", but when we want to know "What is the cost of the node at position X, Y?", we would have to search the whole `openList` for that particular node. We will store costs in their own dictionary indexed by the node position vector to provide fast access to the node costs.

The code in step 6 should look somewhat familiar, as it is similar to the way we stored high scores in Asteroid Belt Assault. By moving the index value up the list until we reach a score that is lower than the one we are inserting, and then placing the score at that location, we generate an automatically sorted list. At the same time as the item is inserted into the `openList`, we set the `nodeCosts` and `nodeStatus` dictionaries to their corresponding values.

Time for action – finding the path

1. Add the `FindPath()` method to the `PathFinder` class:

```
#region Public Methods
static public List<Vector2> FindPath(
    Vector2 startTile,
    Vector2 endTile)
{
    if (TileMap.IsWallTile(endTile) ||
        TileMap.IsWallTile(startTile))
    {
        return null;
    }

    openList.Clear();
    nodeCosts.Clear();
    nodeStatus.Clear();

    PathNode startNode;
    PathNode endNode;

    endNode = new PathNode(null, null, endTile, 0);
```

```
startNode = new PathNode(null, endNode, startTile, 0);
addNodeToOpenList(startNode);
while (openList.Count > 0)
{
    PathNode currentNode = openList[openList.Count - 1];
    if (currentNode.IsEqualToNode(endNode))
    {
        List<Vector2> bestPath = new List<Vector2>();
        while (currentNode != null)
        {
            bestPath.Insert(0, currentNode.GridLocation);
            currentNode = currentNode.ParentNode;
        }
        return bestPath;
    }
    openList.Remove(currentNode);
    nodeCosts.Remove(currentNode.GridLocation);
    foreach (
        PathNode possibleNode in
        findAdjacentNodes(currentNode, endNode))
    {
        if (nodeStatus.ContainsKey(possibleNode.GridLocation))
        {
            if (nodeStatus[possibleNode.GridLocation] ==
                NodeStatus.Closed)
            {
                continue;
            }
            if (
                nodeStatus[possibleNode.GridLocation] ==
                NodeStatus.Open)
            {
                if (possibleNode.TotalCost >=
                    nodeCosts[possibleNode.GridLocation])
                {
                    continue;
                }
            }
        }
    }
}
```

```

        }
        addNodeToOpenList (possibleNode) ;
    }
    nodeStatus [currentNode.GridLocation] =NodeStatus.Closed;
}
return null;
}
#endregion

```

What just happened?

The `FindPath()` method is the heart of the `PathFinder` class, and begins by checking to see if either the start or the end node contains a wall tile. If either of these squares is a wall, it is impossible to find a path between them, so the method immediately returns `null` to the caller.

With that initial check out of the way, the `openList`, `nodeCosts`, and `nodeStatus` collections are cleared in preparation for testing the path between the two points. The two passed vectors are then converted into `PathNode` objects called `startNode` and `endNode`. Recall that we will need to pass `endNode` to every `PathNode` object that we create so that the nodes can measure their indirect costs. Normal nodes contain a reference to their parent node, which is the node we moved to this node from. Neither the `startNode` nor the `endNode` requires a parent node, and the `endNode` does not require a reference to itself, so all of these parameters are passed as `null`.

Next, we add the starting node to the `openList`, and begin a `while` loop, which will continue until we run out of nodes to check, or until we find the path that we are looking for. The loop begins by getting the lowest-cost node from the `openList` (recall that it is sorted in highest-to-lowest order, so that the last item on the list will be the lowest-cost node).

If the node we just pulled from the `openList` is the `endNode`, we have reached the destination and can return the path to the calling code. We do this by inserting the `GridLocation` of the current node at index location zero into the `bestPath` list and then moving to the `parentNode` of the current node and repeating the process until we run out of parent nodes. The result is that we have a list of vectors sorted in order from the starting point to the ending square.

Assuming that the current node is not the node we are trying to reach, we remove the current node from the `openList`, and remove the associated item from the `nodeCosts` dictionary (because the dictionary is used to track the costs for items on the `openList`, and the current node was just removed from that list).

Now, we need to check all of the surrounding squares on the tile map and see if we are able to move into any of them. The `findAdjacentNodes()` method (which we will be adding in a moment) returns a list of all of the nodes that it is legal to move into from the current node.

Each of these possible nodes is then checked to see if it is already on the closed list. If it is, there is no need to examine this node further, so a `continue` statement is executed to proceed to the next item in the loop. The same is true for a node that is already on the open list at a lower cost. It needs no further processing either and another `continue` is executed.

If neither of the `if` statements above resulted in a `continue` (which would skip the rest of that iteration of the loop), the node is added to the `openList` as somewhere we need to potentially investigate for movement. After we have finished processing all of the potential new nodes, the current node's status is marked as `NodeStatus.Closed`.

If we make it all the way to the end of the `FindPath()` method, it means we have run out of nodes on the open list without finding a path from the starting node to the ending node. In this case, we return a `null` value, just like we did in the case of starting or ending on a wall.

Time for action – adjacent squares

1. Add the `findAdjacentNodes()` method to the Helper Methods region of the `PathFinder` class:

```
static private List<PathNode> findAdjacentNodes(  
    PathNode currentNode,  
    PathNode endNode)  
{  
    List<PathNode> adjacentNodes = new List<PathNode>();  
  
    int X = currentNode.GridX;  
    int Y = currentNode.GridY;  
  
    bool upLeft = true;  
    bool upRight = true;  
    bool downLeft = true;  
    bool downRight = true;  
  
    if ((X > 0) && (!TileMap.IsWallTile(X - 1, Y)))  
    {  
        adjacentNodes.Add(new PathNode(  
            currentNode,  
            endNode,
```

```
        new Vector2(X - 1, Y),
        CostStraight + currentNode.DirectCost));
    }
    else
    {
        upLeft = false;
        downLeft = false;
    }
    if ((X < 49) && (!TileMap.IsWallTile(X + 1, Y)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X + 1, Y),
            CostStraight + currentNode.DirectCost));
    }
    else
    {
        upRight = false;
        downRight = false;
    }
    if ((Y > 0) && (!TileMap.IsWallTile(X, Y - 1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X, Y - 1),
            CostStraight + currentNode.DirectCost));
    }
    else
    {
        upLeft = false;
        upRight = false;
    }
    if ((Y < 49) && (!TileMap.IsWallTile(X, Y + 1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
```



```
        new Vector2(X, Y + 1),
        CostStraight + currentNode.DirectCost));
    }
    else
    {
        downLeft = false;
        downRight = false;
    }

    if ((upLeft) && (!TileMap.IsWallTile(X-1,Y-1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X - 1, Y - 1),
            CostDiagonal + currentNode.DirectCost));
    }

    if ((upRight) && (!TileMap.IsWallTile(X + 1, Y - 1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X + 1, Y - 1),
            CostDiagonal + currentNode.DirectCost));
    }

    if ((downLeft) && (!TileMap.IsWallTile(X - 1, Y + 1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X - 1, Y + 1),
            CostDiagonal + currentNode.DirectCost));
    }

    if ((downRight) && (!TileMap.IsWallTile(X + 1, Y + 1)))
    {
        adjacentNodes.Add(new PathNode(
            currentNode,
            endNode,
            new Vector2(X + 1, Y + 1),
            CostDiagonal + currentNode.DirectCost));
    }
}
```

```

    }
    return adjacentNodes;
}

```

- 2.** Open the `Game1.cs` class file and add the following line to the `Initialize()` method:

```
this.IsMouseVisible = true;
```

- 3.** In the `Draw()` method of the `Game1` class, add the following code snippet right before the `spriteBatch.End()` call:

```

// Temporary Code Begin
Vector2 mouseLocation = new Vector2(
    Mouse.GetState().X, Mouse.GetState().Y);

mouseLocation += Camera.Position;

List<Vector2> path = PathFinder.FindPath(
    TileMap.GetSquareAtPixel(mouseLocation),
    TileMap.GetSquareAtPixel(Player.BaseSprite.WorldCenter));

if (!(path == null))
{
    foreach (Vector2 node in path)
    {
        spriteBatch.Draw(
            spriteSheet,
            TileMap.SquareScreenRectangle((int)node.X,
                (int)node.Y),
            new Rectangle(0, 288, 32, 32),
            new Color(128, 0, 0, 80));
    }
}
// Temporary Code End

```

4. Execute Robot Rampage. Move your mouse around on the screen to see the highlighted path generated by the Pathfinder class:



5. After you have finished viewing the path, remove the line you added to the `Initialize()` method and the temporary code from the `Draw()` method.

What just happened?

After caching the X and Y grid coordinates of the node we want to check into local integer variables, the `findAdjacentNodes()` method creates four Boolean variables representing the four diagonal directions and defaults them to true.

In order for a diagonal move to be allowed, both of the adjoining squares need to be floor tiles as well as the diagonal square, otherwise the object moving along the path would pass partially through a wall tile on its way diagonally to the destination square.

Next, a series of four `if` statements check each of the axis-aligned directions (right, left, up, and down) for wall tiles. If the square in question is not a wall tile, it is added to the `adjacentNodes` list, which will be returned from the function. Whenever a node is added, its cost is set to the direct cost of the movement (either `CostStraight` or `CostDiagonal`) plus the cost of reaching the current square.

If the square does contain a wall tile, it is not added to the `adjacentNodes` list and the two diagonals next to it are also set to false, meaning they will not be valid either. For example, if we are evaluating the square above the current square, and it contains a wall, the diagonal moves up/right and up/left will be invalidated.

After the four `if` statements for the straight-line moves have been evaluated, any of the diagonal direction Boolean values that remain true are checked in the same manner. If the square does not contain a wall tile, it is added to the `adjacentNodes` list.

Once all eight directions have been evaluated, the `adjacentNodes` list is returned from the function to be processed by the `FindPath()` method.

In steps two and three, we added some demo code to activate the mouse cursor in the game, and execute the `FindPath()` method during each draw cycle, using the mouse cursor location as the starting point, and the current player position as the ending point of the path. Each square along the path is then drawn using the empty white square on the tile sheet to colorize the square on the screen with a half-transparent red color.

When the mouse position is retrieved, it is in screen coordinates, relative to the upper left corner of the display window. Since we need to convert it to world coordinates, we have to add the current `Camera.Position` value to the mouse location. This is just the opposite of what the `Camera.Transform()` method does, which converts world coordinates to screen coordinates.

Safely placing power-ups

Now that we can find a path between two points, we can update our `WeaponManager` class to take pathing into account when generating power-ups.

Time for action – updating the `WeaponManager` class

1. Add a property to the `Player` class to create a shortcut to the square the player is currently located in:

```
#region Properties
public static Vector2 PathingNodePosition
{
    get
```

```
        {  
            return TileMap.GetSquareAtPixel(BaseSprite.WorldCenter);  
        }  
    }  
#endregion
```

2. Modify the `tryToSpawnPowerup()` method of the `WeaponManager` class and replace the `if` statement that checks to see if the tile the power-up is being placed on is a wall (it currently reads `if (!TileMap.IsWallTile(x,y))`) with the following:

```
if (!(PathFinder.FindPath(  
    new Vector2(x,y),  
    Player.PathingNodePosition)==null))
```

3. Execute Robot Rampage and explore the map, looking for the power-ups.

What just happened?

Since the `FindPath()` method returns null immediately if either the starting or ending square is a wall, this new condition will cover trying to place a power-up on a wall tile as well as accounting for unreachable tiles on the map.

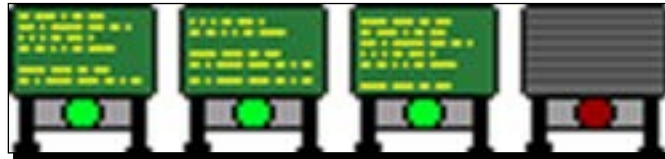
We talked in the power-ups section about not repeatedly attempting to generate a power-up by coming up with new locations until we find one that works, and the potential speed impact of the `FindPath()` method is the reason we do not do this. If we got unlucky and ended up needing to run `FindPath()` a few times when attempting to generate a power-up, it might cause the game to stutter as the path-finding code can be processor-intensive over long distances.

Player goals

The player's purpose in the game world is two-fold. First, they need to destroy enemy robots that are trying to destroy them. Second, they need to shut down the computer terminals that are building the enemy tanks. Since the computers spawn the tanks, we will implement the terminals first and then add our robotic enemies.

Computer terminals

Our path-finding code will be put to the same purpose in the generation of computer terminals that it was for power-ups. After all, the player cannot shut down a terminal that they are unable to reach:



Each terminal will be represented by a Sprite that plays a multi-frame animation when active. When the player shuts down a terminal, the sprite will be replaced by an inactive terminal image.

Time for action – building a computer terminal

1. Add a new class called `ComputerTerminal` to the Robot Rampage project.

2. Add the following `using` directives to the top of the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the `ComputerTerminal` class:

```
#region Declarations
private Sprite activeSprite;
private Sprite inactiveSprite;
public Vector2 MapLocation;
public bool Active = true;
public float LastSpawnCounter = 0;
public float minSpawnTime = 6.0f;
#endregion
```

4. Add a constructor to the `ComputerTerminal` class:

```
#region Constructor
public ComputerTerminal(
    Sprite activeSprite,
    Sprite inactiveSprite,
    Vector2 mapLocation)
{
    MapLocation = mapLocation;
}
```

```
        this.activeSprite = activeSprite;
        this.inactiveSprite = inactiveSprite;
    }
    #endregion
```

5. Add public methods to the ComputerTerminal class:

```
#region Public Methods
public bool IsCircleColliding(Vector2 otherCenter, float radius)
{
    if (!Active)
    {
        return false;
    }

    return activeSprite.IsCircleColliding(otherCenter, radius);
}

public void Deactivate()
{
    Active = false;
}

public void Update(GameTime gameTime)
{
    if (Active)
    {
        float elapsed = (float)gameTime.ElapsedGameTime.
            TotalSeconds;

        LastSpawnCounter += elapsed;
        if (LastSpawnCounter > minSpawnTime)
        {
            LastSpawnCounter = 0;
        }

        activeSprite.Update(gameTime);
    }
    else
    {
        inactiveSprite.Update(gameTime);
    }
}
}
```

```

public void Draw(SpriteBatch spriteBatch)
{
    if (Active)
    {
        activeSprite.Draw(spriteBatch);
    }
    else
    {
        inactiveSprite.Draw(spriteBatch);
    }
}
#endregion

```

What just happened?

The two possible states of each `ComputerTerminal` are represented by the two sprites `activeSprite` and `inactiveSprite`. We also cache the map-square-based location of the `ComputerTerminal` in the `MapLocation` vector. We will use this variable when we spawn enemy robots.

Other than `Update()` and `Draw()`, we have two public methods. `IsCircleColliding()` returns `false` if the terminal is not active. Otherwise, it passes the call on to the `IsCircleColliding()` method of `activeSprite`. The `Deactivate()` method simply sets `Active` to `false`.

During the `Update()` method, we check to see if the sprite is active. If it is, the spawn time mechanism is updated. Currently, when it is time to spawn a new robot our `Update()` code just resets the timer. We will be revisiting this method after we have constructed our enemy robots. Either the `activeSprite` or the `inactiveSprite` (depending on the state of the `Active` variable) is passed the `Update()` call before the method exits.

Similarly, the `Draw()` method uses the `Active` variable to determine which of the `Sprite` objects to pass the `Draw()` call to.

Spawning computer terminals

Now that we have defined the `ComputerTerminal` class, we need to create a way for them to appear on the game map. We will do this with a new class called `GoalManager`, which will be responsible for spawning computer terminals in accessible locations and monitoring to see when the player has shut down a computer terminal.

Time for action – the GoalManager class

1. Add a new class called GoalManager to the Robot Rampage project.
2. Add the following using directives to the top of the class file:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```

3. Modify the declaration of the GoalManager class to make it a static class:

```
static class GoalManager
```

4. Add declarations to the GoalManager class:

```
#region Declarations  
private static List<ComputerTerminal> computerTerminals =  
    new List<ComputerTerminal>();  
private static int activeCount = 0;  
private static int minDistanceFromPlayer = 250;  
private static Random rand = new Random();  
  
private static Texture2D texture;  
private static Rectangle initialActiveFrame;  
private static Rectangle initialDisabledFrame;  
private static int activeFrameCount;  
private static int disabledFrameCount;  
#endregion
```

5. Add a read-only property to the GoalManager class:

```
#region Properties  
public static int ActiveTerminals  
{  
    get { return activeCount; }  
}  
#endregion
```

6. Add the Initialize() method to the GoalManager class:

```
#region Initialization  
public static void Initialize(  
    Texture2D textureSheet,  
    Rectangle initialActiveRectangle,  
    Rectangle initialDisabledRectangle,  
    int activeFrames,
```

```

        int disabledFrames)
    {
        texture = textureSheet;
        initialActiveFrame = initialActiveRectangle;
        initialDisabledFrame = initialDisabledRectangle;
        activeFrameCount = activeFrames;
        disabledFrameCount = disabledFrames;
    }
#endregion

```

7. Add methods to manage ComputerTerminals to the GoalManager class:

```

#region Terminal Management
public static ComputerTerminal TerminalInSquare(
    Vector2 mapLocation)
{
    foreach (ComputerTerminal terminal in computerTerminals)
    {
        if (terminal.MapLocation == mapLocation)
        {
            return terminal;
        }
    }
    return null;
}

public static void DetectShutdowns()
{
    foreach (ComputerTerminal terminal in computerTerminals)
    {
        if (terminal.Active)
        {
            if (terminal.IsCircleColliding(
                Player.BaseSprite.WorldCenter,
                Player.BaseSprite.CollisionRadius))
            {
                terminal.Deactivate();
                activeCount--;
            }
        }
    }
}

```

```
public static void AddComputerTerminal()
{
    int startX = rand.Next(2, TileMap.MapWidth - 2);
    int startY = rand.Next(0, TileMap.MapHeight - 2);
    Vector2 tileLocation = new Vector2(startX, startY);
    if ((TerminalInSquare(tileLocation) != null) ||
        (TileMap.IsWallTile(tileLocation)))
    {
        return;
    }
    if (Vector2.Distance(
        TileMap.GetSquareCenter(startX, startY),
        Player.BaseSprite.WorldCenter) < minDistanceFromPlayer)
    {
        return;
    }
    List<Vector2> path =
        PathFinder.FindPath(
            new Vector2(startX, startY),
            TileMap.GetSquareAtPixel(
                Player.BaseSprite.WorldCenter));

    if (path != null)
    {
        Rectangle squareRect =
            TileMap.SquareWorldRectangle(startX, startY);

        Sprite activeSprite = new Sprite(
            new Vector2(squareRect.X, squareRect.Y),
            texture,
            initialActiveFrame,
            Vector2.Zero);

        for (int x = 1; x < 3; x++)
        {
            activeSprite.AddFrame(
                new Rectangle(
                    initialActiveFrame.X + (x *
                        initialActiveFrame.Width),
                    initialActiveFrame.Y,
```

```

        initialActiveFrame.Width,
        initialActiveFrame.Height));
    }
    activeSprite.CollisionRadius = 15;

    Sprite disabledSprite = new Sprite(
        new Vector2(squareRect.X, squareRect.Y),
        texture,
        initialDisabledFrame,
        Vector2.Zero);

    ComputerTerminal terminal = new ComputerTerminal(
        activeSprite,
        disabledSprite,
        new Vector2(startX, startY));

    float timerOffset = (float)rand.Next(1, 100);
    terminal.LastSpawnCounter = timerOffset / 100f;
    computerTerminals.Add(terminal);

    activeCount++;
}
}
#endregion

```

8. Add public methods to the GoalManager class:

```

#region Public Methods
public static void GenerateComputers(int computerCount)
{
    computerTerminals.Clear();
    activeCount = 0;

    while (activeCount < computerCount)
    {
        AddComputerTerminal();
    }
}

public static void Update(GameTime gameTime)
{
    DetectShutdowns();
    foreach (ComputerTerminal terminal in
        computerTerminals)
    {

```

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

public static void Draw(SpriteBatch spriteBatch)
{
    foreach (ComputerTerminal terminal in
        computerTerminals)
    {
        terminal.Draw(spriteBatch);
    }
}

#endregion
```

- 9.** In the `LoadContent()` method of the `Game1.cs` class file, initialize the `GoalManager` class and generate 10 computer terminals:

```
GoalManager.Initialize(
    spriteSheet,
    new Rectangle(0, 7 * 32, 32, 32),
    new Rectangle(3 * 32, 7 * 32, 32, 32),
    3,
    1);

GoalManager.GenerateComputers(10);
```

- 9.** Still in the `LoadContent()` method of `Game1`, modify the call to `Player.Initialize()` and change the player's initial position to ensure that they will never end up on a block that contains a wall:

```
Player.Initialize(
    spriteSheet,
    new Rectangle(0, 64, 32, 32),
    6,
    new Rectangle(0, 96, 32, 32),
    1,
    new Vector2(32, 32));
```

- 10.** In the `Update()` method of the `Game1` class, call the `GoalManager's Update()` method right after the `EffectsManager` has been updated:

```
GoalManager.Update(gameTime);
```

- 11.** In the `Draw()` method of the `Game1` class, call the `GoalManager's Draw()` method right after the `EffectsManager` has been drawn:

```
GoalManager.Draw(spriteBatch);
```

- 12.** Execute `Robot Rampage` and verify that the computer terminals have been spawned in appropriate locations. Moving over an active terminal switches it off.

What just happened?

The basics of our `GoalManager` should be very familiar by now. The individual terminals are stored in a list that is updated and drawn via loops in the `Update()` and `Draw()` methods.

During the `Initialize()` method, we establish all of the values that will let us create the sprites needed to represent the `ComputerTerminals` that will be created by the `AddComputerTerminal()` method.

When generating terminals, we will want to make sure that we do not place two terminals at the same map location, so the `TerminalInSquare()` method returns the `ComputerTerminal` that occupies a given map square, if one exists. By checking this method when we generate a new computer, we can verify that all of the terminals end up in different locations.

The `DetectShutdowns()` method iterates through each of the terminals in the `computerTerminals` list, checking to see if the player has collided with them. Remember that inactive terminals will always return a false for `IsCircleColliding()`. If the player has collided with a terminal, that terminal's `Deactivate()` method is called to shut it down.

The heart of the `GoalManager` class is the `AddComputerTerminal()` method, which begins by generating a random location on the map to place the terminal. If the randomly determined map square already contains a terminal, or if it is a wall square, the `AddComputerTerminal()` method returns without generating a computer.

We also check to make sure we are not generating the computer within 250 pixels of the player's current location. Given that the computers will be generated at the beginning of a game level, this will result in computers not being placed within 250 pixels of the player's starting location, which is the upper left corner of the map.

Assuming the square passes all of these checks, a path-finding call is made to make sure that the space being considered can be reached from the player's current location. Once again, if it cannot, the method exits without generating the computer.

If everything checks out OK, the two `Sprite` objects are created and then the terminal object is built using the two new `Sprites`. The `LastSpawnCounter` for the terminal is adjusted by the randomly generated `timerOffset` value. By introducing a small random variation to the time at which terminals will spawn enemies, we can spread out the path-finding checks that will be required during the spawns to minimize the processing hit that would take place if all of the terminals spawned enemies during the same frame.

Finally, the terminal is added to the `computerTerminals` list and the `activeCount` variable is incremented.

The `GenerateComputers()` method, which will be called when a level is generated, clears the `computerTerminals` list and adds terminals until `activeCount` reaches the number of computers passed to the `GenerateComputers()` method.

In addition to its normal responsibility to update all of the items in the `computerTerminals` list, the `Update()` method also calls the `DetectShutdowns()` method, allowing the player to turn the computer terminals off.

Enemy robots

The last component we need to make our game playable is the actual enemy robots generated at the computer terminals. Visually, these robots are distinct from the player's tank not only by color, but also by wheel and turret design. The enemy robots are four-wheeled vehicles with rotatable claw-like arms. They will pursue the player through the game map, attempting to physically contact the player and destroy their tank:



Enemy basics

As with most of our game objects, we will build both the `Enemy` class, and a manager for the class that will handle updating and drawing the enemy robots as well as coordinating them with the rest of the game.

Time for action – building the Enemy class

1. Add a new class called Enemy to the Robot Rampage project.
2. Add the following using directives to the top of the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Add declarations to the Enemy class:

```
#region Declarations
public Sprite EnemyBase;
public Sprite EnemyClaws;
public float EnemySpeed = 60f;
public Vector2 currentTargetSquare;
public bool Destroyed = false;
private int collisionRadius = 14;
#endregion
```

4. Add a constructor to the Enemy class:

```
#region Constructor
public Enemy(
    Vector2 worldLocation,
    Texture2D texture,
    Rectangle initialFrame)
{
    EnemyBase = new Sprite(
        worldLocation,
        texture,
        initialFrame,
        Vector2.Zero);

    EnemyBase.CollisionRadius = collisionRadius;

    Rectangle turretFrame = initialFrame;

    turretFrame.Offset(0, initialFrame.Height);
    EnemyClaws = new Sprite(
        worldLocation,
        texture,
        turretFrame,
        Vector2.Zero);
}
#endregion
```


5. Add the Update() and Draw() methods to the Enemy class:

```
#region Public Methods
public void Update(GameTime gameTime)
{
    if (!Destroyed)
    {
        Vector2 direction = determineMoveDirection();
        direction.Normalize();

        EnemyBase.Velocity = direction * EnemySpeed;
        EnemyBase.RotateTo(direction);
        EnemyBase.Update(gameTime);

        Vector2 directionToPlayer =
            Player.BaseSprite.WorldCenter -
            EnemyBase.WorldCenter;
        directionToPlayer.Normalize();

        EnemyClaws.WorldLocation = EnemyBase.WorldLocation;
        EnemyClaws.RotateTo(directionToPlayer);
    }
}

public void Draw(SpriteBatch spriteBatch)
{
    if (!Destroyed)
    {
        EnemyBase.Draw(spriteBatch);
        EnemyClaws.Draw(spriteBatch);
    }
}
#endregion
```

What just happened?

The Enemy class is similar in structure to the Player class. An enemy is composed of two different Sprites that will be overlaid to create the final enemy image. Enemies move at a slightly slower speed than the player.

Between frames, each Enemy will keep track of the square it is currently trying to reach in the `currentTargetSquare` vector.

When the `Enemy` constructor runs, the base `Sprite` is created using the parameters passed to the constructor, while the `Sprite` representing the enemy robot's claws is created by offsetting the initial frame down one row of images on the sprite sheet.

Both the `Update()` and the `Draw()` methods only take action if the enemy has not been destroyed. During `Update()`, the `determineMoveDirection()` method (which we have not yet written) is called. The job of this method will be to return a vector representing the direction that the enemy should be moving in during this frame.

Given that information, the `Enemy` is moved in the same way that the `Player` is moved. The `EnemyBase` `Sprite` is rotated to the direction of movement, and its velocity is set taking the `EnemySpeed` variable into account.

When the `EnemyClaws` `Sprite` is updated, it is rotated to face the player's current position, resulting in the enemy always reaching for the player.

Moving enemies

Before we can successfully compile the `Enemy` class, we need to fill in the missing `determineMoveDirection()` method. This method, along with its two helper methods, will utilize the `PathFinder` class to determine what direction the enemy tank should move in order to chase down the player.

Time for action – enemy AI methods

1. Add movement-related methods to the `Enemy` class:

```
#region AI Methods
private Vector2 determineMoveDirection()
{
    if (reachedTargetSquare())
    {
        currentTargetSquare = getNewTargetSquare();
    }

    Vector2 squareCenter = TileMap.GetSquareCenter(
        currentTargetSquare);

    return squareCenter - EnemyBase.WorldCenter;
}

private bool reachedTargetSquare()
{
    return (
        Vector2.Distance(
```

```
        EnemyBase.WorldCenter,
        TileMap.GetSquareCenter(currentTargetSquare))
    <= 2);
}

private Vector2 getNewTargetSquare()
{
    List<Vector2> path = Pathfinder.FindPath(
        TileMap.GetSquareAtPixel(EnemyBase.WorldCenter),
        TileMap.GetSquareAtPixel(Player.BaseSprite.WorldCenter));

    if (path.Count > 1)
    {
        return new Vector2(path[1].X, path[1].Y);
    }
    else
    {
        return TileMap.GetSquareAtPixel(
            Player.BaseSprite.WorldCenter);
    }
}

#endregion
```

What just happened?

The first thing `determineMoveDirection()` does is call `reachedTargetSquare()` to decide whether the enemy tank has reached the square represented by `currentTargetSquare` or not. This determination is made by taking the distance between the center of the target square and the center of the enemy sprite. If the distance is within two pixels, we assume that we have reached the target square.

If the target square has been reached, we need to decide what the new target square should be, based on the latest positions of both the enemy and the player.

The `getNewTargetSquare()` method uses the `PathFinder.FindPath()` method, passing it the current location of the enemy and the location of the player. The list that is returned represents the shortest path between the two.

If there is more than one entry in the path list, we know we have not reached the player, so we return the second element in the path list (the first element, element number zero will be the location the enemy tank currently resides in).

If only one entry exists in the path list, then the starting and ending squares are the same square, meaning that we have reached the player. We can simply return the player's current square as the destination square.

As implemented, our enemy AI will always move in complete squares, from the center of one square towards the center of the next square along its path. Every time a new square center has been reached, the whole path is re-evaluated since the player may have moved while the enemy was making its way to the current square.

Since our enemies move based on the results of the Pathfinder class, we do not even need to check for enemy-to-wall collisions, because the path-finding code will not return squares that contain wall tiles as potential destinations for our enemy tanks.

The enemy manager

Our EnemyManager class will be very straightforward. Its only responsibilities are to track the active Enemy objects in the game and add them when requested. We will also need to update the ComputerTerminals class to actually trigger enemy spawns.

Time for action – the enemy manager

1. Add a new class called EnemyManager to the Robot Rampage project.

2. Add the following using directives to the top of the class file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Modify the declaration of the class to make it a static class:

```
static class EnemyManager
```

4. Add declarations to the EnemyManager class:

```
#region Declarations
public static List<Enemy> Enemies = new List<Enemy>();
public static Texture2D enemyTexture;
public static Rectangle enemyInitialFrame;
public static int MaxActiveEnemies = 30;
#endregion
```

5. Add the Initialize() method to the EnemyManager class:

```
#region Initialization
public static void Initialize(
    Texture2D texture,
    Rectangle initialFrame)
```

```
{
    enemyTexture = texture;
    enemyInitialFrame = initialFrame;
}
#endregion
```

7. Add the AddEnemy() method to the EnemyManager class:

```
#region Enemy Management
public static void AddEnemy(Vector2 squareLocation)
{
    int startX = (int)squareLocation.X;
    int startY = (int)squareLocation.Y;

    Rectangle squareRect =
        TileMap.SquareWorldRectangle(startX, startY);

    Enemy newEnemy = new Enemy(
        new Vector2(squareRect.X, squareRect.Y),
        enemyTexture,
        enemyInitialFrame);

    newEnemy.currentTargetSquare = squareLocation;
    Enemies.Add(newEnemy);
}
#endregion
```

8. Add the Update() and Draw() methods to the EnemyManager Class:

```
#region Update and Draw
public static void Update(GameTime gameTime)
{
    for (int x = Enemies.Count - 1; x >= 0; x--)
    {
        Enemies[x].Update(gameTime);
        if (Enemies[x].Destroyed)
        {
            Enemies.RemoveAt(x);
        }
    }
}

public static void Draw(SpriteBatch spriteBatch)
{
    foreach (Enemy enemy in Enemies)
    {
```

```

        enemy.Draw(spriteBatch);
    }
}
#endregion

```

- 9.** In the `ComputerTerminal` class, replace the `Update()` method with the following:

```

public void Update(GameTime gameTime)
{
    if (Active)
    {
        float elapsed = (float)gameTime.ElapsedGameTime.
            TotalSeconds;

        LastSpawnCounter += elapsed;
        if (LastSpawnCounter > minSpawnTime)
        {
            if (Vector2.Distance(activeSprite.WorldCenter,
                Player.BaseSprite.WorldCenter) > 128)
            {
                if (EnemyManager.Enemies.Count <
                    EnemyManager.MaxActiveEnemies)
                {
                    EnemyManager.AddEnemy(MapLocation);
                    LastSpawnCounter = 0;
                }
            }
        }

        activeSprite.Update(gameTime);
    }
    else
    {
        inactiveSprite.Update(gameTime);
    }
}

```

- 10.** In the `LoadContent()` method of the `Game1` class, initialize the `EnemyManager`:

```

EnemyManager.Initialize(
    spriteSheet,
    new Rectangle(0, 160, 32, 32));

```

- 11.** In the `Update()` method of the `Game1` class, update the `EnemyManager` right after the `GoalManager` had been updated:

```
EnemyManager.Update(gameTime);
```

- 12.** In the `Draw()` method of the `Game1` class, draw the `EnemyManager` right after the `GoalManager` has been drawn:

```
EnemyManager.Draw(spriteBatch);
```

- 13.** Execute `Robot Rampage`. The computer terminals now spawn robots, which pursue your player tank:



What just happened?

Most of the `EnemyManager` class is standard for our managers. When an enemy is added via the `AddEnemy()` method, the enemy's `currentTargetSquare` member is set to the square that the enemy has been spawned in, meaning that on its first `Update()` cycle, it will execute `PathFinder.FindPath()` to determine a path towards the player.

Updating the WeaponManager

It sure would be nice to be able to use our fancy weaponry to destroy those enemy tanks! We have all of the pieces in place to implement this capability, so let's tie them together and allow enemies to be blown to bits.

Time for action – destroying enemies

1. Add `checkShotEnemyImpacts()` and `checkRocketSplashDamage()` to the Collision Detection region of the `WeaponManager` class:

```
private static void checkShotEnemyImpacts(Sprite shot)
{
    if (shot.Expired)
    {
        return;
    }

    foreach (Enemy enemy in EnemyManager.Enemies)
    {
        if (!enemy.Destroyed)
        {
            if (shot.IsCircleColliding(
                enemy.EnemyBase.WorldCenter,
                enemy.EnemyBase.CollisionRadius))
            {
                shot.Expired = true;
                enemy.Destroyed = true;
                if (shot.Frame == 0)
                {
                    EffectsManager.AddExplosion(
                        enemy.EnemyBase.WorldCenter,
                        enemy.EnemyBase.Velocity / 30);
                }
            }
            else
            {
                if (shot.Frame == 1)
                {
                    createLargeExplosion(shot.WorldCenter);
                    checkRocketSplashDamage(shot.WorldCenter);
                }
            }
        }
    }
}
```



```
        }
    }
}

private static void checkRocketSplashDamage(Vector2 location)
{
    int rocketSplashRadius = 40;
    foreach (Enemy enemy in EnemyManager.Enemies)
    {
        if (!enemy.Destroyed)
        {
            if (enemy.EnemyBase.IsCircleColliding(
                location, rocketSplashRadius))
            {
                enemy.Destroyed = true;
                EffectsManager.AddExplosion(
                    enemy.EnemyBase.WorldCenter,
                    Vector2.Zero);
            }
        }
    }
}
```

- 2.** In the `checkShotWallImpacts()` method of the `WeaponManager` class, add a call to `checkRocketSplashDamage()` to allow rockets impacting walls to destroy nearby enemy tanks. Place this line after the call to `createLargeExplosion()`.
`checkRocketSplashDamage(shot.WorldCenter);`
- 3.** In the `Update()` method of the `WeaponManager` class, add a call to check for collisions between shots and enemy tanks right after the existing call to `checkShotWallImpacts()`:
`checkShotEnemyImpacts(Shots[x]);`
- 4.** Execute the Robot Rampage game and blow away the bad guys.

What just happened?

When a shot collides with an enemy, the shot's `Frame` property is checked. Recall that we use a zero `Frame` value for regular shots, and a one value for rockets. If the value indicates that the shot is a normal shot, the `AddExplosion()` method is called and given the location of the enemy tank.

If the shot is a rocket, the `createLargeExplosion()` method is executed, and followed up with a call to `checkRocketSplashDamage()`. Any enemies within 40 pixels of the detonation point of the rocket are destroyed, and their own small explosion effects triggered.

In either case (rocket or normal shot), the shot is marked as `Expired`, and any enemies destroyed by the shot are marked as `Destroyed`.

Game structure

The pieces of our game are all in place, so our final task is to wrap them in a game playing structure. The structure here will be deliberately simple, and you can expand upon it as you desire.

Time for action – the GameManager class

1. Add a new class called `GameManager` to the Robot Rampage project.
2. Add the following `using` directive to the top of the `GameManager` class:

```
using Microsoft.Xna.Framework;
```

3. Modify the declaration of the class to make it a static class:

```
static class GameManager
```

4. Add declarations to the `GameManager` class:

```
#region Declarations
public static int Score = 0;
public static int CurrentWave = 0;
public static int BaseTerminalCount = 8;
public static int MaxTerminalCount = 15;
public static int CurrentTerminalCount = 8;
public static Vector2 PlayerStartLoc = new Vector2(32, 32);
#endregion
```

5. Add public methods to the GameManager class:

```
#region Public Methods
public static void StartNewWave()
{
    CurrentWave++;
    if (CurrentTerminalCount < MaxTerminalCount)
    {
        CurrentTerminalCount++;
    }

    Player.BaseSprite.WorldLocation = PlayerStartLoc;
    Camera.Position = Vector2.Zero;
    WeaponManager.CurrentWeaponType =
        WeaponManager.WeaponType.Normal;
    WeaponManager.Shots.Clear();
    WeaponManager.PowerUps.Clear();
    EffectsManager.Effects.Clear();
    EnemyManager.Enemies.Clear();
    TileMap.GenerateRandomMap();
    GoalManager.GenerateComputers(CurrentTerminalCount);
}

public static void StartNewGame()
{
    CurrentWave = 0;
    Score = 0;
    StartNewWave();
}
#endregion
```

What just happened?

The GameManager class will handle tracking the player's score and the wave (or level) the player is currently on. The StartNewWave() method resets all of the lists and control values to their defaults for a new level, generates a random map, and then generates computer terminals on the map.

The StartNewGame() method simply resets the CurrentWave and Score values, and then calls StartNewWave() to begin the first wave.

Keeping score

As our game stands, we do not award points to the player for their actions. We will change this by adding the necessary code to the `WeaponManager` and `GoalManager` classes.

Time for action – awarding points

1. In the `checkShotEnemyImpacts()` method of the `WeaponManager` class, add the following line after the line that reads `enemy.Destroyed = true;`
`GameManager.Score += 10;`
2. In the `checkRocketSplashDamage()` method of the `WeaponManager` class, add the following line after the line that reads `enemy.Destroyed = true;`
`GameManager.Score += 10;`
3. In the `DetectShutdowns()` method of the `GoalManager` class, add the following line right after the line that reads `activeCount--;`
`GameManager.Score += 100;`

What just happened?

Since we can destroy enemies with either normal shots or rocket splash damage, we need to award points in either case. Shutting down a computer is worth ten times as many points as destroying a single enemy.

Updating Game1

Finally, to wrap our game in a game structure, we need to build it into the `Game1` class. We will include a game state tracker much like we used in *Asteroid Belt Assault*, with a similar division of code in the `Update()` and `Draw()` methods depending on the current game state.

Time for action – updating the Game1 class

1. Add the following declarations to the `Game1` class:

```
enum GameStates {TitleScreen, Playing, WaveComplete, GameOver};
GameStates gameState = GameStates.TitleScreen;

float gameOverTimer = 0.0f;
float gameOverDelay = 6.0f;

float waveCompleteTimer = 0.0f;
float waveCompleteDelay = 6.0f;
```

- 2.** Remove the call to `GoalManager.GenerateComputers()` from the `LoadContent()` method of the `Game1` class.

- 3.** Add the `checkPlayerDeath()` method to the `Game1` class:

```
private void checkPlayerDeath()
{
    foreach (Enemy enemy in EnemyManager.Enemies)
    {
        if (enemy.EnemyBase.IsCircleColliding(
            Player.BaseSprite.WorldCenter,
            Player.BaseSprite.CollisionRadius))
        {
            gameState = GameStates.GameOver;
        }
    }
}
```

- 4.** Replace the existing `Update()` method of the `Game1` class with the following:

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

    switch (gameState)
    {
        case GameStates.TitleScreen:
            if ((GamePad.GetState(PlayerIndex.One).Buttons.A ==
                ButtonState.Pressed) ||
                (Keyboard.GetState().IsKeyDown(Keys.Space)))
            {
                GameManager.StartNewGame();
                gameState = GameStates.Playing;
            }
            break;

        case GameStates.Playing:
            Player.Update(gameTime);
            WeaponManager.Update(gameTime);
            EnemyManager.Update(gameTime);
            EffectsManager.Update(gameTime);
    }
}
```

```

        GoalManager.Update(gameTime);
        if (GoalManager.ActiveTerminals == 0)
        {
            gameState = GameStates.WaveComplete;
        }
        break;

    case GameStates.WaveComplete:
        waveCompleteTimer +=
            (float)gameTime.ElapsedGameTime.TotalSeconds;
        if (waveCompleteTimer > waveCompleteDelay)
        {
            GameManager.StartNewWave();
            gameState = GameStates.Playing;
            waveCompleteTimer = 0.0f;
        }
        break;

    case GameStates.GameOver:
        gameOverTimer +=
            (float)gameTime.ElapsedGameTime.TotalSeconds;
        if (gameOverTimer > gameOverDelay)
        {
            gameState = GameStates.TitleScreen;
            gameOverTimer = 0.0f;
        }
        break;
    }

    base.Update(gameTime);
}

```

5. Replace the current `Draw()` method of the `Game1` class with the following:

```

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

    spriteBatch.Begin();

    if (gameState == GameStates.TitleScreen)
    {
        spriteBatch.Draw(
            titleScreen,
            new Rectangle(0, 0, 800, 600),

```

```
        Color.White);
    }

    if ((gameState == GameStates.Playing) ||
        (gameState == GameStates.WaveComplete) ||
        (gameState == GameStates.GameOver))
    {
        TileMap.Draw(spriteBatch);
        WeaponManager.Draw(spriteBatch);
        Player.Draw(spriteBatch);
        EnemyManager.Draw(spriteBatch);
        EffectsManager.Draw(spriteBatch);
        GoalManager.Draw(spriteBatch);

        checkPlayerDeath();

        spriteBatch.DrawString(
            pericles14,
            "Score: " + GameManager.Score.ToString(),
            new Vector2(30, 5),
            Color.White);

        spriteBatch.DrawString(
            pericles14,
            "Terminals Remaining: " +
                GameManager.ActiveTerminals,
            new Vector2(520, 5),
            Color.White);
    }

    if (gameState == GameStates.WaveComplete)
    {
        spriteBatch.DrawString(
            pericles14,
            "Beginning Wave " +
                (GameManager.CurrentWave+1).ToString(),
            new Vector2(300, 300),
            Color.White);
    }

    if (gameState == GameStates.GameOver)
    {
        spriteBatch.DrawString(
            pericles14,
```

```

        "G A M E O V E R!",
        new Vector2(300,300),
        Color.White);
    }

    spriteBatch.End();

    base.Draw(gameTime);
}

```

6. Execute your game!

What just happened?

Just as in Asteroid Belt Assault, we track a `gameState` variable that determines what happens during each `Update()` and `Draw()` cycle. The value of `GoalManager.ActiveComputers` is checked during each update cycle, and when the player completes a wave, there is a delay while **Beginning Wave...** is displayed, followed by a switch back to `GameStates.Playing`.

While playing, the `checkPlayerDeath()` method is executed during each cycle, checking to see if the player has collided with any enemy tanks. If so, the game state switches to `GameStates.GameOver`, which has its own delay before switching back to the `TitleScreen` state.

Have a go hero

There are a number of options for expanding on Robot Rampage. Here are a few suggestions:

- ◆ Robot Rampage is decidedly silent! Import the sound system from Asteroid Belt Assault and add new sound effects for Robot Rampage. Update the code to play sound effects when appropriate.
- ◆ As stated in the text above, the game state management system is fairly basic. Implement multiple player lives, allowing them to regenerate after they have been destroyed.
- ◆ Add a new weapon to the player's arsenal! Create a weapon that fires a large burst of 12 projectiles in an arc in front of the player tank, but give their `Particles` a very short duration, so the shots will destroy a host of enemy tanks close up, but will not reach more than a few map squares away from the player.

Summary

Robot Rampage covers some important ground in the form of path finding and providing a basis for a very simple form of artificial intelligence for the enemy tanks. In addition, we looked at topics such as:

- ◆ Expanding the visual effects system we introduced in Asteroid Belt Assault
- ◆ Allowing the player to use different types of weaponry
- ◆ Adding enemy robots with the ability to chase the player through the tile-based game map
- ◆ Implementing goals for the player to accomplish to complete a game level

8

Gemstone Hunter – Put on Your Platform Shoes

Gemstone Hunter places the player in the role of an adventuring, treasure hunting archeologist, roaming the wilderness collecting gemstones while avoiding zombies, skeletons, and mummies.

Unlike the other games presented in this book, Gemstone Hunter will not automatically generate levels. For this reason, the game can be viewed more along the lines of a project starter to explore topics like combining Windows Forms and XNA to produce a level editor, and an introduction to the basics of building a platform-style game.

The XNA Creator's Club website provides a number of "Starter Kits" that contain sample code and images to get you started on developing specific types of games. The Platform Starter Kit was introduced with XNA 3.1 and included in the XNA Game Studio distribution. We are going to make use of some of the graphics from this Starter Kit in Gemstone Hunter. We will not use the actual starter kit itself, however, as we want to build on some of the code we have established in our prior games and focus on key aspects of the platform genre.

In this chapter, we will begin the development of Gemstone Hunter by building a map editor for the game. In order to do so, we will cover:

- ◆ Expanding our existing tile map engine
- ◆ Adding multiple projects to a Visual Studio solution
- ◆ Adding a Windows Form to an XNA game project

- ◆ Communicating between Windows Forms and our game code
- ◆ Loading and saving map files via serialization

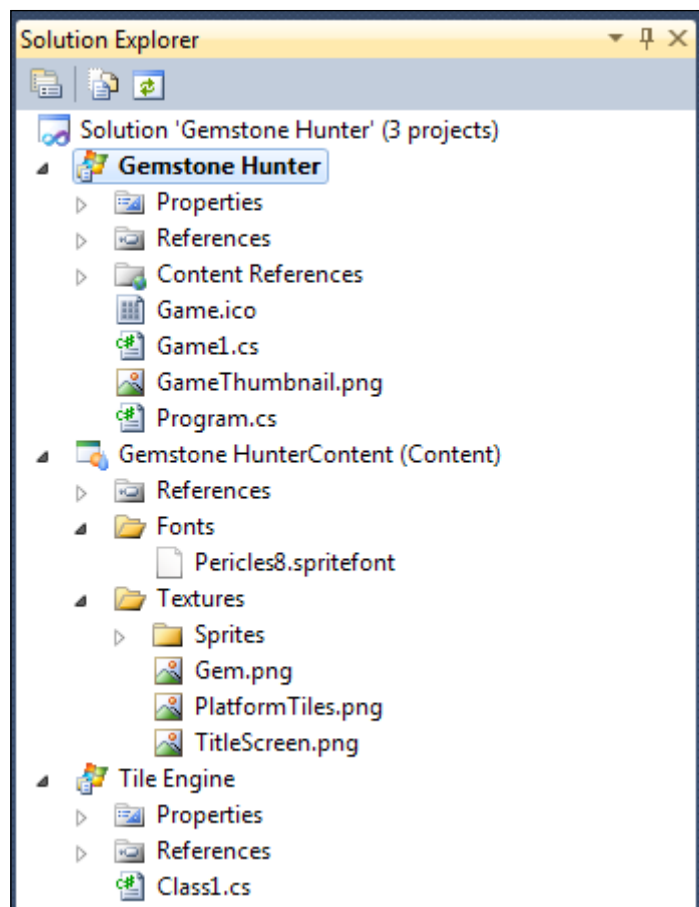
Borrowing graphics

We will not be directly using XNA's Platform Starter Kit, but we will be borrowing the graphical resources for the player's character and enemy monsters from the project. We will begin by creating the project that will eventually house our completed game:



Time for action – creating projects

1. Download `0669_08_GRAPHICPACK.zip` from the book's website and extract the contents to a temporary folder. Leave this folder open in Windows Explorer.
2. Inside Visual Studio, select **New Project...** from the **File** menu.
3. Create a new Windows Game (4.0) project called **Gemstone Hunter**.
4. Right-click on the **Gemstone Hunter Content** project and add a new folder called **Textures**.
5. Switch back to the Windows Explorer window and highlight the **Sprites** folder, as well as the three **.PNG** files (**Gem.png**, **PlatformTiles.png**, and **TitleScreen.png**). Right-click on one of the highlighted files and select **Copy**.
6. Switch back to the Visual Studio window and right-click on the **Textures** folder in the content project and select **Paste**. This should result in a **Sprites** folder under **Textures** folder and all three images being added to your project.
7. Right-click on the **Gemstone Hunter Content** project and add a new folder called **Fonts**.
8. Create a new **SpriteFont** object called **Pericles8** in the **Fonts** folder. Set the font name to **Pericles** and the size to **8**.
9. At the top of the Solution Explorer window, right-click on the solution (**Solution 'Gemstone Hunter'**) and select **Add | New Project...**
10. From the new project window, select the **Windows Game Library (4.0)** template. Name the project **Tile Engine** and add it to the solution:

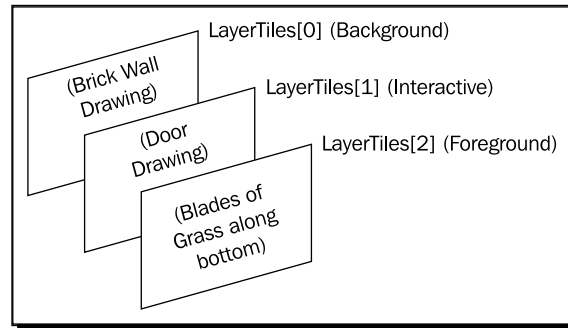


What just happened?

We now have three projects in our solution—The Gemstone Hunter game project, the associated Content project, and the Tile Engine "Game Library" project. The game project itself will be detailed in *Chapter 9*. The "Tile Engine" project will contain the code for, not surprisingly, the game's tile engine, which will be shared with the third project, the "Level Editor" that we will create shortly.

A more advanced tile engine

In Robot Rampage, we built a simple tile engine that displayed a single layer of tiles from a two dimensional array of integers, which represented the tiles associated with each map square. For Gemstone Hunter, we will construct a new tile engine that handles multiple tile layers, including a layer of tiles that are drawn in the foreground, appearing in front of the player:



Since we need to store more information about an individual map square, we will begin by defining a class that will contain all of the information we need about a particular square.

Time for action – the MapSquare class

1. Add a new class called `MapSquare.cs` to the "Tile Engine" project.
2. Add the following using directives to the `MapSquare` class:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Graphics;
```
3. Modify the declaration of the `MapSquare` class by adding the `[Serializable]` attribute before the class declaration, and making the class public:

```
[Serializable]  
public class MapSquare
```

4. Add the declarations region to the `MapSquare` class:

```
#region Declarations  
public int[] LayerTiles = new int[3];  
public string CodeValue = "";  
public bool Passable = true;  
#endregion
```

5. Add a constructor to the MapSquare class:

```
#region Constructor
public MapSquare(
    int background,
    int interactive,
    int foreground,
    string code,
    bool passable)
{
    LayerTiles[0] = background;
    LayerTiles[1] = interactive;
    LayerTiles[2] = foreground;
    CodeValue = code;
    Passable = passable;
}
#endregion
```

6. Add the TogglePassable() method to the MapSquare class:

```
#region Public Methods
public void TogglePassable()
{
    Passable = !Passable;
}
#endregion
```

What just happened?

In step three, we added a new type of information to our code file called an attribute. Attributes, indicated by text in square brackets, indicate to the C# compiler that the item following it should be treated differently in some way compared to the norm for that item type.

In our case, we want to be able to save the tile map that we will be creating to a file and then load that file in both the editor and the game projects. To do this, we need to indicate to C# that the object is serializable—that is, the object can be converted into a byte-stream, which can be stored and reloaded at a later point.



Modifying serializable classes

We need to be fairly sure we have covered all of the bases with our MapSquare class, because once we have used the class to save map files, any changes made (even renaming variables) will cause reloading the saved data to either fail completely, or return corrupted results. This is because the binary serialization method we will be using essentially grabs the in-memory representation of the data and saves it to disk, reloading it in the same manner. If the definition of the structure changes in any way, the binary format that was output will not match the new in-memory format, resulting in unpredictable problems with our loaded maps.

The `LayerTiles` array stores three integer values representing the tile images on background, interactive, and foreground layers. Depending on the needs of your game, you could create any number of layers. When we draw them, we will use the index of the layer to determine the depth at which the sprite is drawn, ensuring that they appear in the correct order on the screen.

While building our map, we can associate a string value with each individual map square by setting the `CodeValue` variable. This variable will allow us to create special features such as map transitions, traps, and invisible barriers to enemy movement.

Finally, to determine if the map square blocks player movement we can check the `Passable` member variable. If it is false, the player cannot move into this square. The default for all MapSquares is to be open to movement (`Passable` set to `true`).

Rebuilding the camera

Just as we did in Robot Rampage, we will use a `Camera` object to represent the player's view of our game world, and track all object positions in world-based coordinates.

Our Camera class is pulled directly from Robot Rampage, with one minor addition:

Time for action – the Camera class

1. Create a new class called `Camera.cs` in the "Tile Engine" project.
2. Add the following `using` directive to the top of the class file:

```
using Microsoft.Xna.Framework;
```
3. Modify the declaration of the `Camera` class to make it public and static:

```
public static class Camera
```
4. Add declarations to the `Camera` class:

```
#region Declarations
```

```

private static Vector2 position = Vector2.Zero;
private static Vector2 viewPortSize = Vector2.Zero;
private static Rectangle worldRectangle = new Rectangle
    (0, 0, 0, 0);
#endregion

```

5. Add properties to the Camera class:

```

#region Properties
public static Vector2 Position
{
    get { return position; }
    set
    {
        position = new Vector2(
            MathHelper.Clamp(value.X,
                worldRectangle.X, worldRectangle.Width -
                ViewPortWidth),
            MathHelper.Clamp(value.Y,
                worldRectangle.Y, worldRectangle.Height -
                ViewPortHeight));
    }
}

public static Rectangle WorldRectangle
{
    get { return worldRectangle; }
    set { worldRectangle = value; }
}

public static int ViewPortWidth
{
    get { return (int)viewPortSize.X; }
    set { viewPortSize.X = value; }
}

public static int ViewPortHeight
{
    get { return (int)viewPortSize.Y; }
    set { viewPortSize.Y = value; }
}

public static Rectangle ViewPort
{
    get
    {
        return new Rectangle(

```



```
        (int)Position.X, (int)Position.Y,
        ViewPortWidth, ViewPortHeight);
    }
}
#endregion
```

6. Add methods to the Camera class:

```
#region Public Methods
public static void Move(Vector2 offset)
{
    Position += offset;
}

public static bool ObjectIsVisible(Rectangle bounds)
{
    return (ViewPort.Intersects(bounds));
}

public static Vector2 WorldToScreen(Vector2 worldLocation)
{
    return worldLocation - position;
}

public static Rectangle WorldToScreen(Rectangle worldRectangle)
{
    return new Rectangle(
        worldRectangle.Left - (int)position.X,
        worldRectangle.Top - (int)position.Y,
        worldRectangle.Width,
        worldRectangle.Height);
}

public static Vector2 ScreenToWorld(Vector2 screenLocation)
{
    return screenLocation + position;
}

public static Rectangle ScreenToWorld(Rectangle screenRectangle)
{
    return new Rectangle(
        screenRectangle.Left + (int)position.X,
        screenRectangle.Top + (int)position.Y,
        screenRectangle.Width,
        screenRectangle.Height);
}
#endregion
```

What just happened?

The `Transform()` methods from the Robot Rampage Camera have been renamed to `WorldToScreen()`, and a new pair of methods called `ScreenToWorld()` have been added. We will need to respond to mouse events in the map editor, and the mouse position is reported in screen coordinates. These new methods will assist in determining the map square underneath the mouse cursor.

We saw in Robot Rampage that converting world coordinates to screen coordinates was a simple matter of subtracting the position of the camera from the coordinate. The reverse is also true. To convert from a screen coordinate to a world map coordinate, we add the position of the camera.

Constructing the tile engine

The basic concepts behind our original tile-based map engine are unchanged, and indeed most of the code for our new tile engine can be brought over from Robot Rampage. We need to make modifications to the way the map itself is stored, and the way we access (for both reading and setting) map tiles to accommodate the `MapSquare` class instead of a simple array of integers.

Time for action – the `TileMap` class – part 1

1. In the Tile Engine project, rename the `Class1.cs` file that was generated by the Game Library project template to `TileMap.cs`. Visual Studio will ask you if you wish to rename all references to the class as well. Go ahead and click on **Yes**. We have not referenced our new class anywhere, so no other code is actually updated.
2. Double-click on the new `TileMap.cs` file to open it in the Visual Studio editor.
3. Add the following `using` directives to the top of the class file:


```
using Microsoft.Xna.Framework.Storage;
using System.IO;
using System.Xml.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```
4. Modify the declaration of the `TileMap` class to make the class public and static:


```
public static class TileMap
```
5. Add declarations to the `TileMap` class:


```
#region Declarations
public const int TileWidth = 48;
public const int TileHeight = 48;
public const int MapWidth = 160;
```

```
public const int MapHeight = 12;
public const int MapLayers = 3;
private const int skyTile = 2;

static private MapSquare[,] mapCells =
    new MapSquare[MapWidth, MapHeight];

public static bool EditorMode = false;

public static SpriteFont spriteFont;
static private Texture2D tileSheet;
#endregion
```

6. Add the Initialize() method to the TileMap class:

```
#region Initialization
static public void Initialize(Texture2D tileTexture)
{
    tileSheet = tileTexture;
    for (int x = 0; x < MapWidth; x++)
    {
        for (int y = 0; y < MapHeight; y++)
        {
            for (int z=0; z < MapLayers; z++)
            {
                mapCells[x,y] = new MapSquare(skyTile, 0, 0, "", true);
            }
        }
    }
}
#endregion
```

7. Add the Tile and Tile Sheet Handling region to the TileMap class:

```
#region Tile and Tile Sheet Handling
public static int TilesPerRow
{
    get { return tileSheet.Width / TileWidth; }
}

public static Rectangle TileSourceRectangle(int tileIndex)
{
    return new Rectangle(
        (tileIndex % TilesPerRow) * TileWidth,
        (tileIndex / TilesPerRow) * TileHeight,
        TileWidth,
        TileHeight);
}
#endregion
```

What just happened?

Before we continue with the implementation of the `TileMap` class, let's look at the difference we can see so far from the same class in *Robot Rampage*.

We have not used the `System.Runtime.Serialization.Formatters.Binary` assembly before, but in order to write our level files out to disk (and read them back in again) we need a way to convert the array of `MapSquare` objects in memory into a format that can be written to a file. This assembly provides a formatter object that can perform this translation.

You may notice that both the declaration and initialization regions are quite a bit shorter than they were in *Robot Rampage*. In *Gemstone Hunter*, we will define a tile sheet image that contains the tiles we will use in the game. Unlike our previous games, we will not simply store all of our game's graphics on the same sprite sheet, but break it up into multiple sheets:



By defining a single sprite sheet to hold only tiles for the tile map, we can treat the sheet in a special manner, and decide that the sheet will be evenly divided into as many tiles as will fit on the image. Our tile sheet image is 480 by 480 pixels, and with a 48 by 48 tile size, we have 10 rows of 10 tiles, for 100 total tiles available to our game. We could always increase the size of the image to add more tiles, though we would want to keep it to increments of 48 pixels in each direction to make the math easier.

We will number the tiles starting with zero in the upper left corner of the tile sheet, and progressing across a row. When we reach the end of the row, we return to the left side of the image and start a new row. The two methods in the `Tile` and `Tile Sheet Handling` region (step seven) replace the array of pre-defined tiles by providing a way to locate the source rectangle for any tile we wish to draw on the tile sheet image.

Our map is still represented by an array, but this time around it is a two-dimensional array of `MapSquare` objects instead of simple integers. We have also rearranged our terminology to reflect the more complex nature of our tile map. What we referred to as squares in *Robot Rampage*, we now call cells. Any of our code that dealt with getting or setting information about map tiles in *Robot Rampage* will need to be updated to handle the entire `MapSquare` object in each cell instead of a simple integer value.

The `TileMap` itself will include support for being used in editing mode, which can be toggled by setting the `EditorMode` member variable. While in editor mode, we will draw the contents of the `CodeValue` member of the `MapSquare` class on top of each square, so the `TileMap` class needs a `SpriteFont` for use with `SpriteBatch.DrawString()`.

Our `Initialize()` method is greatly simplified by the removal of the `tiles` array, allowing us to establish all of the `MapCells` as `MapSquares` with empty tiles on each layer. Our tile sheet contains a fully transparent tile in the upper left corner (tile zero) and a blue sky tile in the third position (tile two), so by filling the map with squares containing tile two on the background layer, and tile zero on the other two layers, we end up with an empty map with a blue sky background. This will simply save us time when creating a new map with the map editor by letting us skip drawing the sky on each map.

Time for action – the `TileMap` class – part 2

1. Add methods dealing with locating map cells to the `TileMap` class:

```
#region Information about Map Cells
static public int GetCellByPixelX(int pixelX)
{
    return pixelX / TileWidth;
}

static public int GetCellByPixelY(int pixelY)
{
    return pixelY / TileHeight;
}

static public Vector2 GetCellByPixel(Vector2 pixelLocation)
{
    return new Vector2(
        GetCellByPixelX((int)pixelLocation.X),
        GetCellByPixelY((int)pixelLocation.Y));
}

static public Vector2 GetCellCenter(int cellX, int cellY)
{
    return new Vector2(
        (cellX * TileWidth) + (TileWidth / 2),
        (cellY * TileHeight) + (TileHeight / 2));
}

static public Vector2 GetCellCenter(Vector2 cell)
{
    return GetCellCenter(
        (int)cell.X,
        (int)cell.Y);
}
```

```
}
static public Rectangle CellWorldRectangle(int cellX, int cellY)
{
    return new Rectangle(
        cellX * TileWidth,
        cellY * TileHeight,
        TileWidth,
        TileHeight);
}
static public Rectangle CellWorldRectangle(Vector2 cell)
{
    return CellWorldRectangle(
        (int)cell.X,
        (int)cell.Y);
}
static public Rectangle CellScreenRectangle(int cellX, int cellY)
{
    return Camera.WorldToScreen(CellWorldRectangle(cellX, cellY));
}
static public Rectangle CellScreenRectangle(Vector2 cell)
{
    return CellScreenRectangle((int)cell.X, (int)cell.Y);
}
static public bool CellIsPassable(int cellX, int cellY)
{
    MapSquare square = GetMapSquareAtCell(cellX, cellY);
    if (square == null)
        return false;
    else
        return square.Passable;
}
static public bool CellIsPassable(Vector2 cell)
{
    return CellIsPassable((int)cell.X, (int)cell.Y);
}
static public bool CellIsPassableByPixel(Vector2 pixelLocation)
{
    return CellIsPassable(
        GetCellByPixelX((int)pixelLocation.X),
        GetCellByPixelY((int)pixelLocation.Y));
}
```

```
static public string CellCodeValue(int cellX, int cellY)
{
    MapSquare square = GetMapSquareAtCell(cellX, cellY);
    if (square == null)
        return "";
    else
        return square.CodeValue;
}

static public string CellCodeValue(Vector2 cell)
{
    return CellCodeValue((int)cell.X, (int)cell.Y);
}
#endregion
```

2. Add methods for manipulating MapSquares to the TileMap class:

```
#region Information about MapSquare objects
static public MapSquare GetMapSquareAtCell(int tileX, int tileY)
{
    if ((tileX >= 0) && (tileX < MapWidth) &&
        (tileY >= 0) && (tileY < MapHeight))
    {
        return mapCells[tileX, tileY];
    }
    else
    {
        return null;
    }
}

static public void SetMapSquareAtCell(
    int tileX,
    int tileY,
    MapSquare tile)
{
    if ((tileX >= 0) && (tileX < MapWidth) &&
        (tileY >= 0) && (tileY < MapHeight))
    {
        mapCells[tileX, tileY] = tile;
    }
}

static public void SetTileAtCell(
    int tileX,
    int tileY,
```

```

        int layer,
        int tileIndex)
    {
        if ((tileX >= 0) && (tileX < MapWidth) &&
            (tileY >= 0) && (tileY < MapHeight))
        {
            mapCells[tileX, tileY].LayerTiles[layer] = tileIndex;
        }
    }

    static public MapSquare GetMapSquareAtPixel(int pixelX, int
pixelY)
    {
        return GetMapSquareAtCell(
            GetCellByPixelX(pixelX),
            GetCellByPixelY(pixelY));
    }

    static public MapSquare GetMapSquareAtPixel(Vector2 pixelLocation)
    {
        return GetMapSquareAtPixel(
            (int)pixelLocation.X,
            (int)pixelLocation.Y);
    }
}
#endregion

```

What just happened?

Much of the code we need for dealing with the tile map is unchanged from our simpler tile engine, aside from the changes to accommodate our new cell and MapSquare terminology. We also use the MapSquare type when getting and setting the contents of map cells instead of integers.

The SetTileAtCell() method may seem out of place among the methods dealing with MapSquare objects. Its purpose is to provide a way to change the tile index of a single layer in a cell without repackaging the cell's entire MapSquare object. By passing SetTileAtCell() a cell location, layer number, and tile index, we can change the content of a single layer—exactly what we will need to do when building the map editor.

Because the game engine will need easy access to the Passable and CodeValue members of a cell (without the need to deal with the tile layer values), we have created the shortcut methods CellIsPassable() and CellCodeValue(). When the time comes to move the player and enemy objects during game play, we will make extensive use of these methods to determine what map squares are accessible to game entities.

Drawing the tile map

We are now ready to assemble the code necessary to draw the enhanced tile map to the screen. We need to account for all three layers of the map, ensuring that each will be drawn in the proper relationship to the others—the background layer appearing furthest away, the interactive layer drawn above it, and finally the foreground layer drawn nearest to the screen.

Time for action – the TileMap class – part 3

1. Add the Draw() method to the TileMap class:

```
#region Drawing
static public void Draw(SpriteBatch spriteBatch)
{
    int startX = GetCellByPixelX((int)Camera.Position.X);
    int endX = GetCellByPixelX((int)Camera.Position.X +
        Camera.ViewPortWidth);

    int startY = GetCellByPixelY((int)Camera.Position.Y);
    int endY = GetCellByPixelY((int)Camera.Position.Y +
        Camera.ViewPortHeight);

    for (int x = startX; x <= endX; x++)
        for (int y = startY; y <= endY; y++)
        {
            for (int z = 0; z < MapLayers; z++)
            {
                if ((x >= 0) && (y >= 0) &&
                    (x < MapWidth) && (y < MapHeight))
                {
                    spriteBatch.Draw(
                        tileSheet,
                        CellScreenRectangle(x,y),
                        TileSourceRectangle(
                            mapCells[x,y].LayerTiles[z]),
                        Color.White,
                        0.0f,
                        Vector2.Zero,
                        SpriteEffects.None,
                        1f - ((float)z * 0.1f));
                }
            }
        }
    if (EditorMode)
    {
        DrawEditModeItems(spriteBatch, x, y);
    }
}
```

```

        }
    }
}

public static void DrawEditModeItems(
    SpriteBatch spriteBatch,
    int x,
    int y)
{
    if ((x < 0) || (x >= MapWidth) ||
        (y < 0) || (y >= MapHeight))
        return;

    if (!CellIsPassable(x,y))
    {
        spriteBatch.Draw(
            tileSheet,
            CellScreenRectangle(x, y),
            TileSourceRectangle(1),
            new Color(255, 0, 0, 80),
            0.0f,
            Vector2.Zero,
            SpriteEffects.None,
            0.0f);
    }

    if (mapCells[x, y].CodeValue != "")
    {
        Rectangle screenRect = CellScreenRectangle(x, y);

        spriteBatch.DrawString(
            spriteFont,
            mapCells[x, y].CodeValue,
            new Vector2(screenRect.X, screenRect.Y),
            Color.White,
            0.0f,
            Vector2.Zero,
            1.0f,
            SpriteEffects.None,
            0.0f);
    }
}

#endregion

```

What just happened?

Once again, the `Draw()` method is very familiar. We still use the position of the camera to determine the range of cells that need to be drawn to the screen, but this time we nest a third loop inside the horizontal and vertical loops, which draws the tiles from each of the three layers.

This `SpriteBatch.Draw()` call is unlike any of the others we have made before, because this time we are specifying a layer depth as the last parameter of the call. In the past, when we have used the advanced form of the `Draw()` method, we have always left this parameter at a value of `0f`.



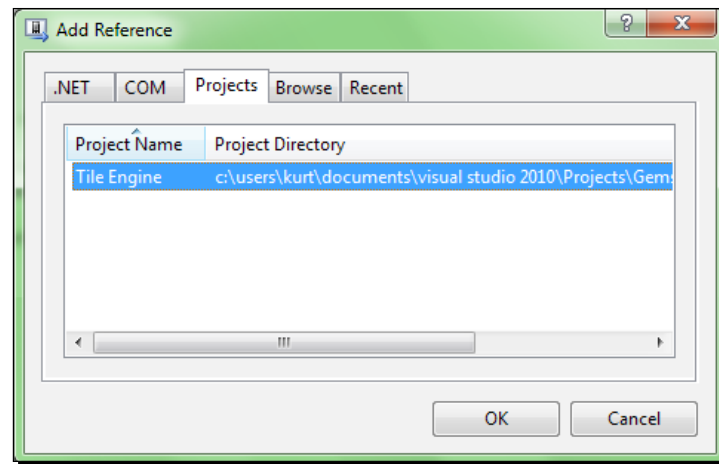
Layer depths

By specifying the layer depth at which to draw the sprite, we can execute the draw calls in any order and allow the graphics card to place them properly according to their sorted depth. In order for this to work, we need to specify some additional parameters to the `SpriteBatch.Begin()` method call we will use in the `Game1.cs` file's `Draw()` method. We need to specify `SpriteSortMode.BackToFront` as the first parameter of the call. This tells the `SpriteBatch` object to pay attention to the layer depth information, otherwise the order in which the sprites are drawn will still be used for sorting, and the layer depths will be ignored. The `FrontToBack` reverses the meaning of the layer depth parameter, making items at `1.0f` closer to the camera than items at `0.0f`. None of the other modes (`Immediate`, `Deferred`, and `Texture`) will allow us to properly sort sprites, because they either rely on the drawing order (`Immediate` and `Deferred`) or group the drawn sprites in order by the source texture they come from (`Texture`).

If our tile engine is currently in editor mode, the `DrawEditModeItems()` method is called after each tile is drawn. This uses the white tile at tile index one to draw a semi-transparent red block over any square that is not passable by the player. Additionally, the method uses `SpriteBatch.DrawString()` to display the content of the `CodeValue` variable associated with each map cell if it is not empty.

Time for action – adding the tile map to the game project

1. Right click on the **Gemstone Hunter** project in Solution Explorer and click on **Add Reference...**
2. Click on the **Projects** tab in the **Add Reference** window and ensure that the **Tile Engine** project is selected. Click on **OK**:



3. Open the **Game1.cs** file in the **Gemstone Hunter** project and add the following using directive to the top of the file:

```
using Tile_Engine;
```

4. Add the following to the `LoadContent()` method of the `Game1` class:

```
TileMap.Initialize(
    Content.Load<Texture2D>(@"Textures\PlatformTiles"));
TileMap.SetTileAtCell(3, 3, 1, 10);

Camera.WorldRectangle = new Rectangle(0, 0, 160 * 48, 12 * 48);
Camera.Position = Vector2.Zero;
Camera.ViewPortWidth = 800;
Camera.ViewPortHeight = 600;
```

4. Replace the current `Draw()` method of the `Game1` class with the following:

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

    spriteBatch.Begin(
        SpriteSortMode.BackToFront,
        BlendState.AlphaBlend);
    TileMap.Draw(spriteBatch);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

5. Execute the project.

What just happened?

By referencing the Tile Engine project from the Gemstone Hunter project, we can utilize the code from the Tile Engine project by including a `using` statement referencing the `Tile_Engine` namespace.

During the `LoadContent()` method, we initialize the `TileMap` and `Camera` classes, and add a single tile to the map so that we will see it when we run the application.

Finally, we see the special form of the `SpriteBatch.Draw()` call we need to make to display the different tile layers sorted in the proper order. In addition to specifying the sort mode, we also need to specify the way transparent sprites are blended together. The default is `BlendState.AlphaBlend`, which is what we want to keep. Since there is no `SpriteBatch.Draw()` call that allows us to specify only the sort mode, we must supply the blend mode even though it is normally the default.

Why go to all of this trouble creating multiple projects and referencing them instead of including the `MapSquare`, `Camera`, and `TileMap` classes directly into the Gemstone Hunter project? Since we are going to need to display the tile engine in both the game and the level editor, splitting it out into its own project and referencing it allows us to create only one set of source files for the map components. If we were to include it directly in the game, we would need to make another copy of those three classes for our map editor project. Any time we need to update the classes, we would need to remember to update them in both places, increasing the chance of introducing errors and inconsistencies into our project.

The map editor project

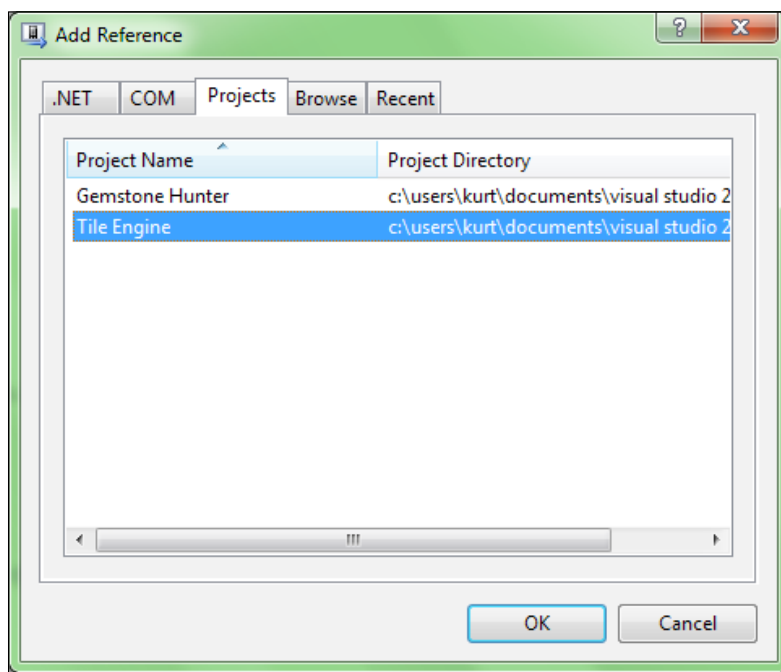
With the tile engine in place, we are now ready to begin building the map editor we will use to create levels for the Gemstone Hunter game. The map editor will combine both an XNA Game and a Windows Forms form to take advantage of the Windows Forms controls (menus, buttons, checkboxes, and so on) to save us the time of recreating all of these controls within XNA.

Creating the map editor project

Since we know that we want to create a Windows Forms application for our level editor, it is tempting to use the Windows Forms application template that is included with Visual C#. However, it is much easier to add a Windows Forms object to an XNA game project than to work the other way around and try to incorporate all of the components of an XNA project into the Windows Forms template.

Time for action – creating the level editor project

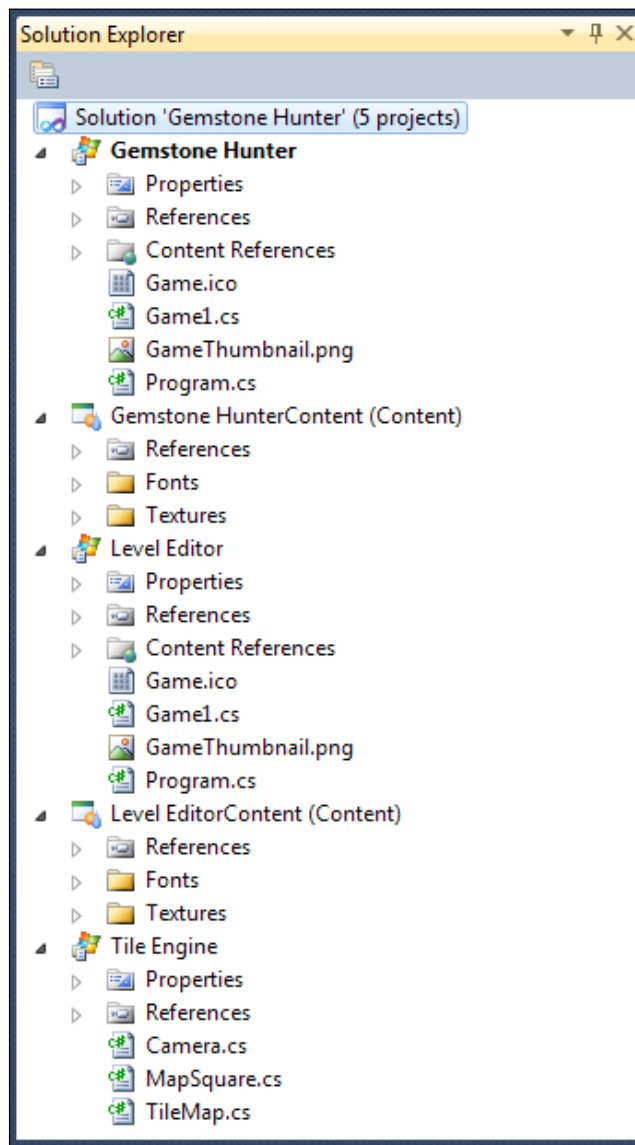
1. In the Solution Explorer window, right-click on the top-most item that reads **Solution 'Gemstone Hunter' (3 Projects)** and select **Add | New Project....**
2. Select the **Windows Game (4.0)** project template.
3. Name the project **Level Editor** and click on **OK**.
4. Right-click on the **Level Editor** project and select **Add Reference....**
5. On the **Projects** tab of the Add Reference window, select **Tile Engine** and click on **OK**:



6. Expand the Gemstone Hunter Content project and right-click on the **Fonts** folder and select **Copy**.
7. Right-click on the Level Editor Content project and select **Paste**.
8. Repeat steps 6 and 7 to copy the **Textures** folder from the Gemstone Hunter Content project to the Level Editor Content project.
9. Right-click on the **Level Editor** project in the Solution Explorer window and click on **Set as StartUp Project**.

What just happened?

Your solution now has five separate projects, the game project (simply called Gemstone Hunter), the game's Content project, the Tile Engine game library, the Level Editor, and the Level Editor's Content project. By setting the Level Editor as the startup project, whenever we execute our code from the development environment, the level editor will be the application that starts (as opposed to starting the actual game). This setting is just a convenience for us while we work on the editor:



Our level editor will use the same content that our real game will use, but we cannot simply add a content reference to the same content project from both "game" projects. XNA Game Studio does allow multiple projects to share a content folder, but only if those projects target different platforms (Windows, Xbox, Windows Phone). Since both of our projects target the Windows platform, we need to duplicate the content project items.

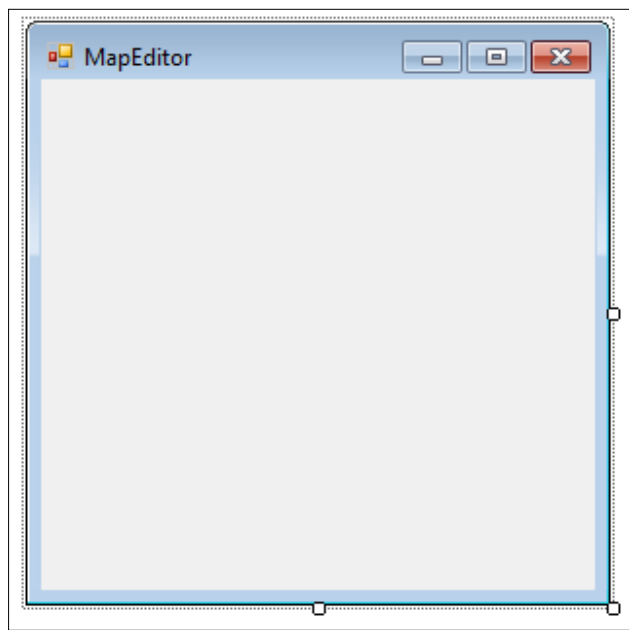
Just like our Gemstone Hunter project, the Level Editor project contains a reference to the Tile Engine project, allowing it to make use of the tile engine code without duplicating it.

Adding a form

We will begin the construction of the level editor by adding a Windows Form to our project and linking it to the XNA Game to allow the output of the game to be displayed on a PictureBox control on the form.

Time for action – adding a form

1. Right-click on the Level Editor project in Solution Explorer and select **Add | Windows Form**.
2. Name the form `MapEditor.cs` and click on the **Add** button.
3. The **MapEditor** form will automatically open in Design mode as a blank window:



4. In the properties window (right-click on the form and select **Properties** if you have hidden the properties window), set the **Size** property to **700, 670** pixels.
5. On the left edge of the screen, open the **Toolbox** panel (**View | Other Windows | Toolbox** if it is hidden) and expand the **All Windows Forms** section. Locate the **MenuStrip** control and drag an instance of it onto the **MapEditor** form. Leave the menu items empty for now.
6. Drag a new **PictureBox** control from the **Toolbox** panel onto the form.
7. Click on the newly created **PictureBox** and set the following properties in the Properties window:
 - a. **Name: pctSurface**
 - b. **Anchor: Top, Bottom, Left, Right**
 - c. **Location: 184, 27**
 - d. **Modifiers : Public**
 - e. **Size: 471, 576**
8. Right-click on the `MapEditor.cs` file in Solution Explorer and select **View Code** to open the source code for the `MapEditor` form.
9. Add the following `using` directives to the `MapEditor.cs` class file:

```
using Microsoft.Xna.Framework;
using System.IO;
using Tile_Engine;
```
10. Add the following declaration to the `MapEditor` class:

```
public Game1 game;
```
11. In the Level Editor project, double-click on the `Program.cs` file and replace the `Main()` method with the following:

```
static void Main(string[] args)
{
    MapEditor form = new MapEditor();
    form.Show();
    form.game = new Game1(
        form.pctSurface.Handle,
        form,
        form.pctSurface);
    form.game.Run();
}
```

- 12.** Still in the Level Editor project, open the `Game1.cs` class file and add the following declarations to the declarations area:

```
IntPtr drawSurface;  
System.Windows.Forms.Form parentForm;  
System.Windows.Forms.PictureBox pictureBox;
```

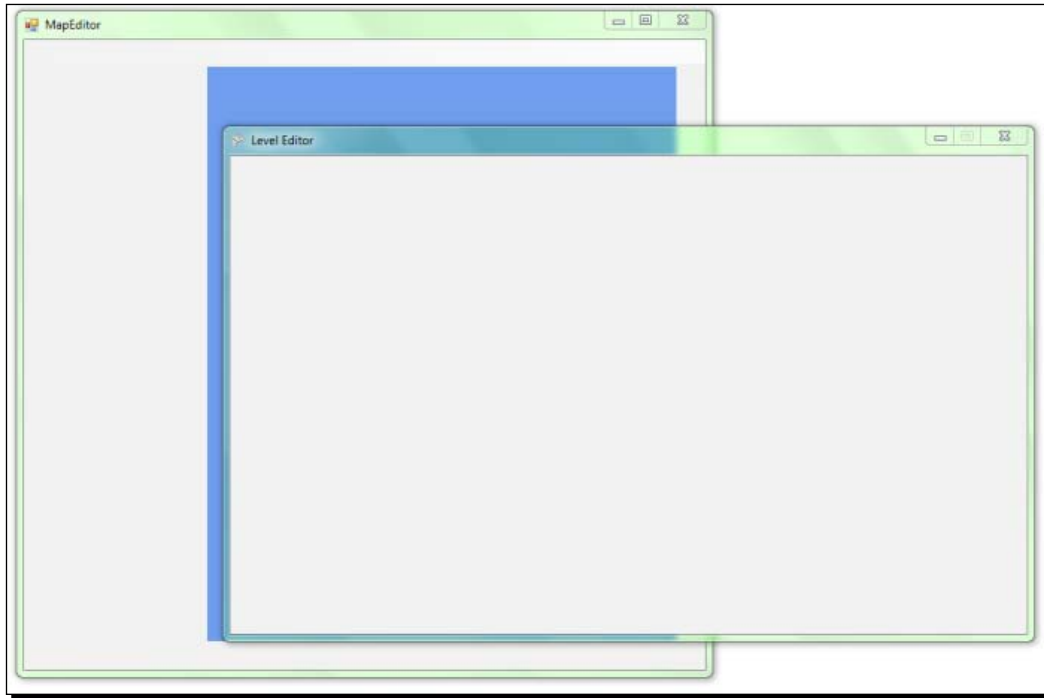
- 13.** Replace the `Game1` constructor with the following:

```
public Game1(IntPtr drawSurface,  
             System.Windows.Forms.Form parentForm,  
             System.Windows.Forms.PictureBox surfacePictureBox)  
{  
    graphics = new GraphicsDeviceManager(this);  
    Content.RootDirectory = "Content";  
  
    this.drawSurface = drawSurface;  
    this.parentForm = parentForm;  
    this.pictureBox = surfacePictureBox;  
  
    graphics.PreparingDeviceSettings +=  
        new EventHandler<PreparingDeviceSettingsEventArgs>(  
            graphics_PreparingDeviceSettings);  
  
    Mouse.WindowHandle = drawSurface;  
}
```

- 14.** Add the `graphics_PreparingDeviceSettings()` event handler to the `Game1` class:

```
void graphics_PreparingDeviceSettings(object sender,  
    PreparingDeviceSettingsEventArgs e)  
{  
  
    e.GraphicsDeviceInformation.PresentationParameters.  
    DeviceWindowHandle = drawSurface;  
  
}
```

15. Execute the project:



What just happened?

As it turns out, we can add a form to an XNA Game project the same way we would add a form to a standard Windows application. The form will not show up by default, however. The `Program.cs` file is the driver behind the C# project, and the `Main()` method gets executed when the application starts.



Other methods

This is not the only method for integrating Windows Forms and XNA. Check out the Winforms samples at the XNA Creators Club website (http://creators.xna.com/en-US/sample/winforms_series1) for additional information. The Creators Club website has many samples on other topics as well.

For a normal XNA Game, the `Main()` method creates an instance of the `Game1` class and calls its `Run()` method. In order to combine our game with a Windows Form, we have altered this start up process.

Instead of creating an instance of the `Game1` class in the `Program` class (where `Main()` lives), we create an instance of our `MapEditor` form class and then create an instance of `Game1` inside the form.

This will simplify addressing components of the `Game1` class from the `MapEditor` form, allowing us to change properties in the game object in response to user interaction with form controls.

We pass the window handle of the `PictureBox` to the `Game1` constructor. The window handle uniquely identifies the display area of `PictureBox`, allowing our code to redirect XNA's drawing commands from its own game window to the area defined by the `PictureBox`.

In order to tell XNA that the graphics we draw should be displayed on the `PictureBox`, we add an event handler to the `PreparingDeviceSettings` event of the `GraphicsDeviceManager` class. In this event handler, we simply set the `DeviceWindowHandle` associated with the graphics device to `drawSurface`, the value that we passed in when the instance of the `Game1` class was created inside `Program.cs`.

The last thing the constructor does is tell XNA's `Mouse` class that it should report coordinates relative to the `PictureBox` that the game will be drawn onto. If we do not make this setting, we will not be able to determine where on our game display the mouse cursor is located.

Alas, we still have a few problems. The most obvious is that there is a big, empty window that shows up on top of our level editor window. This is the window that would normally contain the XNA game. We have moved the output of the XNA drawing commands to a new drawing surface, but the old window still gets created.

The second problem will not be apparent right away, but will cause us trouble when we resize the map editor form while it is running. When the size of the `PictureBox` changes, we need to let XNA know that the back-buffer size has changed, and let our `Camera` class know that the view port on the display has changed as well.

To address both of these problems, we will add additional event handlers to the system events that occur when the visibility of the game's empty window changes, and when the picture box is resized.



Event handlers

Just about everything that happens in Windows happens in response to events. When the user clicks on a button, resizes a window, selects an item from a menu, or any number of other actions, Windows notifies everything that might be impacted by that action that the event has taken place. All Windows Forms controls have event handlers that determine how they respond to those events. In C#, we can easily add an event handler to an object, even if it already has event handlers, using the += notation with the EventHandlers collection on the object.

Time for action – adding event handlers

1. Add the following directive to the Game1 class of the Level Editor project:
`using Tile_Engine;`
2. Add the following declaration to the declarations area of the Game1 class of the Level Editor project:
`System.Windows.Forms.Control gameForm;`
3. Add the following code to the constructor of the Game1 class:

```
gameForm =  
    System.Windows.Forms.Control.FromHandle(this.Window.Handle);  
gameForm.VisibleChanged +=  
    new EventHandler(gameForm_VisibleChanged);  
gameForm.SizeChanged +=  
    new EventHandler(pictureBox_SizeChanged);
```

4. Add the two event handlers to the Game1 class:

```
private void gameForm_VisibleChanged(object sender, EventArgs e)  
{  
    if (gameForm.Visible == true)  
        gameForm.Visible = false;  
}  
  
void pictureBox_SizeChanged(object sender, EventArgs e)  
{  
    if (parentForm.WindowState !=  
        System.Windows.Forms.FormWindowState.Minimized)  
    {  
        graphics.PreferredBackBufferWidth = pictureBox.Width;  
        graphics.PreferredBackBufferHeight = pictureBox.Height;
```

```
        Camera.ViewPortWidth = pictureBox.Width;
        Camera.ViewPortHeight = pictureBox.Height;
        graphics.ApplyChanges();
    }
}
```

5. Execute the application.
6. To end the application, you will need to return to the Visual Studio interface and select **Stop Debugging** from the **Debug** menu.

What just happened?

At first, the `gameForm_VisibleChanged()` method may look odd. It may seem that it would prevent the game from ever being displayed, as it sets the `gameForm.Visible` property to `false` if it ever ends up as `true`.

Remember though, that the form (or window) that is automatically generated by XNA will always be empty. Our game's display is now being redirected to the `PictureBox` on our `MapEditor` form. This means that we really do want to make sure that the game's form is never visible. Whenever its visibility changes, we ensure that `Visible` is set to `false` to keep it from appearing.

When the size of the `PictureBox` changes—since it is anchored to the sides of the `MapEditor` form, resizing the form will resize the `PictureBox`—we want to update the `GraphicsDeviceManager` with the new size of our display area, and pass those updates along to the `Camera` class. We need to be careful to check the `WindowState` of the parent form when processing the resize event. Because the back-buffer width and height must be greater than zero, if we attempt to set them when the form has been minimized, the application will crash.

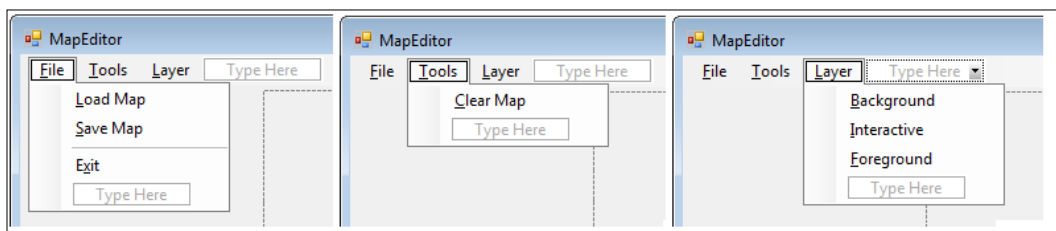
Clicking on the window close button or pressing `Alt + F4` to end the application will no longer work, because the hidden window that `Game1` would normally run in will not close automatically.

Filling Out Our Form

Right now, we just have the familiar blue XNA window being displayed on our form. We need to add a number of other controls to the form to build a functional level editor.

Time for action – creating the menu bar

1. Double-click on the `MapEditor.cs` file in Solution Explorer to open the `MapEditor` form in the design window.
2. Click on the empty `MenuStrip` you previously added to the form, and add menu entries for the following items:
 - **&File**
 - **&Load Map**
 - **&Save Map**
 - – (A single dash, creating a separator line)
 - **E&xit**
 - **&Tools**
 - **&Clear Map**
 - **&Layer**
 - **&Background**
 - **&Interactive**
 - **&Foreground**



3. Double-click on the **Exit** item under the **File** menu to have C# automatically generate an event handler for the Exit menu item.
4. Enter the following code into the `exitToolStripMenuItem_Click()` event handler:

```
game.Exit();  
Application.Exit();
```

What just happened?

We now have a standard Windows menu attached to our form with a few entries for our level editor. In order to add code to menu items other than the **Exit** command, we need to make modifications to our `Game1` class, so we will come back to them after we have laid out all of the items on our display.

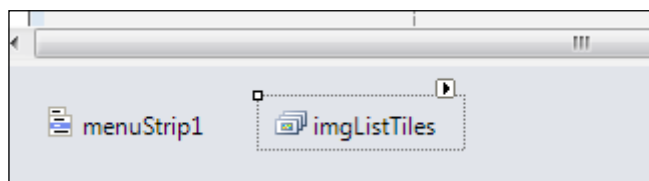


What are all those ampersands?

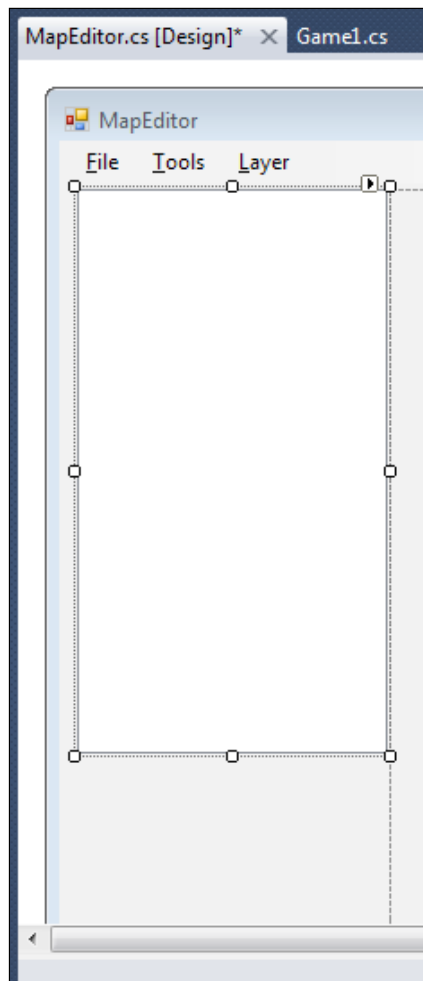
In step two above, each of the menu item entries contains an ampersand (&) character, usually as the first character in the entry. When the `MenuStrip` control sees these characters, instead of displaying them in the menu, it causes the next character in the name to be underlined and treated as a shortcut key. By labelling the **File** menu as **&File**, the item will be displayed as **File**, and pressing **Alt + F** will open the **File** menu. Items within the **File** menu can then be accessed by pressing their own shortcut keys (**L** for **Load**, **S** for **Save**, and so on).

Time for action – tile selection controls

1. Expand the `Textures` folder in the Level Editor Content project.
2. Click on the `PlatformTiles.png` file. The Properties window below Solution Explorer will update to display the properties of the image file.
3. Change the **Copy to Output Directory** property to **Copy if newer**.
4. Switch back to the Design mode view of the `MapEditor` form.
5. Add an `ImageList` control to the `MapEditor` form by double-clicking on the control in the Toolbox window. It will show up in the gray area below the form, as it is a non-visible control. Set the following properties on the `ImageList`:
 - a. **Name** : `imgListTiles`
 - b. **ColorDepth** : `Depth32Bit`
 - c. **ImageSize** : `48, 48`



- 6.** Add a ListView control to the MapEditor form, and give it the following properties:
- a. **Name :** listTiles
 - b. **HideSelection :** False
 - c. **LargeImageList :** imgListTiles
 - d. **Location :** 10, 27
 - e. **MultiSelect:** False
 - f. **Size :** 173, 315
 - g. **TileSize :** 48, 48
 - h. **View :** Tile



7. Right-click on `MapEditor.cs` in Solution Explorer and select **View Code**.

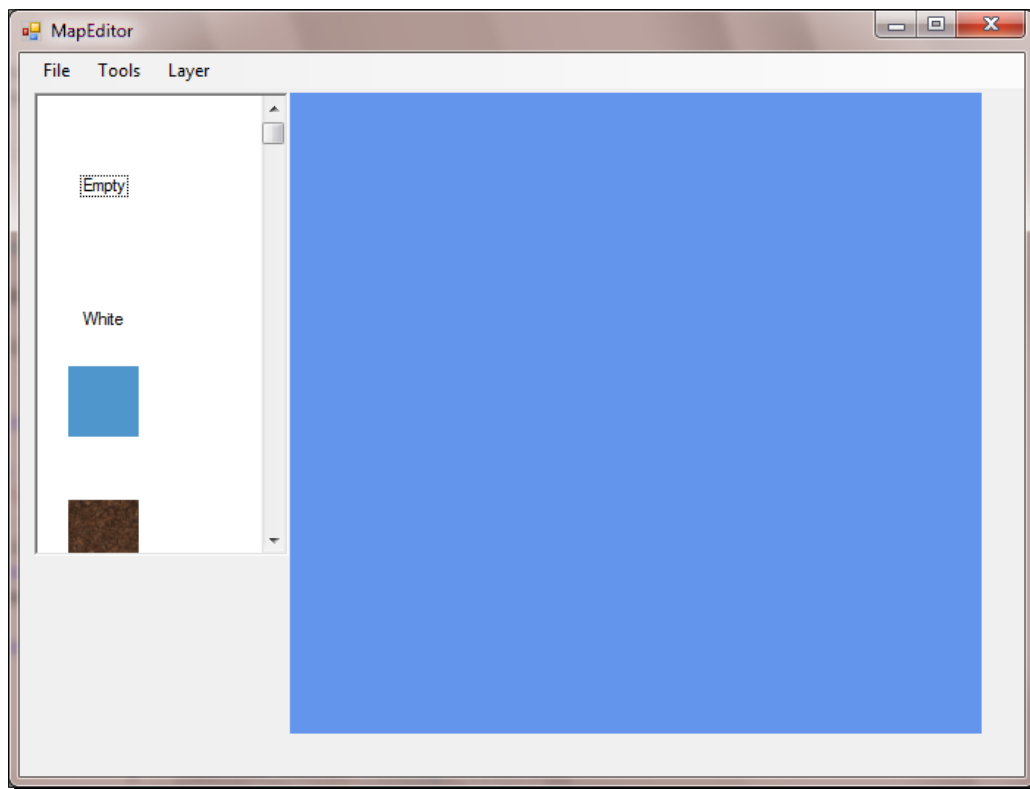
8. Add the following helper method to the `MapEditor` class:

```
private void LoadImageList()
{
    string filepath = Application.StartupPath +
        @"Content\Textures\PlatformTiles.png";
    Bitmap tileSheet = new Bitmap(filepath);
    int tilecount=0;
    for (int y = 0; y < tileSheet.Height / TileMap.TileHeight;
        y++)
    {
        for (int x = 0; x < tileSheet.Width / TileMap.TileWidth;
            x++)
        {
            Bitmap newBitmap = tileSheet.Clone(new
                System.Drawing.Rectangle(
                    x * TileMap.TileWidth,
                    y * TileMap.TileHeight,
                    TileMap.TileWidth,
                    TileMap.TileHeight),
                System.Drawing.Imaging.PixelFormat.DontCare);

            imgListTiles.Images.Add(newBitmap);
            string itemName = "";
            if (tilecount == 0)
            {
                itemName = "Empty";
            }
            if (tilecount == 1)
            {
                itemName = "White";
            }
            listTiles.Items.Add(new
                ListViewItem(itemName, tilecount++));
        }
    }
}
```

- 9.** Double-click on the `MapEditor.cs` file to reopen the Design mode view of the form.
- 10.** Double-click on the title bar for the MapEditor window, causing C# to automatically generate an event handler for the `MapEditor_Load` event.
- 11.** Add the following line to the `MapEditor_Load()` event handler:

```
LoadImageList();
```
- 12.** Execute the application:



- 13.** End the application by selecting **Exit** from the **File** menu.

What just happened?

After executing the project, you should have a scrollable view of the tiles in the tile set in the `ListView` control in the upper-left corner of the editor window. The `LoadImageList()` helper method reads the `PlatformTiles.png` file and splits it up into tile-sized chunks, which it stores inside the `imgListTiles` `ImageList` control.

Notice that we needed to modify the properties of the `PlatformTiles.png` file so that it gets copied to the output directory, because normally the content pipeline would convert the PNG file into an XNB file, which is unreadable by the standard Windows Bitmap class. The XNB file will still be created and copied, but the PNG file will also be placed in the output folder where our `MapEditor` code can find it.

As each image is added to the `imgListTiles` control, entries are also added to the `listTiles` `ListView` control. The first and second tiles are given special labels in the `ListView` (`Empty` and `White`) as they will both appear to be empty white squares since the background of the `ListView` itself is white.

The `MapEditor_Load()` event handler runs when the form is loaded, as its name implies. We will be expanding on this event handler as we add more controls to the form, since it gives us a convenient place to perform initialization.

Time for action – scroll bars

1. In the Toolbox window, double-click on the `VScrollBar` control to add it to the form. Give it the following properties:
 - a. **Name : `vScrollBar1`**
 - b. **Anchor : Top, Bottom, Right**
 - c. **LargeChange : 48**
 - d. **Location : 658, 27**
 - e. **Size : 17, 576**
2. In the Toolbox window, double-click on the `HScrollBar` control to add it to the form. Give it the following properties:
 - a. **Name : `hScrollBar1`**
 - b. **Anchor : Bottom, Left, Right**
 - c. **LargeChange : 48**
 - d. **Location : 184, 606**
 - e. **Size : 474, 17**

- 3.** Add the `FixScrollBarScales()` helper method to the `MapEditor` class:

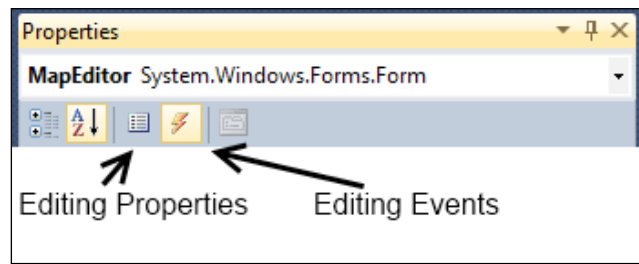
```
private void FixScrollBarScales()
{
    Camera.ViewPortWidth = pctSurface.Width;
    Camera.ViewPortHeight = pctSurface.Height;

    Camera.Move(Vector2.Zero);

    vScrollBar1.Minimum = 0;
    vScrollBar1.Maximum =
        Camera.WorldRectangle.Height -
        Camera.ViewPortHeight;

    hScrollBar1.Minimum = 0;
    hScrollBar1.Maximum =
        Camera.WorldRectangle.Width -
        Camera.ViewPortWidth;
}
```

- 4.** Edit the `MapEditor_Load()` method to include a call to `FixScrollBarScales()`:
- ```
FixScrollBarScales();
```
- 5.** Double-click on `MapEditor.cs` in `SolutionExplorer` to reopen the Design mode view of the `MapEditor` form.
- 6.** Click on the title bar of the `MapEditor` window to select the form as the active control.
- 7.** In the Properties window, ensure that the drop-down box at the top of the window reads **MapEditor System.Windows.Forms.Form**.
- 8.** Still in the Properties widow, click on the yellow lightning bolt button in the toolbar to switch the view from properties to event handlers:



9. Scroll down and locate the **Resize** event and double-click on the empty box to the right of the event name, causing C# to automatically generate an event handler for the `MapEditor_Resize` event.
10. Add the following to the `MapEditor_Resize()` method:  

```
FixScrollBarScales();
```
11. Switch back to properties view in the Properties window by going back to the Design mode view of the form and clicking on the small page icon to the left of the lightning bolt icon in the Properties window toolbar.

### ***What just happened?***

We now have scroll bars attached to the sides of the game's display area. When the form is initially displayed, and then again whenever it is resized, the scroll bars will be rescaled so that they cover the entire area of the game's tile map.

We will use these scroll bars to move around on the map while editing, though their actual implementation will again be tied to changes to the `Game1` class.

### **Time for action – final controls**

1. Add a `GroupBox` to the `MapEditor` form, and give it the following properties:
  - a. **Name :** `groupBoxRightClick`
  - b. **Location :** 10, 346
  - c. **Size :** 173, 103
  - d. **Text :** Right Click Mode
2. Add a `RadioButton` control inside the `groupBoxRightClick` `GroupBox` by dragging it from the Toolbox window and dropping it inside the Groupbox. Give it the following properties:
  - a. **Name :** `radioPassable`
  - b. **Checked :** True
  - c. **Location :** 6, 17
  - d. **Text :** Toggle Passable

3. Add another RadioButton control inside the **groupBoxRightClick** control, with the following properties:
  - a. **Name : radioCode**
  - b. **Location : 6, 35**
  - c. **Text : Code**
4. Add a TextBox control inside the **groupBoxRightClick** control with these properties:
  - a. **Name : txtNewCode**
  - b. **Location : 62, 36**
  - c. **Size : 103, 20**
5. Add a Label control inside the **groupBoxRightClick** control with these properties:
  - a. **Name : lblCurrentCode**
  - b. **Location : 60, 59**
  - c. **Text : ---**
6. Add a ComboBox control inside the **groupBoxRightClick** control with these properties:
  - a. **Name : cboCodeValues**
  - b. **DropDownStyle : DropDownList**
  - c. **Location : 5, 75**
  - d. **Size : 160, 21**
7. Add a Label control below the group box with these properties:
  - a. **Name : lblMapNumber**
  - b. **Location : 12, 452**
  - c. **Text : Map Number:**
8. Add a ComboBox control below the group box with these properties:
  - a. **Name : cboMapNumber**
  - b. **DropDownStyle : DropDownList**
  - c. **Location : 81, 452**
  - d. **Size : 94, 21**

- 9.** Modify the `MapEditor_Load()` method of the `MapEditor` class by adding the following to the existing code:

```
cboCodeValues.Items.Clear();
cboCodeValues.Items.Add("Gemstone");
cboCodeValues.Items.Add("Enemy");
cboCodeValues.Items.Add("Lethal");
cboCodeValues.Items.Add("EnemyBlocking");
cboCodeValues.Items.Add("Start");
cboCodeValues.Items.Add("Clear");
cboCodeValues.Items.Add("Custom");

for (int x = 0; x < 100; x++)
{
 cboMapNumber.Items.Add(x.ToString().PadLeft(3, '0'));
}

cboMapNumber.SelectedIndex = 0;
TileMap.EditorMode = true;
```

- 10.** Return to the Design view of the `MapEditor` form and double-click on the `cboCodeValues` combo box (the combo box inside the group box). Update the automatically generated event handler to read:

```
private void cboCodeValues_SelectedIndexChanged(object sender,
EventArgs e)
{
 txtNewCode.Enabled = false;
 switch(
 cboCodeValues.Items[cboCodeValues.SelectedIndex].ToString())
 {
 case "Gemstone" :
 txtNewCode.Text = "GEM";
 break;

 case "Enemy" :
 txtNewCode.Text = "ENEMY";
 break;

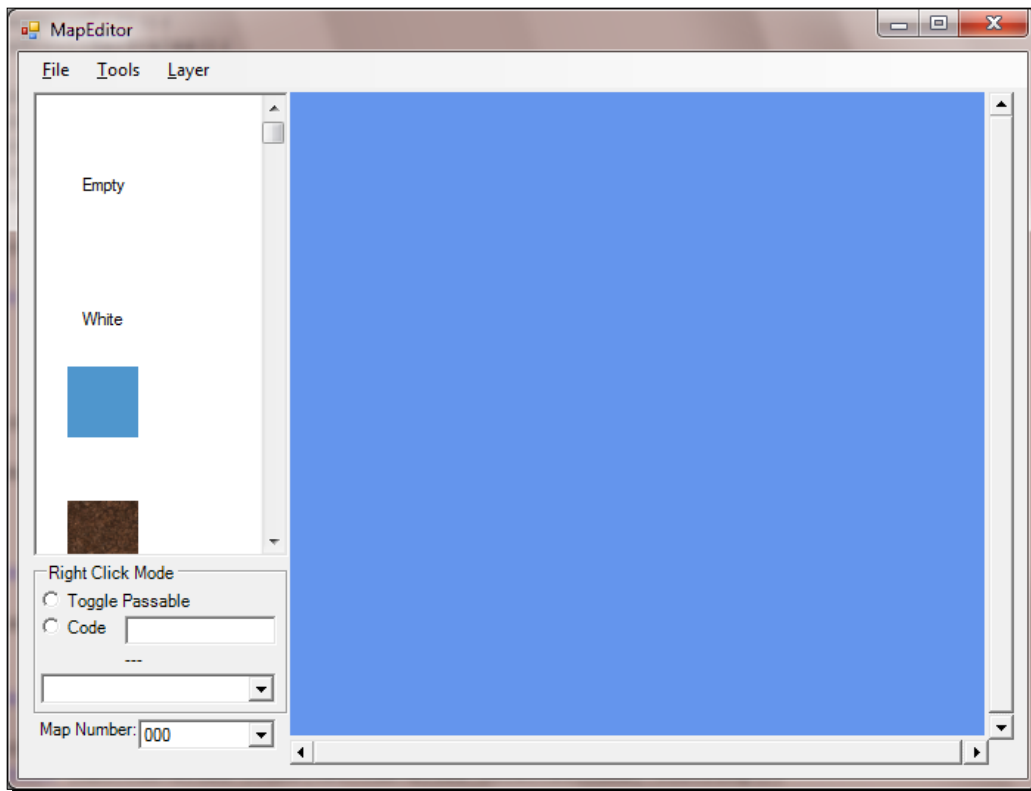
 case "Lethal" :
 txtNewCode.Text = "DEAD";
 break;

 case "EnemyBlocking":
 txtNewCode.Text = "BLOCK";
 break;
```



```
 case "Start":
 txtNewCode.Text = "START";
 break;
 case "Clear":
 txtNewCode.Text = "";
 break;
 case "Custom":
 txtNewCode.Text = "";
 txtNewCode.Enabled = true;
 break;
 }
}
```

**11.** Execute the application:



## ***What just happened?***

We now have all of the interactive controls we need for our level editor. The codes drop-down box provides a number of standard code values that we will use during level creation, with the ability to add custom codes, which are entered in the textbox above it.

## **Updating the Game1 class**

We currently have a Windows form that contains our XNA Game display, but the two pieces of the level editor application do not yet share any information or allow the user to actually edit maps. It is time to begin updating our game to support level editing.

### **Time for action – updating Game1**

1. Double-click on the `Game1.cs` file in the Level Editor project to open it in the editor.

2. Add the following declarations to the Game1 declarations area:

```
public int DrawLayer = 0;
public int DrawTile = 0;
public bool EditingCode = false;
public string CurrentCodeValue = "";
public string HoverCodeValue = "";

public MouseState lastMouseState;
System.Windows.Forms.VScrollBar vscroll;
System.Windows.Forms.HScrollBar hscroll;
```

3. Add the following lines to the Game1 constructor:

```
vscroll =
 (System.Windows.Forms.VScrollBar)parentForm.Controls[
 "vScrollBar1"];
hscroll =
 (System.Windows.Forms.HScrollBar)parentForm.Controls[
 "hScrollBar1"];
```

4. Modify the `LoadContent()` method of the Game1 class to read:

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

 Camera.ViewPortWidth = pictureBox.Width;
 Camera.ViewPortHeight = pictureBox.Height;
 Camera.WorldRectangle =
```

```
 new Rectangle(
 0,
 0,
 TileMap.TileWidth * TileMap.MapWidth,
 TileMap.TileHeight * TileMap.MapHeight
);

 TileMap.Initialize(
 Content.Load<Texture2D>(@"Textures\PlatformTiles"));
 TileMap.spriteFont =
 Content.Load<SpriteFont>(@"Fonts\Pericles8");
 lastMouseState = Mouse.GetState();
 pictureBox_SizeChanged(null, null);
}
```

**5. Modify the Draw() method of the Game1 class to read:**

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

 spriteBatch.Begin(
 SpriteSortMode.BackToFront,
 BlendState.AlphaBlend);
 TileMap.Draw(spriteBatch);
 spriteBatch.End();

 base.Draw(gameTime);
}
```

## ***What just happened?***

To simplify communications between the Windows Form and the XNA Game, we have declared a number of public member variables that our Windows Form code will be able to update in response to user-generated events. We have also loaded a `SpriteFont` to draw code values with, and a `MouseState` variable to hold the state of the mouse between frames.

Finally, we declare two objects that reference the scroll bars on the level editor form. We will use these to sync up the display of the tile map to the location of the scroll bars.

The `LoadContent()` method is fairly standard, setting the size of the tile map, and loading the tile images and sprite font. The `TileMap` class's `spriteFont` member is set to the font we loaded, and the `lastMouseState` member is initialized. Right before exiting, the `LoadContent()` method calls `pictureBox_SizeChanged()` to make sure that the graphics device has the proper dimensions for the display window.

In our `Draw()` method, we have again used the expanded form of the `SpriteBatch.Begin()` call in order to specify the `SpriteSortMode.BackToFront` parameter.

## Time for action – the Game1 Update method

1. Replace the current `Update()` method in the `Game1` class with the following:

```
protected override void Update(GameTime gameTime)
{
 Camera.Position = new Vector2(hscroll.Value, vscroll.Value);
 MouseState ms = Mouse.GetState();
 if ((ms.X > 0) && (ms.Y > 0) &&
 (ms.X < Camera.ViewPortWidth) &&
 (ms.Y < Camera.ViewPortHeight))
 {
 Vector2 mouseLoc = Camera.ScreenToWorld(
 new Vector2(ms.X, ms.Y));
 if (Camera.WorldRectangle.Contains(
 (int)mouseLoc.X, (int)mouseLoc.Y))
 {
 if (ms.LeftButton == ButtonState.Pressed)
 {
 TileMap.SetTileAtCell(
 TileMap.GetCellByPixelX((int)mouseLoc.X),
 TileMap.GetCellByPixelY((int)mouseLoc.Y),
 DrawLayer,
 DrawTile);
 }
 if ((ms.RightButton == ButtonState.Pressed) &&
 (lastMouseState.RightButton ==
 ButtonState.Released))
 {
 if (EditingCode)
 {

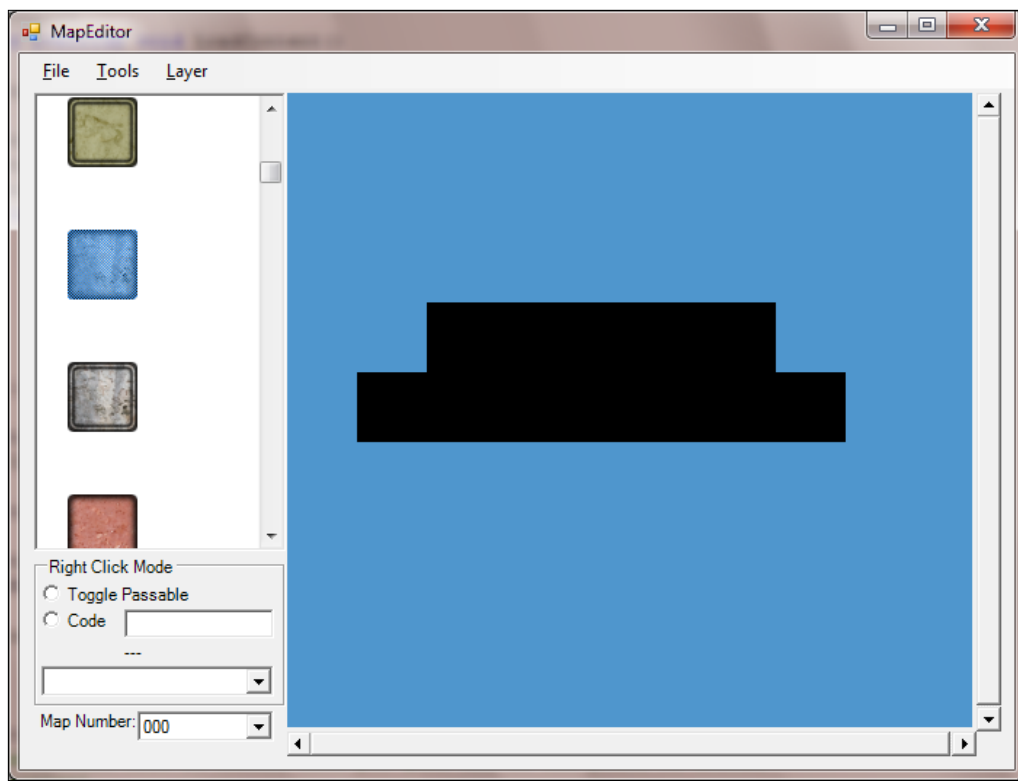
```

```
 TileMap.GetMapSquareAtCell(
 TileMap.GetCellByPixelX((int)mouseLoc.X),
 TileMap.GetCellByPixelY((int)mouseLoc.Y)
).CodeValue = CurrentCodeValue;
 }
 else
 {
 TileMap.GetMapSquareAtCell(
 TileMap.GetCellByPixelX((int)mouseLoc.X),
 TileMap.GetCellByPixelY((int)mouseLoc.Y)
).TogglePassable();
 }
}

HoverCodeValue =
 TileMap.GetMapSquareAtCell(
 TileMap.GetCellByPixelX(
 (int)mouseLoc.X),
 TileMap.GetCellByPixelY(
 (int)mouseLoc.Y)).CodeValue;
}

lastMouseState = ms;
base.Update(gameTime);
}
```

- 2.** Execute the application. Attempting to draw tiles at this point results in black holes being punched in the all-blue map:



### ***What just happened?***

The first thing our `Update()` method does is to set the game's camera position based on the current values of the horizontal and vertical scroll bars on the level editor form. When we scroll the scroll bars, the game map will scroll as well.

Next, we verify that the mouse coordinates are within the view port of the camera, and thus within the `PictureBox` control on the editor form. If they are, we determine the world-based coordinates of the mouse cursor.

If the left mouse button has been pressed, we update the tile under the cursor on the current `DrawLayer` with the current `DrawTile`. The right mouse button is a bit more complicated.

We will use the right mouse button to set two different types of information—either toggling on and off the `Passable` property of the `MapSquare`, or setting its `CodeValue` property, depending on the mode selected on the `MapEditor` form.

In either case, we will only make a change to the underlying tile if the right mouse button is pressed during this frame and was not pressed during the previous frame. This eliminates updating the same square multiple times on a single button press (which would make toggling passability on and off difficult!) During the first frame after the button is pressed, the `MapSquare` will be updated. Until the button is released, no other updates will occur.

The `HoverCodeValue` member is set to the `CodeValue` of the `MapSquare` under the mouse cursor. This value will be used by the `MapEditor` form and displayed on a label on the screen to provide an alternative method of viewing the code for an individual square.

Finally, `lastMouseState` is updated to the current mouse state and the `Update()` method is completed.

## Connecting the form to the game

As we saw at the end of the last section, attempting to draw tiles to the map at this point only leaves empty spaces. This is because the `DrawLayer` and `DrawTile` integers both default to zero, so we are drawing a fully-transparent tile onto the background layer.

Now that we have made the necessary updates to the XNA side of the level editor, we need to flesh out the event handlers for the controls we placed on the `MapEditor` form to update the control variables inside the `Game1` class.

### Time for action – completing the editor – part 1

1. Open the `MapEditor` form in Design mode.
2. Double-click on the `listTiles` `ListView` control to automatically generate an event handler for the `SelectedIndexChanged` event. Update the event code to read:

```
private void listTiles_SelectedIndexChanged(
 object sender, EventArgs e)
{
 if (listTiles.SelectedIndices.Count > 0)
 {
 game.DrawTile =
 listTiles.SelectedIndices[0];
 }
}
```

3. Return to the Design view of the MapEditor form and double-click on the radio button labelled **Toggle Passable** to generate a handler for the `CheckChanged` event. Update the event handler to read:

```
private void radioPassable_CheckedChanged(object sender, EventArgs e)
{
 if (radioPassable.Checked)
 {
 game.EditingCode = false;
 }
 else
 {
 game.EditingCode = true;
 }
}
```

4. Return to the Design view and double-click on the **Code** radio button. Update the `CheckChanged` event handler to read:

```
private void radioCode_CheckedChanged(object sender, EventArgs e)
{
 if (radioPassable.Checked)
 {
 game.EditingCode = false;
 }
 else
 {
 game.EditingCode = true;
 }
}
```

5. Return to the Design view and double-click on the `txtNewCode` textbox. Update the `TextChanged` event handler to read:

```
private void txtNewCode_TextChanged(object sender, EventArgs e)
{
 game.CurrentCodeValue = txtNewCode.Text;
}
```



- 6.** Return to the Design view, and double-click on the **Background** menu item under the **Layer** menu. Update the Click event handler to read:

```
private void backgroundToolStripMenuItem_Click(
 object sender, EventArgs e)
{
 game.DrawLayer = 0;
 backgroundToolStripMenuItem.Checked = true;
 interactiveToolStripMenuItem.Checked = false;
 foregroundToolStripMenuItem.Checked = false;
}
```

- 7.** Generate a Click event handler for the **Interactive layer** menu item and update it to read:

```
private void interactiveToolStripMenuItem_Click(
 object sender, EventArgs e)
{
 game.DrawLayer = 1;
 backgroundToolStripMenuItem.Checked = false;
 interactiveToolStripMenuItem.Checked = true;
 foregroundToolStripMenuItem.Checked = false;
}
```

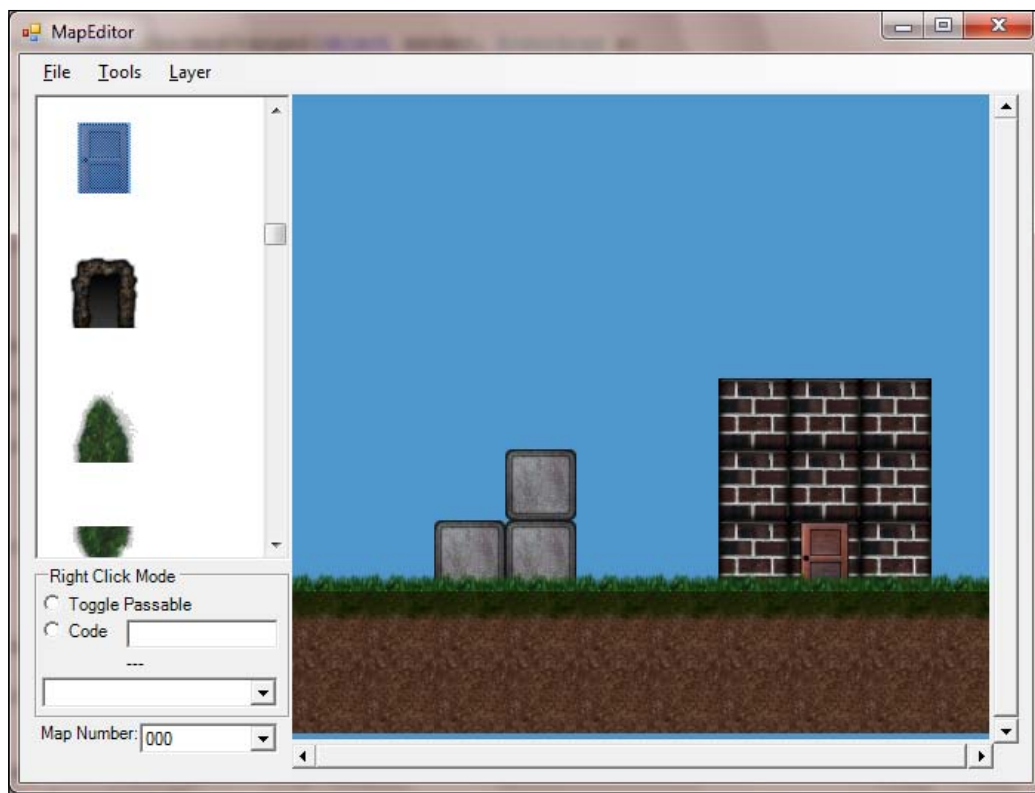
- 8.** Generate a Click event handler for the **Foreground layer** menu item and update it to read:

```
private void foregroundToolStripMenuItem_Click(
 object sender, EventArgs e)
{
 game.DrawLayer = 2;
 backgroundToolStripMenuItem.Checked = false;
 interactiveToolStripMenuItem.Checked = false;
 foregroundToolStripMenuItem.Checked = true;
}
```

- 9.** In the `MapEditor_Load()` method, add the following to the end of the method to indicate the starting layer for the editor:

```
backgroundToolStripMenuItem.Checked = true;
```

- 10.** Execute the application and use the editor to draw tiles to the map:



11. Maximize the display window and attempt to use the scroll bars to scroll around the map display.

### ***What just happened?***

All of our event handlers for the form controls simply pass information along to the appropriate variables in the `Game1` class. Selecting a layer from the menu bar updates the `DrawLayer` value, while the radio buttons toggle the `EditingCode` variable between `true` and `false`. Any time the `txtNewCode` control's `Text` property is changed, the current contents of the `TextBox` are copied to the `CurrentCodeValue` member of the `Game1` class.

In step 11 above, you probably noticed that the game window does not update the current position when you are scrolling via the scroll bars until you release the mouse button, even though the scroll bar's marker moves the whole time. This happens because the movement of the scroll bar is preventing the game loop from executing while the user is in the process of manipulating the scroll bars.

In addition, the scroll bars do not match up to the map window until we resize the display at least once. This is because the form's Load event happens prior to the initialization of the Game1 instance (if you go back and look at Program.cs, the form is created and shown first, and then the game is initialized) so the WorldRectangle property of the Camera class is {0, 0, 0, 0}. This results in FixScrollBarScales() setting negative numbers for the maximum values of the scroll bars.

We can fix both of these problems by adding a Timer to the form that initially calls FixScrollBarScales() and repeatedly calls the game's Tick() method.

## Time for action – fixing the scrolling delay

1. Reopen the Design mode view of the MapEditor window.
2. Double-click on the Timer control in the Toolbox window to add a new instance to the MapEditor form. As with the ImageList control, the Timer is not visible, and will appear in the editor as an icon and label below the design window. Give the timer control the following properties:

- ❑ **Name : timerGameUpdate**
- ❑ **Enabled : True**
- ❑ **Interval : 20**

3. Double-click on the timerGameUpdate control to generate a Tick event handler and add the following code to it:

```
private void timerGameUpdate_Tick(object sender, EventArgs e)
{
 if (hScrollBar1.Maximum < 0)
 {
 FixScrollBarScales();
 }

 game.Tick();

 if (game.HoverCodeValue != lblCurrentCode.Text)
 lblCurrentCode.Text = game.HoverCodeValue;
}
```

4. Execute the application. Draw a few tiles on the map and use the scroll bars to verify that they function as expected.

## What just happened?

Using the scroll bars does not prevent the Timer control from firing its `Tick` event, so by executing the game's `Tick()` method from within the `timerGameUpdate_Tick()` event handler, we can force the game's `Update()` and `Draw()` methods to run even when they normally would not.

The last item in the `timerGameUpdate_Tick()` handler checks to see if the `HoverCodeValue` inside the `Game1` class has been updated since it was last copied to the label displaying it on the Windows Form. If it has, the form label is updated as well.

## Loading and Saving Maps

The last thing we need to address to complete the Gemstone Hunter Level Editor is how we will load and save our map files. There are a number of ways we could store our level maps, but we will implement a very simple method that does not require parsing XML or creating a text file with a special format to store the map.

### Time for action – implementing loading and saving

1. In the Tile Engine project open the `TileMap.cs` class file.
2. Add the Loading and Saving Maps region to the `TileMap` class:

```
#region Loading and Saving Maps
public static void SaveMap(FileStream fileStream)
{
 BinaryFormatter formatter = new BinaryFormatter();
 formatter.Serialize(fileStream, mapCells);
 fileStream.Close();
}

public static void LoadMap(FileStream fileStream)
{
 try
 {
 BinaryFormatter formatter = new BinaryFormatter();
 mapCells =
 (MapSquare[,])formatter.Deserialize(fileStream);
 fileStream.Close();
 }
 catch
 {
 }
}
```

```
 ClearMap();
 }
}

public static void ClearMap()
{
 for (int x = 0; x < MapWidth; x++)
 for (int y = 0; y < MapHeight; y++)
 for (int z = 0; z < MapLayers; z++)
 {
 mapCells[x, y] = new MapSquare(2, 0, 0, "", true);
 }
}
#endregion
```

3. Back in the Level Editor project, open the MapEditor form in Design mode.
4. Double-click on the **Load Map** item in the **File** menu to create the Click event handler and update its code to read:

```
private void loadMapToolStripMenuItem_Click(
 object sender,
 EventArgs e)
{
 try
 {
 TileMap.LoadMap(new FileStream(
 Application.StartupPath + @"\MAP" +
 cboMapNumber.Items[cboMapNumber.SelectedIndex] +
 ".MAP",
 FileMode.Open));
 }
 catch
 {
 System.Diagnostics.Debug.Print("Unable to load map file");
 }
}
```

5. Double-click on the **Save Map** item in the **File** menu and update its Click handler to read:

```
private void saveMapToolStripMenuItem_Click(
 object sender,
 EventArgs e)
```

```

 {
 TileMap.SaveMap(new FileStream(
 Application.StartupPath + @"\MAP" +
 cboMapNumber.Items[cboMapNumber.SelectedIndex] + ".MAP",
 FileMode.Create));
 }

```

6. Double-click on the **Clear Map** item in the **Tools** menu and update its `Click` handler to read:

```

private void clearMapToolStripMenuItem_Click(
 object sender,
 EventArgs e)
{
 TileMap.ClearMap();
}

```

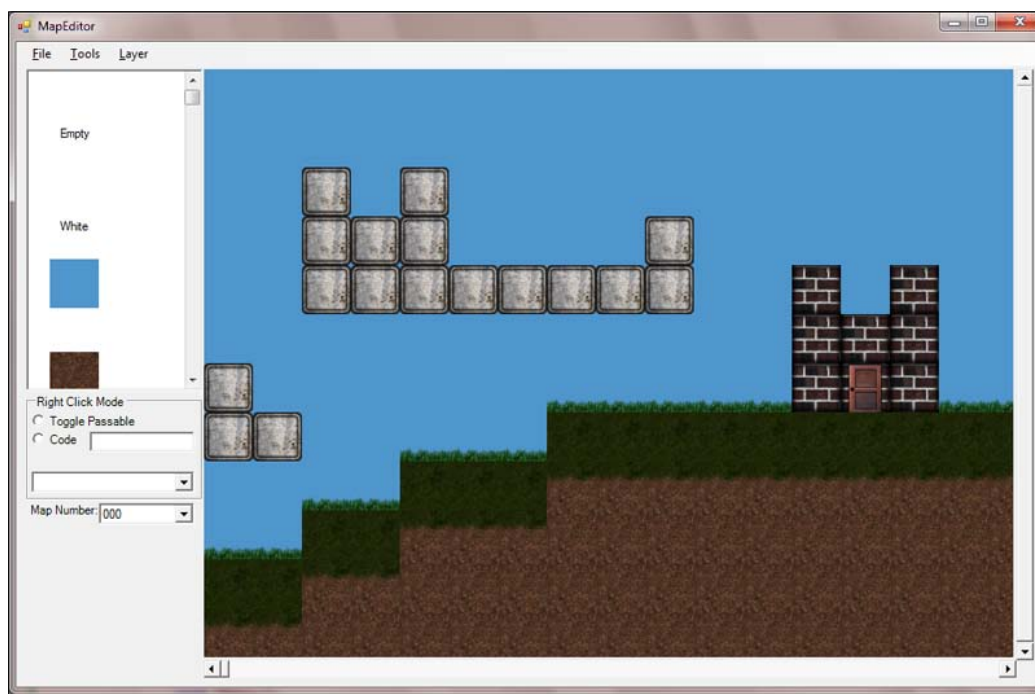
7. Execute the application and create a simple map. Save it to disk, update the map, and reload it.

### ***What just happened?***

When a map is saved, we create a `FileStream` object, which represents our file on disk. We then create an instance of the `BinaryFormatter` class and call its `Serialize()` method, passing in the stream to serialize the data from the object we wish to serialize. In our case, it is the array containing the `MapSquare` objects that represent our game map.

When loading the map, the process is exactly the same, except that we use the `Deserialize()` method of the `BinaryFormatter` class to reverse the process, converting the binary data on disk back into its in-memory representation. By surrounding the attempt to load the map with a `try...catch` block, we can take action (clearing the map to an empty blue sky) instead of simply crashing the level editor.

From the Windows Form side of the system, we call the new `LoadMap()` and `SaveMap()` methods, passing in the `FileStream` object that is created based on the `cboMapNumber` drop-down list. Our maps will be saved in files named "MAP###.MAP", with the three digit number taken from the `cboMapNumber` list. While using our maps in the Level Editor, they will be stored in the same directory our Level Editor executable is running from (normally, the `Visual Studio 2010\Projects\Gemstone Hunter\Level Editor\Level Editor\bin\x86\debug` folder inside your `Documents` folder). In the following screenshot, we have created a sample map using the level editor. This map currently has no passability or code information on it, and it contains only tile information for now:



## Passability

When building maps, we will use the right mouse button to toggle each individual map square as either passable or impassable. When a square is marked as impassable, it will be tinted red by the editor, indicating that both monsters and the player will treat the square as a solid wall, no matter what visual representation the square has.

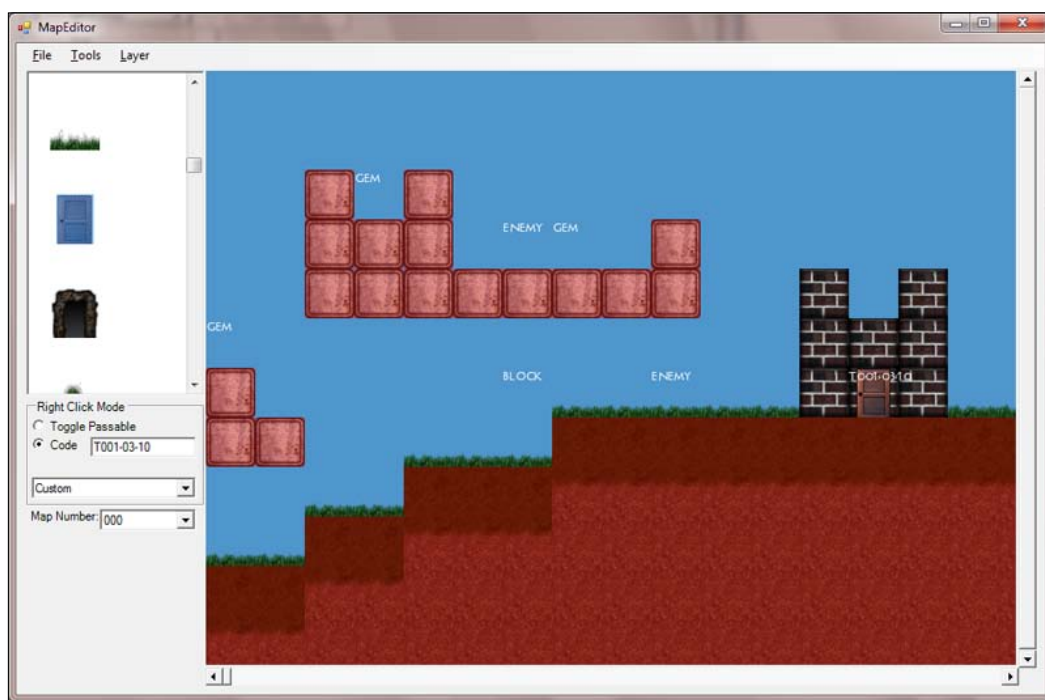
Without this information, the player would fall straight through the level and off the map—something we will need to account for in the game engine when we build it in the next chapter.

## Map codes

Each MapSquare can be assigned a code value that will allow the game to implement special behavior for that square. We have pre-defined a handful of code values including:

- ◆ Gemstone (GEM)—a gem will be spawned at this location for the player to collect.
- ◆ Enemy (ENEMY)—an enemy will be spawned at this location.
- ◆ Lethal (DEAD)—contacting this square will kill the player.
- ◆ Enemy Blocking (BLOCK)—players can move through these squares, but enemies will treat them as walls. This allows us to confine enemies to an elevated platform, for example.
- ◆ Start (START)—if no position is set because of a map transition, the player will start the map in this square.

In addition, we will define a special code for map transitions. In the following image, we have a code value of **T-001-03-10** on the MapSquare containing the door into the brick building. At runtime, we will interpret this code value to mean "Transition (T) to map 001, at location 03, 10". In this way, we can link maps together and allow the player to move between them:





The previous image shows the map from the previous section with both passability and code information filled in. In Editor mode, the `TileEngine` class displays the code values as text blocks on each map square. These codes mean nothing to the editor, so just like passability information we will need to account for it in the game engine.

## One last issue

Remember way back when we hid the empty `Game1` form, the one where we added code to the **Exit** menu item to properly terminate the application? We can clear up what happens when the user clicks on the X button in the upper-right corner of the window to close the application.

### Time for action – handling the `FormClosed` event

1. Open the `MapEditor` form in Design mode.
2. Select the form as the current object by clicking on the form's title bar in the Design window.
3. Switch to Event editing mode in the Properties window by clicking on the lightning-bolt button.
4. Scroll down to the `FormClosed` event and double-click in the empty box to the right of the event name to create the `MapEditor_FormClosed()` event handler.
5. Update the event handler to read:

```
private void MapEditor_FormClosed(
 object sender,
 FormClosedEventArgs e)
{
 game.Exit();
 Application.Exit();
}
```

### ***What just happened?***

When the form closes, we need to shut down both the XNA game and the overall application, otherwise the system will not release the resources, and the program will still be running invisibly in the background.

## Have a go hero

The Gemstone Hunter Level Editor project is fairly rough around the edges. It is not exactly a model example of Windows Forms development, but then few purpose-built internal game development tools are.

If you feel like diving further into Windows Forms development, here are a few suggestions for improving on the level editor:

- ◆ Currently, the level editor does not alert you if you try to load a map after you have made changes to the current map. By adding checks to the `Update()` method of the `Game1` class, you could flag the map as having changed and issue the appropriate warnings to the user when they try to load a new map.
- ◆ Marking squares as impassable requires an individual click on each square. You could expand the number of radio buttons to include marking squares as passable and impassable as separate tasks, thus allowing the user to hold down the mouse button and draw large blocks of impassable squares.
- ◆ On the more game-focused side of things, try creating a few levels! The level editor supports up to 100 levels (000 through 099), so there is plenty of room to experiment.

## Summary

We now have a working—if not pretty—level editor. In Chapter 8, we:

- ◆ Added multiple layers and other types of data to the `TileMap` class we built for Robot Rampage
- ◆ Created a multi-project Visual Studio solution that shares code between projects
- ◆ Added a Windows Forms form to our XNA Game Studio project and modified the program's start-up process to render the form to a `PictureBox` control on the form
- ◆ Implemented methods to allow communication between the Windows Form and the XNA game, including synchronized scroll bars, and updating member variables in the `Game1` class in response to Windows Forms controls events
- ◆ Implemented methods to load and save map files via the `BinaryFormatter` class

In Chapter 9, we will flesh out the Gemstone Hunter project and cover the basics of building a platform-style game using the maps we create with the level editor from this chapter.



# 9

## Gemstone Hunter – Standing on Your Own Two Pixels

*With our level editor completed, we can now move on to the building of the Gemstone Hunter game itself. As with the other games in this book, we will build on what we have learned in previous projects and introduce some new concepts.*

In this chapter, we will cover the following topics:

- ◆ A new approach to animating game objects using named animation strips
- ◆ A more object-oriented approach to game objects
- ◆ Platform game physics, allowing the player to run, jump, and squash enemies
- ◆ Processing map codes when a map is loaded to spawn objects in the game world
- ◆ Using map codes at runtime to generate in-game effects

### Animation strips

In all of the other game projects in this book, our graphical resources have been confined to a single sprite sheet, onto which we have consolidated all of the images needed for our gameplay elements.

This works well for many small games, but it is certainly not the only way to organize your content. Since we are borrowing content from the XNA Platform Starter Kit, we will use it in the format it has been provided to us instead of creating new sprite sheets.

For each type of entity we will display in Gemstone Hunter, we have one or more PNG files containing multiple image frames for a single animation. For example, the `Run.png` file for the main character from the Platform Starter Kit looks like this:



Each frame is the same size (48 by 48 pixels in this case) and the size of the image file itself determines the number of frames contained in the animation. The run animation is 480 pixels wide, at 48 pixels per frame, so there are 10 frames in the animation.

## **Time for action – building the Animation class**

1. In the Gemstone Hunter project, add a new class file called `AnimationStrip.cs`.
2. Add the following using directives to the `AnimationStrip` class:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

3. Modify the declaration of the `AnimationStrip` class to make it a public class:

```
public class AnimationStrip
```

4. Add declarations to the `AnimationStrip` class:

```
#region Declarations
private Texture2D texture;
private int frameWidth;
private int frameHeight;

private float frameTimer = 0f;
private float frameDelay = 0.05f;

private int currentFrame;

private bool loopAnimation = true;
private bool finishedPlaying = false;

private string name;
private string nextAnimation;
#endregion
```

**5. Add properties to the AnimationStrip class:**

```
#region Properties
public int FrameWidth
{
 get { return frameWidth; }
 set { frameWidth = value; }
}

public int FrameHeight
{
 get { return frameHeight; }
 set { frameHeight = value; }
}

public Texture2D Texture
{
 get { return texture; }
 set { texture = value; }
}

public string Name
{
 get { return name; }
 set { name = value; }
}

public string NextAnimation
{
 get { return nextAnimation; }
 set { nextAnimation = value; }
}

public bool LoopAnimation
{
 get { return loopAnimation; }
 set { loopAnimation = value; }
}

public bool FinishedPlaying
{
 get { return finishedPlaying; }
}

public int FrameCount
{
```

```
 get { return texture.Width / frameWidth; }
 }

 public float FrameLength
 {
 get { return frameDelay; }
 set { frameDelay = value; }
 }

 public Rectangle FrameRectangle
 {
 get
 {
 return new Rectangle(
 currentFrame * frameWidth,
 0,
 frameWidth,
 frameHeight);
 }
 }
}
#endregion
```

**6.** Add a constructor to the AnimationStrip class:

```
#region Constructor
public AnimationStrip(Texture2D texture, int frameWidth, string
 name)
{
 this.texture = texture;
 this.frameWidth = frameWidth;
 this.frameHeight = texture.Height;
 this.name = name;
}
#endregion
```

**7.** Add public methods to the AnimationStrip class:

```
#region Public Methods
public void Play()
{
 currentFrame = 0;
 finishedPlaying = false;
}
}
```

```
public void Update(GameTime gameTime)
{
 float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
 frameTimer += elapsed;
 if (frameTimer >= frameDelay)
 {
 currentFrame++;
 if (currentFrame >= FrameCount)
 {
 if (loopAnimation)
 {
 currentFrame = 0;
 }
 else
 {
 currentFrame = FrameCount - 1;
 finishedPlaying = true;
 }
 }
 frameTimer = 0f;
 }
}
#endregion
```

### ***What just happened?***

Each `AnimationStrip` is defined primarily by the `Texture2D` object that contains the images for the animation and the frame size information passed into the constructor. From these two pieces of information, we can play a basic animation by including the timing loop logic that we have used many times previously.

The `Play` method sets the `currentFrame` variable to zero and `finishedPlaying` to `false`, resulting in future calls to the `Update()` method advancing the frame until the last frame in the image is reached.

What happens after the last frame is displayed is dependent on the value of `loopAnimation`. If this Boolean variable is `true`, the `currentFrame` counter will be set back to zero, and the animation will continue to play in a continuous loop.



If `loopAnimation` is `false`, the animation will continue to display the last frame of the animation, setting `finishedPlaying` to `true`. We can check this value from our game code to determine what we should do after an animation has finished playing.

In order to help decide what to play next, each `AnimationStrip` has `name` and `nextAnimation` string variables. For example, the graphics for the player's character in the Platform Starter Kit contains an animation for jumping. When we load this animation, we can specify that the jump animation's `nextAnimation` property is "idle".

When we build our game objects, we can watch for an animation to finish playing and then start playing the animation specified by `nextAnimation` to present a smooth transition between animated states.

It is important to note that the `AnimationStrip` class does not do any drawing. It simply provides a texture source and frame rectangle to another object that will perform the actual drawing to the display. The game objects themselves will keep track of their positions and current animation states, requesting the appropriate information from their `AnimationStrips` when they need to be drawn.

## Animated game objects

The basis for all of our game objects apart from the tile-based map (the player, enemies, and gemstones) will be a class called `GameObject`. This class will provide support for playing animations and collision detection with the tile map.

In many ways, the `GameObject` class is similar to the `Sprite` class we built for our other games. Because the `GameObject` class does not hold texture information like the `Sprite` class did, we have given it a new name to better describe its functionality.

Just as we did in *Robot Rampage*, we will track the position of all of our game objects in world coordinates, translating those to screen coordinates as necessary with the `Camera` class.

### Time for action – building the `GameObject` class – Part 1

1. Add a new class called `GameObject.cs` to the Gemstone Hunter project.
2. Add the following `using` directives to the `GameObject` class:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Tile_Engine;
```

**3.** Modify the declaration of the `GameObject` class to make the class public:

```
public class GameObject
```

**4.** Add declarations to the `GameObject` class:

```
#region Declarations
protected Vector2 worldLocation;
protected Vector2 velocity;
protected int frameWidth;
protected int frameHeight;

protected bool enabled;
protected bool flipped = false;
protected bool onGround;

protected Rectangle collisionRectangle;
protected int collideWidth;
protected int collideHeight;
protected bool codeBasedBlocks = true;

protected float drawDepth = 0.85f;
protected Dictionary<string, AnimationStrip> animations =
 new Dictionary<string, AnimationStrip>();
protected string currentAnimation;
#endregion
```

**5.** Add properties to the `GameObject` class:

```
#region Properties
public bool Enabled
{
 get { return enabled; }
 set { enabled = value; }
}

public Vector2 WorldLocation
{
 get { return worldLocation; }
 set { worldLocation = value; }
}

public Vector2 WorldCenter
{
 get
 {
 return new Vector2(
 (int)worldLocation.X + (int)(frameWidth/2),
 (int)worldLocation.Y + (int)(frameHeight/2));
 }
}
```

```
 }
 public Rectangle WorldRectangle
 {
 get
 {
 return new Rectangle(
 (int)worldLocation.X,
 (int)worldLocation.Y,
 frameWidth,
 frameHeight);
 }
 }
 public Rectangle CollisionRectangle
 {
 get
 {
 return new Rectangle(
 (int)worldLocation.X + collisionRectangle.X,
 (int)WorldRectangle.Y + collisionRectangle.Y,
 collisionRectangle.Width,
 collisionRectangle.Height);
 }
 set { collisionRectangle = value; }
 }
#endregion
```

## ***What just happened?***

As we can see, most of the properties of the `GameObject` class are carried over from the `Sprite` class from our previous games. Properties like `WorldLocation`, `WorldCenter`, and `CollisionRectangle` all have the same meaning for `GameObjects` as they did for `Sprites`.

We do, however, have some new additions. All of the animation strips for our characters represent the character facing to the left. When the character should be facing right, we will use a parameter of the `SpriteBatch.Draw()` method to flip the image horizontally. The `flipped` member variable will keep track of the character's current facing.

Our player will have the ability to jump, but logically should only be able to jump when standing on the ground. Jumping while already in the air would certainly produce strange results. The `onGround` Boolean will be set to `true` whenever the player has come to rest on a solid floor block, indicating to the jump code that a jump is allowed.

Recall that in our level editor we created a `CodeValue` entry called `Enemy Blocking`, which indicates that a particular square is impassable to enemies, while not restricting player movement the way the `Passable` flag does. If the `codeBaseBlocks` variable is set to `true`, our collision detection code will take these blocks into account when detecting impassable walls and floors. For all game objects except the player, this value will be `true`, so that is the default set in the declaration.

In order to draw the `GameObjects` to the screen and have them displayed at the appropriate depth compared to the tile map layers, we need to specify a float that will be passed as the `layerDepth` parameter of the `SpriteBatch.Draw()` method. In this case, the `drawDepth` parameter is set to `0.85f`. When our tile layers are drawn, they are drawn at depths of `1.0f` and `0.9f` for the background and interactive layers, and `0.7f` for the foreground layer.

This means that game objects will be drawn behind the foreground layer but above the background and interactive layers, allowing for objects such as doors to be placed on the interactive layer over a background and still have the player walk in front of them.

Unlike a `Sprite`, the `GameObject` does not contain any texture data of its own, so the last piece of information we need to track is the list of available animations for the object. These are stored in the `animations` Dictionary object, indexed by the name we will assign to the animation when it is loaded—names like "idle", "run", and "jump". The `currentAnimation` string variable holds the name of the currently playing animation.

## Drawing, animation, and movement

Now that we have the basics of the `GameObject` class, we need to be able to draw the appropriate `AnimationStrip` to the display and allow the object to move.

### Time for action – building the `GameObject` class – part 2

1. Add the `updateAnimation()` helper method to the `GameObject` class:

```
#region Helper Methods
private void updateAnimation(GameTime gameTime)
{
 if (animations.ContainsKey(currentAnimation))
 {
 if (animations[currentAnimation].FinishedPlaying)
 {
 PlayAnimation(animations[currentAnimation].
 NextAnimation);
 }
 }
 else
```

```
 {
 animations[currentAnimation].Update(gameTime);
 }
 }
}
#endregion
```

- 2.** Add a new region called "Public Methods" to the GameObject class:

```
#region Public Methods
#endregion
```

- 3.** Inside the Public Methods region, add the PlayAnimation() method:

```
public void PlayAnimation(string name)
{
 if (!(name==null) && animations.ContainsKey(name))
 {
 currentAnimation = name;
 animations[name].Play();
 }
}
```

- 4.** Still in the Public Methods region, add the Update() method:

```
public virtual void Update(GameTime gameTime)
{
 if (!enabled)
 return;

 float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

 updateAnimation(gameTime);
 if (velocity.Y != 0)
 {
 onGround = false;
 }

 Vector2 moveAmount = velocity * elapsed;
 moveAmount = horizontalCollisionTest(moveAmount);
 moveAmount = verticalCollisionTest(moveAmount);
 Vector2 newPosition = worldLocation + moveAmount;
 newPosition = new Vector2(
 MathHelper.Clamp(newPosition.X, 0,
 Camera.WorldRectangle.Width - frameWidth),
 MathHelper.Clamp(newPosition.Y, 2*(-TileMap.TileHeight),
 Camera.WorldRectangle.Height - frameHeight));
 worldLocation = newPosition;
}
```

**5.** Still in the Public Methods region, add the `Draw()` method:

```
public virtual void Draw(SpriteBatch spriteBatch)
{
 if (!enabled)
 return;

 if (animations.ContainsKey(currentAnimation))
 {
 SpriteEffects effect = SpriteEffects.None;
 if (flipped)
 {
 effect = SpriteEffects.FlipHorizontally;
 }

 spriteBatch.Draw(
 animations[currentAnimation].Texture,
 Camera.WorldToScreen(WorldRectangle),
 animations[currentAnimation].FrameRectangle,
 Color.White, 0.0f, Vector2.Zero, effect, drawDepth);
 }
}
```

### ***What just happened?***

The `updateAnimation()` helper method is called from the `Update()` method itself, and is broken up for the sake of readability. The method first checks to make sure that an animation with the name corresponding to the value of `currentAnimation` actually exists in the `animations` Dictionary. If it does, and the current animation has finished playing, the `updateAnimation()` method will use the `PlayAnimation()` method to begin playing the animation indicated by the `NextAnimation` property of the currently playing `AnimationStrip`.

If the current animation has not completed, its own `Update()` method is called, allowing the `AnimationStrip` to advance to the current frame.

`PlayAnimation()` itself also checks to make sure that whatever value is passed to it is a valid animation for this `GameObject`. If it is, the object's `currentAnimation` is set, and the named animation's `Play()` method is executed.

When the `GameObject`'s `Update()` method is called, it verifies that the object is enabled, and calls the `updateAnimation()` helper method. Next, `Update()` checks to see if the object's current `velocity` has a `Y` component that is not zero. If it does, we know that we cannot be on the ground and set the `onGround` variable to `false`.

In order to determine where our `GameObject` will move during this frame, we multiply the `velocity` (in our standard pixels per second scale) by the `elapsed` game time for this frame, resulting in the movement we wish the `GameObject` to have for the frame.

This movement, however, can be restricted by surrounding game tiles. In order to determine what portions of our movement are available to us, we will call the `horizontalCollisionTest()` and `verticalCollisionTest()` methods, which we will implement in just a moment. The result of these two methods is to modify the `moveAmount` variable to account for any potential collisions, so after they have been tested we can add the move amount to the `GameObject`'s current position.

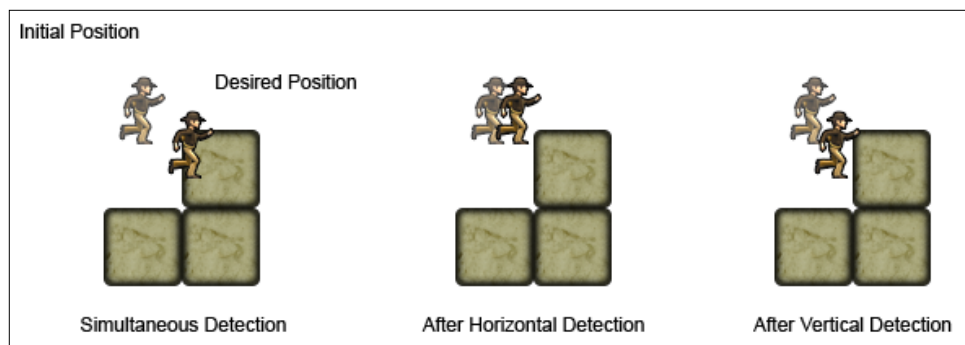
We do not, however, apply it directly. First we create a `Vector2` variable called `newPosition` and then use `MathHelper.Clamp()` to limit the object's position to the area of the game board, with the exception of the top of the screen. By specifying `2 * (-TileMap.TileHeight)` as the minimum value for the vertical component of the `newPosition` vector, we allow the `GameObject` to actually be positioned up to two tiles above the top of the game map. This will allow our player (in a manner very similar to a certain red-clad plumber who has been known to run about in a tile-based world from time to time) to jump slightly above the screen.

When the `GameObject` is drawn, we establish a value for the `SpriteEffects` parameter of the `SpriteBatch.Draw()` call by examining the `flipped` Boolean variable. The texture for the object is retrieved from the currently playing `AnimationStrip`, and the sprite is drawn at the pre-defined drawing depth.

## Map-based collision detection

As we did with *Robot Rampage*, we split map-based collision detection into horizontal and vertical components. We process the horizontal component first, and then the vertical component based upon the results of the horizontal check.

Why perform these checks separately? Consider the following diagram showing the requested movement of the player's character:



If we process both the horizontal and vertical collision checks at the same time, the `GameObject` will become suspended in mid-air, unable to fall downward because the horizontal component of the object's movement would place the object inside a wall. Simply checking the new collision box will result in the entirety of the movement being negated.

By checking the horizontal movement first, the player can bump up against the wall block, stopping his horizontal movement. When the vertical component of the movement is then checked separately, the player can fall downwards toward the floor because the collision box has been adjusted for the new horizontal movement (or lack thereof) and will not result in a wall collision when testing for vertical movement.

## Time for action – building the `GameObject` class – part 3

1. Add the map-based collision detection methods to the `GameObject` class:

```
#region Map-Based Collision Detection Methods
private Vector2 horizontalCollisionTest(Vector2 moveAmount)
{
 if (moveAmount.X == 0)
 return moveAmount;

 Rectangle afterMoveRect = CollisionRectangle;
 afterMoveRect.Offset((int)moveAmount.X, 0);
 Vector2 corner1, corner2;
 if (moveAmount.X < 0)
 {
 corner1 = new Vector2(afterMoveRect.Left,
 afterMoveRect.Top + 1);
 corner2 = new Vector2(afterMoveRect.Left,
 afterMoveRect.Bottom - 1);
 }
 else
 {
 corner1 = new Vector2(afterMoveRect.Right,
 afterMoveRect.Top + 1);
 corner2 = new Vector2(afterMoveRect.Right,
 afterMoveRect.Bottom - 1);
 }
 Vector2 mapCell1 = TileMap.GetCellByPixel(corner1);
 Vector2 mapCell2 = TileMap.GetCellByPixel(corner2);
 if (!TileMap.CellIsPassable(mapCell1) ||
 !TileMap.CellIsPassable(mapCell2))
 {
 moveAmount.X = 0;
 velocity.X = 0;
 }
}
```



```
 }
 if (codeBasedBlocks)
 {
 if (TileMap.CellCodeValue(mapCell1) == "BLOCK" ||
 TileMap.CellCodeValue(mapCell2) == "BLOCK")
 {
 moveAmount.X = 0;
 velocity.X = 0;
 }
 }
 return moveAmount;
}

private Vector2 verticalCollisionTest(Vector2 moveAmount)
{
 if (moveAmount.Y == 0)
 return moveAmount;

 Rectangle afterMoveRect = CollisionRectangle;
 afterMoveRect.Offset((int)moveAmount.X, (int)moveAmount.Y);
 Vector2 corner1, corner2;
 if (moveAmount.Y < 0)
 {
 corner1 = new Vector2(afterMoveRect.Left + 1,
 afterMoveRect.Top);
 corner2 = new Vector2(afterMoveRect.Right - 1,
 afterMoveRect.Top);
 }
 else
 {
 corner1 = new Vector2(afterMoveRect.Left + 1,
 afterMoveRect.Bottom);
 corner2 = new Vector2(afterMoveRect.Right - 1,
 afterMoveRect.Bottom);
 }

 Vector2 mapCell1 = TileMap.GetCellByPixel(corner1);
 Vector2 mapCell2 = TileMap.GetCellByPixel(corner2);
 if (!TileMap.CellIsPassable(mapCell1) ||
 !TileMap.CellIsPassable(mapCell2))
 {
 if (moveAmount.Y > 0)
 onGround = true;
 moveAmount.Y = 0;
 velocity.Y = 0;
 }
 if (codeBasedBlocks)
 {
```

---

```

 if (TileMap.CellCodeValue(mapCell1) == "BLOCK" ||
 TileMap.CellCodeValue(mapCell2) == "BLOCK")
 {
 if (moveAmount.Y > 0)
 onGround = true;
 moveAmount.Y = 0;
 velocity.Y = 0;
 }
 }
 return moveAmount;
}
#endregion

```

### ***What just happened?***

Our collision detection methods begin by building a `Rectangle` that represents the location that the `GameObject` will be in if the requested movement is completed. They do this by taking a copy of the current `CollisionRectangle` property for the object and offsetting it by the movement amount.

In the case of horizontal movement, which is checked first, only the `x` component of the movement is considered. When the `afterMoveRect` is constructed for the vertical movement check, we also apply the horizontal movement, since it will have been corrected for horizontal collision detection already.

In order to actually test for collisions, we only need to check the two outermost pixels of the new movement area. For example, in the previous diagram, the player's character is moving to the right. By checking the upper-right and lower-right pixels during the horizontal collision detection method, we will cover moving right into any impassable map square, because our character is the same size as a map tile.

After determining the pixel locations of these two corners, we can retrieve their tile map values and check `CellIsPassable()` to determine if the square should block movement. If this game entity is also blocked by invisible `CodeValue` blocks (`codeBasedBlocks` is `true`) we also check to see if the squares we are testing against contain the code `BLOCK` and disallow movement into them if so.

We have one special case to handle for vertical movement—setting the value of `onGround` when the player is standing on a surface. By setting this value to `true` whenever we detect an impassable block while travelling in a positive `y` direction, we can indicate to later code that the player can jump from their current position.

## The Player

Now that we have the `GameObject` class to base the components of our game world on, we can begin creating the entities that will populate the game. We will start with the player, and put enough temporary code into place in the `Game1` class to move and animate the brown-clad archaeologist adventurer.

### Time for action – creating the Player class

1. Add a new class file called `Player.cs` to the Gemstone Hunter project.

2. Add the following using directives to the Player class:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Input;
using Tile_Engine;
```

3. Modify the declaration of the Player class to make it public, and derive from the `GameObject` class:

```
public class Player : GameObject
```

4. Add declarations to the Player class:

```
private Vector2 fallSpeed = new Vector2(0, 20);
private float moveScale = 180.0f;
private bool dead = false;
```

5. Add the `Dead` property to the Player class:

```
public bool Dead
{
 get { return dead; }
}
```

6. Create a constructor for the Player class:

```
#region Constructor
public Player(ContentManager content)
{
 animations.Add("idle",
 new AnimationStrip(
 content.Load<Texture2D>(@"Textures\Sprites\Player\
 Idle"),
 48,
```

```

 "idle"));
animations["idle"].LoopAnimation = true;
animations.Add("run",
 new AnimationStrip(
 content.Load<Texture2D>(@"Textures\Sprites\Player\ Run"),
 48,
 "run"));
animations["run"].LoopAnimation = true;
animations.Add("jump",
 new AnimationStrip(
 content.Load<Texture2D>(@"Textures\Sprites\Player\ Jump"),
 48,
 "jump"));
animations["jump"].LoopAnimation = false;
animations["jump"].FrameLength = 0.08f;
animations["jump"].NextAnimation = "idle";
animations.Add("die",
 new AnimationStrip(
 content.Load<Texture2D>(@"Textures\Sprites\Player\Die"),
 48,
 "die"));
animations["die"].LoopAnimation = false;
frameWidth = 48;
frameHeight = 48;
CollisionRectangle = new Rectangle(9, 1, 30, 46);
drawDepth = 0.825f;
enabled = true;
codeBasedBlocks = false;
PlayAnimation("idle");
}
#endregion

```

- 7.** In the `Game1.cs` class for the Gemstone Hunter project, add a declaration for an instance of the `Player` class:
 

```
Player player;
```
- 8.** In the `Initialize()` method of the `Game1` class, set the window size for the game:
 

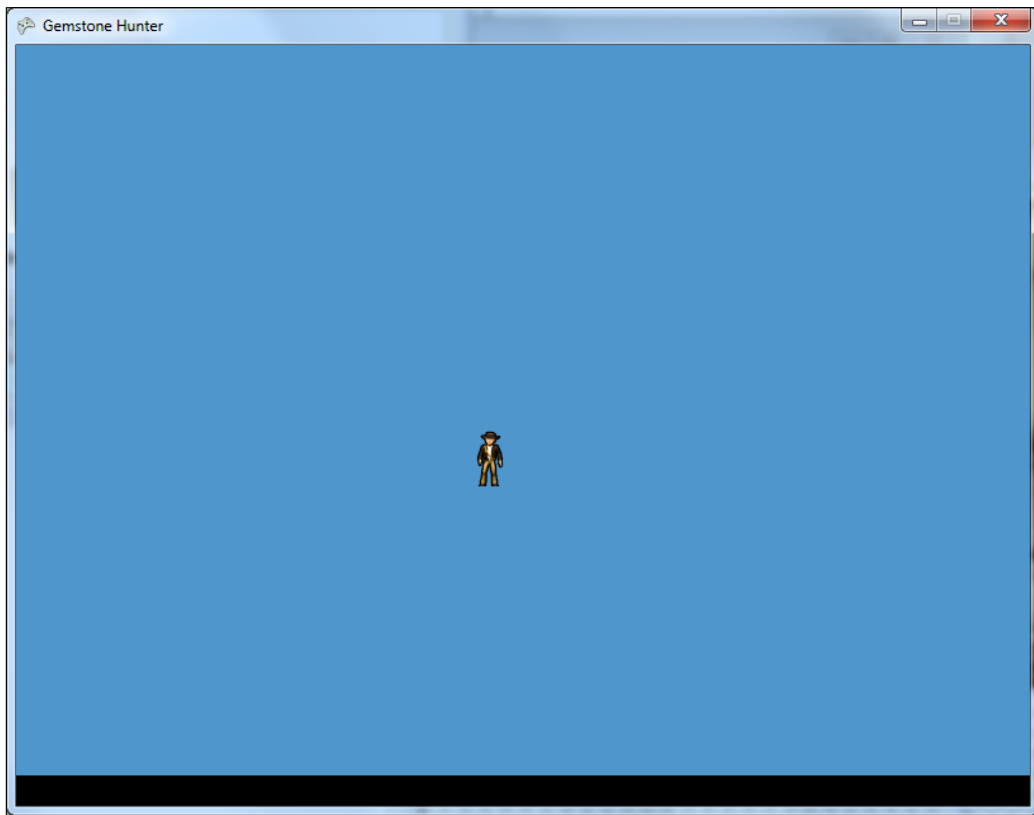
```
this.graphics.PreferredBackBufferWidth = 800;
this.graphics.PreferredBackBufferHeight = 600;
this.graphics.ApplyChanges();
```

- 9.** In the `LoadContent()` method of the `Game1` class that was carried over from Chapter 8, remove the line that reads `TileMap.SetTileAtCell(3,3,1,10);`
- 10.** In the `LoadContent()` method of the `Game1` class, add the following to initialize the camera and the player object:  

```
player = new Player(Content);
player.WorldLocation = new Vector2(350, 300);
```
- 11.** Add the following to the `Update()` method of the `Game1` class:  

```
player.Update(gameTime);
```
- 12.** Add the following code to the `Draw()` method of the `Game1` class inside the existing `spriteBatch.Begin()` and `spriteBatch.End()` block:  

```
player.Draw(spriteBatch);
```
- 13.** Right-click on the Gemstone Hunter project in Solution Explorer and select **Set as StartUp Project**.
- 14.** Execute the Gemstone Hunter project to view the player object:



## ***What just happened?***

Our Player class currently needs only three member variables beyond those that already exist in the `GameObject` class—`fallSpeed` to determine the rate of acceleration when the player is falling, `moveScale` to determine how fast the player runs when moving along the X axis, and `dead` to flag when the character has been killed. We will be adding a few more items to this list in later sections as we expand on the features of the game.

The constructor for the Player class does all of the work of initializing the various `AnimationStrips` used by the class, including an animation for running, jumping, player death, and standing idle. Individual properties are set on these animations based on how we wish to play them.

For example, the idle and running animations have their `LoopAnimation` property set to true. While these animations are active, we want them to cycle in a continuous loop until we set them to something different. Because these animations loop, there is no need to specify a `NextAnimation` value. It will never be activated.

The jumping animation, on the other hand, should only be played once when the player jumps. After completing, the animation will default back to idle, assuming the player is not pressing any movement keys at the time the animation ends.

Finally, the animation named `die` neither loops, nor has a `NextAnimation` setting. In this case, we want the animation to play once, and then continue to display the final frame of the player lying on the ground until we take action to respawn the player, reload the level, or transition to a "Game Over" state.

In steps 7 through 12, we implement just enough code to allow us to display the player object to the screen, setting up the game world, and creating the player object inside it. Of course, at the moment the player cannot move, run, jump, or animate!

## **Running...**

We will override the `GameObject`'s `Update()` method in the Player class to handle user input and handle switching between animations as appropriate. We will build the `Update()` method in several stages, adding functionality with each iteration.

### **Time for action – overriding the `Update()` method – part 1**

1. Create a new region in the Player class for public methods:

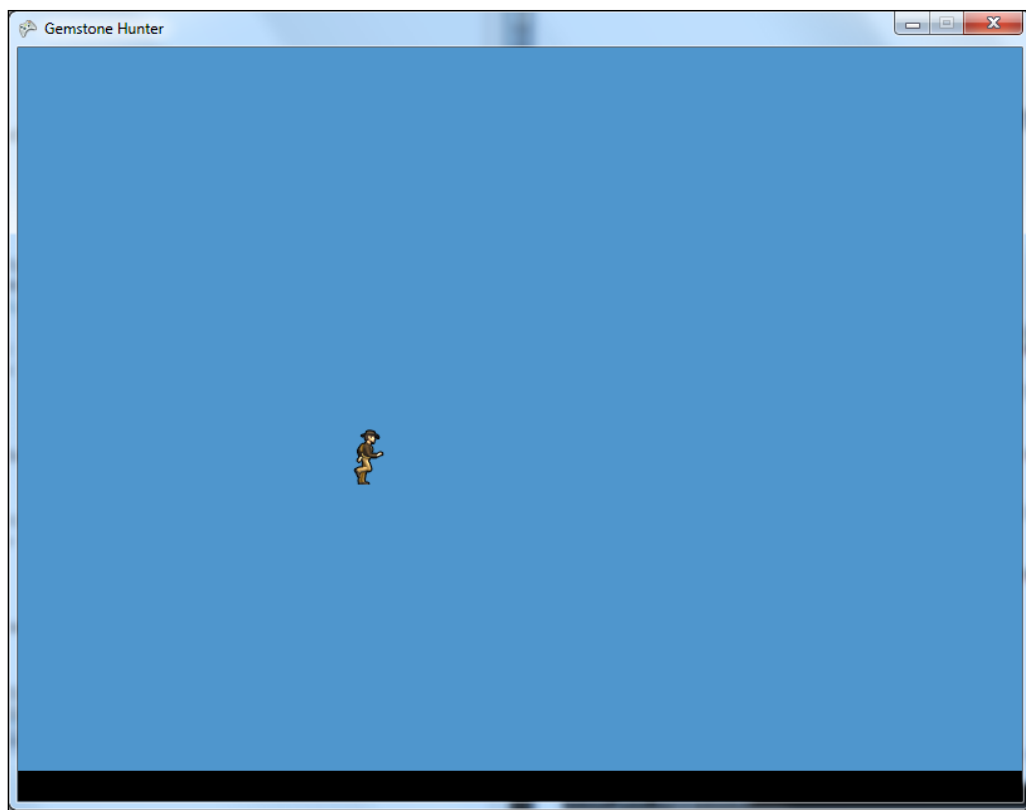
```
#region Public Methods
#endregion
```

- 2.** Add the initial `Update()` override method to the Public Methods region of the Player class:

```
public override void Update(GameTime gameTime)
{
 if (!Dead)
 {
 string newAnimation = "idle";
 velocity = new Vector2(0, velocity.Y);
 GamePadState gamePad = GamePad.GetState(PlayerIndex.One);
 KeyboardState keyState = Keyboard.GetState();
 if (keyState.IsKeyDown(Keys.Left) ||
 (gamePad.ThumbSticks.Left.X < -0.3f))
 {
 flipped = false;
 newAnimation = "run";
 velocity = new Vector2(-moveScale, velocity.Y);
 }
 if (keyState.IsKeyDown(Keys.Right) ||
 (gamePad.ThumbSticks.Left.X > 0.3f))
 {
 flipped = true;
 newAnimation = "run";
 velocity = new Vector2(moveScale, velocity.Y);
 }
 if (newAnimation != currentAnimation)
 {
 PlayAnimation(newAnimation);
 }
 }

 base.Update(gameTime);
}
```

- 3.** Execute the Gemstone Hunter project and move the player left and right:



### ***What just happened?***

The animation that the player object will display will be determined by the current input state. By default, we will assume that there is no input and the animation should be changed to `idle`.

We then zero out the `x` component of the player's `velocity`, again assuming that there is no input and, therefore, no lateral movement. The state of the gamepad and the keyboard are captured and then the current input values are evaluated.

If the player indicates via the keyboard or the gamepad that they wish to move `left` or `right`, we set the `newAnimation` variable to `run` and set the `x` component of the `velocity` vector to the `moveScale` value, negating it for `left` movement and leaving the value positive for `right` movement.

When moving `left`, the `flipped` variable is set to `false`. Moving `right` sets the variable to `true`. If neither `left` nor `right` movement is made, the value of `flipped` is left unchanged. Recall that when the `AnimationStrip` is drawn, `flipped` will be used to produce both left and right images from the same bitmap.



Finally, right before calling `base.Update()` to allow the normal `GameObject` update process to take place, the value that we wish to assign for animation is checked against the animation that is currently playing. If they are already the same, no change is made. If the `newAnimation` value is different from `currentAnimation`, the new animation is played. Without this check, a player running would always display the first frame of the running animation, appearing to glide without animating because the animation would be restarted (by the `Play()` method) in every frame.

## ...and jumping

Currently our player begins near the center of an empty screen and can run left or right but has no vertical movement capability. In order to allow the player to jump, we will also need to allow the player to fall.

### Time for action – overriding the `Update()` method – part 2

1. Inside the Public Methods region of the `Player` class, add the `Jump()` method:

```
public void Jump()
{
 velocity.Y = -500;
}
```

2. Add the following to the `Update()` method right after the `if` statement for detecting movement to the right, and before the check for which animation to play:

```
if (keyState.IsKeyDown(Keys.Space) ||
 (gamePad.Buttons.A == ButtonState.Pressed))
{
 if (onGround)
 {
 Jump();
 newAnimation = "jump";
 }
}
```

```
if (currentAnimation == "jump")
 newAnimation = "jump";
```

3. Still in the `Update()` method, add the following line right before the call to `base.Update()` (outside of the conditional for the player being dead—being dead will not prevent the player's body from falling due to gravity):

```
velocity += fallSpeed;
```

4. Execute the Gemstone Hunter project. Use the *Space bar* or the *A* button on the gamepad to jump.

### ***What just happened?***

When you execute the game, the player will fall with increasing speed and land on the bottom of the window. Even though we are not currently displaying a map, the bounds of the map are considered impassable.

Our simple simulation of gravity (adding 20 downward pixels per second to the player's current velocity) means that all we need to do to initiate a jump is to set the *y* component of the player's velocity to a larger negative number. As the player rises, our gravity will slow them down, eventually stopping and reversing the movement upward.

Finally, notice that no matter what we have indicated from other movement commands, the jump animation will override them. Additionally, if the jump animation is already playing (`currentAnimation == "jump"`), we also set `newAnimation` to `jump`, so as not to interrupt it.

The `jump` animation has a `NextAnimation` property, which will ensure that it runs once and then returns to `idle`. Just as before, this check is to make the animation appear smoothly. Without it, the first frame of the `jump` animation would play, and if the player were pressing the left or right arrow keys at that time, the `run` animation would take over, preventing the full `jump` animation from playing. The full `jump` animation looks like this:



### **Staying on the screen**

One issue that is evident right away is that you can simply run straight off the right edge of the screen, and the game's camera does not follow your character. We can fix this by adding a helper method to the player class:

### **Time for action – repositioning the camera**

1. Add the Helper Methods region and the `repositionCamera()` method to the `Player` class:

```
#region Helper Methods
private void repositionCamera()
{
 int screenLocX = (int)Camera.WorldToScreen(worldLocation).X;
```

```
 if (screenLocX > 500)
 {
 Camera.Move(new Vector2(screenLocX - 500, 0));
 }
 if (screenLocX < 200)
 {
 Camera.Move(new Vector2(screenLocX - 200, 0));
 }
 }
#endregion
```

2. In the `Update()` method of the `Player` class, add a call to reposition the camera right before the call to `base.Update()`:  
`repositionCamera();`
3. Execute the `Gemstone Hunter` application and move towards the right side of the screen.

## ***What just happened?***

During each update frame, the current screen position of the character is checked. If the character has gotten too close to the edge of the screen (200 pixels from the left edge, or 300 pixels from the right edge), the camera's position is adjusted to keep the character within those bounds.

If the camera is already as far in either direction as it can go, nothing will happen—this behavior is built into the `Camera` class. This allows the character to move to the edge of the map instead of requiring us to impose some kind of artificial barrier in the game world.

At the moment, however, we are still running around on a huge empty expanse. It may be a bit difficult to recognize that the camera is in fact scrolling when you get near the edge of the screen. To resolve that problem, we need to begin displaying our level map.

## **Loading levels**

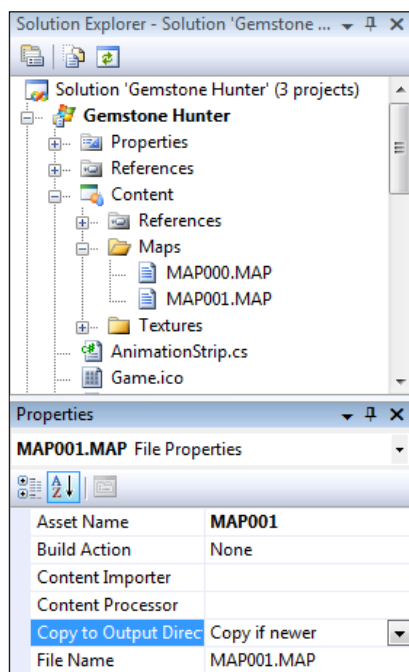
Included in the content package for the `Gemstone Hunter` game are a few `.MAP` files in their own folder. These maps were created using the `Level Editor`, and we will use them to implement level loading. Feel free to replace these files with levels of your own design!

## **The LevelManager class**

Initially, our `LevelManager` class will only be responsible for loading `MAP` files into the `TileEngine` class. As we add entities (enemies and gemstones) to the game, `LevelManager` will be expanded to track and control them as well.

## Time for action – building the LevelManager class

1. Right-click on the Gemstone Hunter Content project and add a new folder called Maps.
2. Add the MAP001.MAP and MAP002.MAP sample maps from 0669\_08\_GRAPHICPACK to the Maps folder.
3. Click on each .MAP file in Solution Explorer and, in the **Properties** window, set the **Build Action** to **None** and the **Copy to Output Directory** property to **Copy if newer**.



4. Add a new class file called `LevelManager.cs` to the Gemstone Hunter project.
5. Add the following using directives to the `LevelManager` class:

```
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Tile_Engine;
```

- 6.** Modify the declaration of the `LevelManager` class to make it `public` and `static`:

```
public static class LevelManager
```

- 7.** Add declarations to the `LevelManager` class:

```
#region Declarations
private static ContentManager Content;
private static Player player;
private static int currentLevel;
private static Vector2 respawnLocation;
#endregion
```

- 8.** Add properties to the `LevelManager` class to access the `currentLevel` and `respawnLocation` variables:

```
#region Properties
public static int CurrentLevel
{
 get { return currentLevel; }
}

public static Vector2 RespawnLocation
{
 get { return respawnLocation; }
 set { respawnLocation = value; }
}
#endregion
```

- 9.** Add the `Initialize()` method to the `LevelManager` class:

```
#region Initialization
public static void Initialize(
 ContentManager content,
 Player gamePlayer)
{
 Content = content;
 player = gamePlayer;
}
#endregion
```

- 10.** Create a new region in the `LevelManager` class for Public Methods:

```
#region Public Methods
#endregion
```

- 11.** Add the `LoadLevel()` method to the Public Methods region of the `LevelManager` class:

```
public static void LoadLevel(int levelNumber)
{
 TileMap.LoadMap((System.IO.FileStream)TitleContainer.
 OpenStream(
 @"Content\Maps\MAP" +
 levelNumber.ToString().PadLeft(3, '0') + ".MAP"));
 for (int x = 0; x < TileMap.MapWidth; x++)
 {
 for (int y = 0; y < TileMap.MapHeight; y++)
 {
 if (TileMap.CellCodeValue(x, y) == "START")
 {
 player.WorldLocation = new Vector2(
 x * TileMap.TileWidth,
 y * TileMap.TileHeight);
 }
 }
 }
 currentLevel = levelNumber;
 respawnLocation = player.WorldLocation;
}
```

- 12.** Add the `ReloadLevel()` method to the Public Methods region of the `LevelManager` class:

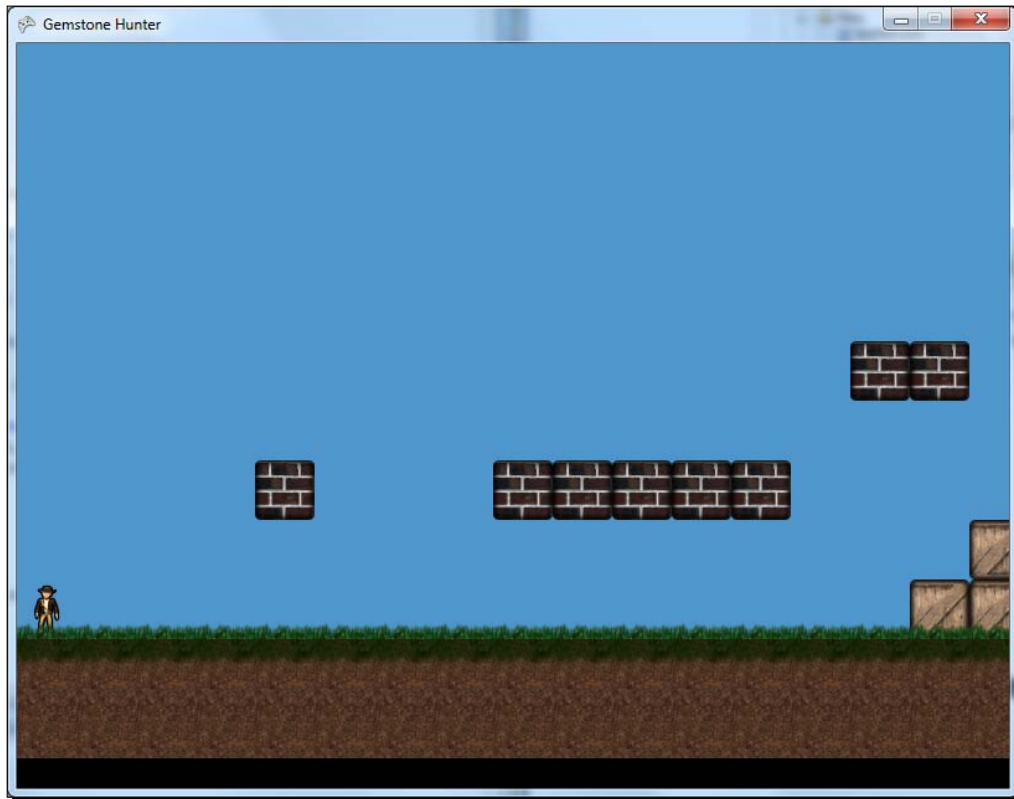
```
public static void ReloadLevel()
{
 Vector2 saveRespawn = respawnLocation;
 LoadLevel(currentLevel);
 respawnLocation = saveRespawn;
 player.WorldLocation = respawnLocation;
}
```

- 13.** In the `LoadContent()` method of the `Game1` class, remove the line that positions the player (`player.WorldLocation = new Vector2(350, 300);`).

- 14.** Modify the `LoadContent()` method of the `Game1` class by adding the following to the end of the method:

```
LevelManager.Initialize(Content, player);
LevelManager.LoadLevel(0);
```

**15.** Execute the Gemstone Hunter application:



***What just happened?***

The `LevelManager` holds on to references for the game's `ContentManager` instance and the active instance of the `Player` class. In the case of the player, we need this reference so that we can position the player after a level has been loaded.

The `ContentManager` instance is not used yet, but we will be passing it along to our `Enemy` and `Gemstone` classes when we create instances of them shortly.

To load a map, we use XNA's `TitleContainer` object to retrieve a stream located within the `Content` folder deployed with our game. We pass this stream to the `TileMap`'s `LoadMap()` method, causing it to load the map for us. The `MAP` files produced by the `Level Editor` project are named `MAPXXX.MAP`, where `XXX` represents a three digit number. We use the `PadLeft()` method to fill in any missing leading zeros when determining the name of the map file to load.

After `LoadMap()` has finished, the `LoadLevel()` method examines each square on the map, looking for a `START` code value. If it finds one, the player will be moved to that location on the map. We will expand on this loop later to spawn enemies and gemstones when a level loads.



#### Layer depth and draw order

In the previous section, when we added the `Player.Draw()` call to the `Game1` class's `Draw()` method, we did not specify where to place it in relation to the existing `TileMap.Draw()` call that was already there, because we are using layer depth sorting. If we draw the map after the player, the player will still appear above the background and interactive layers as we would expect.

Keep in mind, however, that when multiple items with the same layer depth are drawn, the most recently drawn object will appear above any previously drawn objects at that layer depth, just like the default functionality when layer depth sorting is not used and all sprites are drawn at `0.0f`.

We keep track of the current level and a vector location at which to respawn the player so that we can reload the level if the player dies. Since door transitions will be able to place the player at a location other than that marked with a `START` level code, we track the respawn location separately so we can restart the level at the place the player entered it.

## Gemstones

As Gemstone Hunter stands right now, you can explore your first world map, but there is nothing in it. Any `GEM` and `ENEMY` codes assigned to map squares currently do nothing inside the game itself.

The first non-player object we will add to the game world will be the gemstones that the main character is seeking to collect while playing.

### Time for action – building the Gemstone class

1. Add a new class file called `Gemstone.cs` to the Gemstone Hunter project.
2. Add the following using directives to the Gemstone class:
 

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Tile_Engine;
```
3. Modify the declaration of the Gemstone class to make the class public, and derive it from the `GameObject` class:
 

```
public class Gemstone : GameObject
```



**4.** Add a constructor for the Gemstone class:

```
#region Constructor
public Gemstone(ContentManager Content, int cellX, int cellY)
{
 worldLocation.X = TileMap.TileWidth * cellX;
 worldLocation.Y = TileMap.TileHeight * cellY;
 frameWidth = TileMap.TileWidth;
 frameHeight = TileMap.TileHeight;
 animations.Add("idle",
 new AnimationStrip(
 Content.Load<Texture2D>(@"Textures\Gem"),
 48,
 "idle"));
 animations["idle"].LoopAnimation = true;
 animations["idle"].FrameLength = 0.15f;
 PlayAnimation("idle");
 drawDepth = 0.875f;
 CollisionRectangle = new Rectangle(9, 24, 30, 24);
 enabled = true;
}
#endregion
```

**5.** Back in the LevelManager class, add a declaration to hold a list of Gemstones:

```
private static List<Gemstone> gemstones = new List<Gemstone>();
```

**6.** In the LoadLevel() method of the LevelManager class, right after the call to TileMap.LoadMap(), clear the gemstones list:

```
gemstones.Clear();
```

**7.** Still in the LoadLevel() method, add a condition to the loop that examines the code values in each square to check for GEM codes. This can be placed right after the condition for START codes:

```
if (TileMap.CellCodeValue(x, y) == "GEM")
{
 gemstones.Add(new Gemstone(Content, x, y));
}
```

**8.** Add a new Update() method to the LevelManager class:

```
public static void Update(GameTime gameTime)
{
 foreach (Gemstone gemstone in gemstones)
 {
 gemstone.Update(gameTime);
 }
}
```

- 9.** Add a new `Draw()` method to the `LevelManager` class:

```
public static void Draw(SpriteBatch spriteBatch)
{
 foreach (Gemstone gem in gemstones)
 gem.Draw(spriteBatch);
}
```

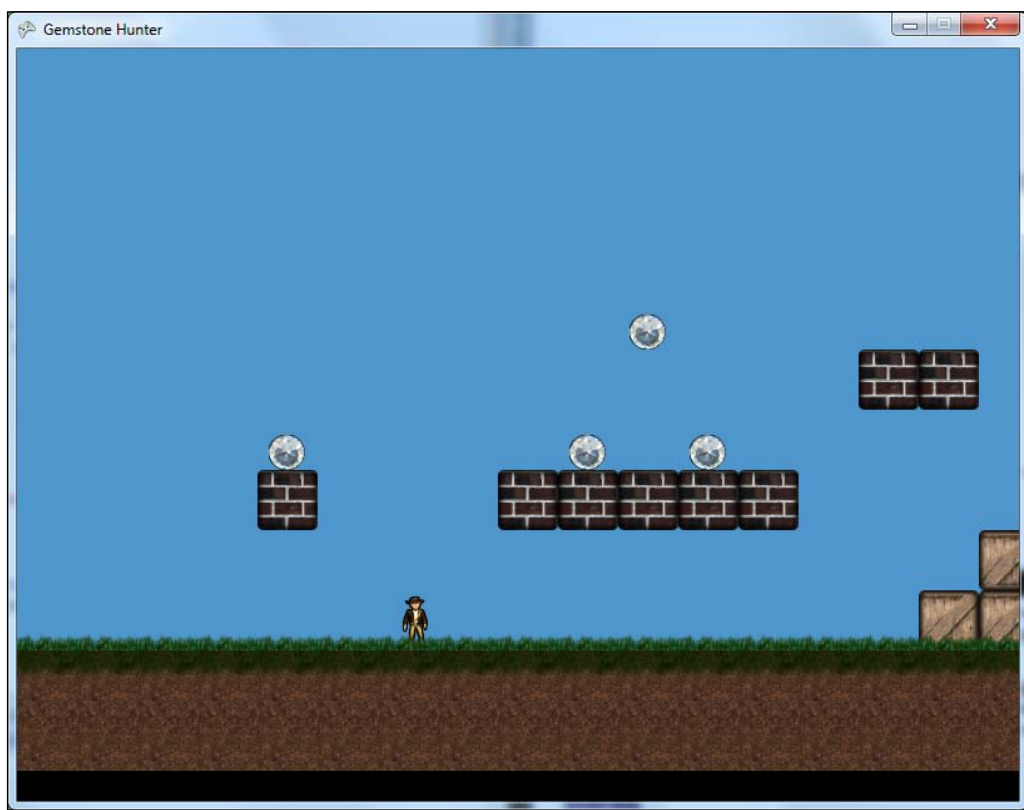
- 10.** In the `Game1` class, call the `LevelManager`'s `Update()` method after the player has been updated:

```
LevelManager.Update(gameTime);
```

- 11.** Still in the `Game1` class, modify the `Draw()` method to include a call to draw the level manager right after the player has been drawn:

```
LevelManager.Draw(spriteBatch);
```

- 12.** Execute the `Gemstone Hunter` application:



## ***What just happened?***

Because our `GameObject` class provides all of the behavior we need for gemstones, our `Gemstone` class only needs a constructor, which will load the appropriate `AnimationStrip` and initialize its location and collision area.

We have updated the `LoadLevel()` method in the `LevelManager` class to scan the level for GEM code values and create gems at these locations. Since the `Gemstone` class defines no movement capabilities, the gemstones will remain stationary in the blocks they spawn in, not subject to the whims of gravity.

## **Scoring**

The player needs to be able to pick up the gemstones, and receive points for doing so. In order to allow for this, we will need to make a handful of additions to the `Player` class, and further modify the `LevelManager` class to detect collisions between the player and the gemstones spawned in the world.

### **Time for action – implementing score tracking**

1. Add a declaration to the `Player` class to store the player's score:

```
private int score = 0;
```

2. Add a property to the `Player` class to allow access to the score variable:

```
public int Score
{
 get { return score; }
 set { score = value; }
}
```

3. Replace the current `Update()` method in the `LevelManager` class with the following new version of the method:

```
public static void Update(GameTime gameTime)
{
 if (!player.Dead)
 {
 for (int x = gemstones.Count - 1; x >= 0; x--)
 {
 gemstones[x].Update(gameTime);
 if (player.CollisionRectangle.Intersects(
 gemstones[x].CollisionRectangle))
 {
 gemstones.RemoveAt(x);
 player.Score += 10;
 }
 }
 }
}
```

```

 }
 }
}

```

4. In the `Game1` class, add a declaration for a `SpriteFont` instance that we can use to draw the player's score and a vector pointing to the location on the screen where the score will be displayed:

```

SpriteFont pericles8;
Vector2 scorePosition = new Vector2(20, 580);

```

5. In the `LoadContent()` method of the `Game1` class, initialize the `pericles8` font:

```
pericles8 = Content.Load<SpriteFont>(@"Fonts\Pericles8");
```

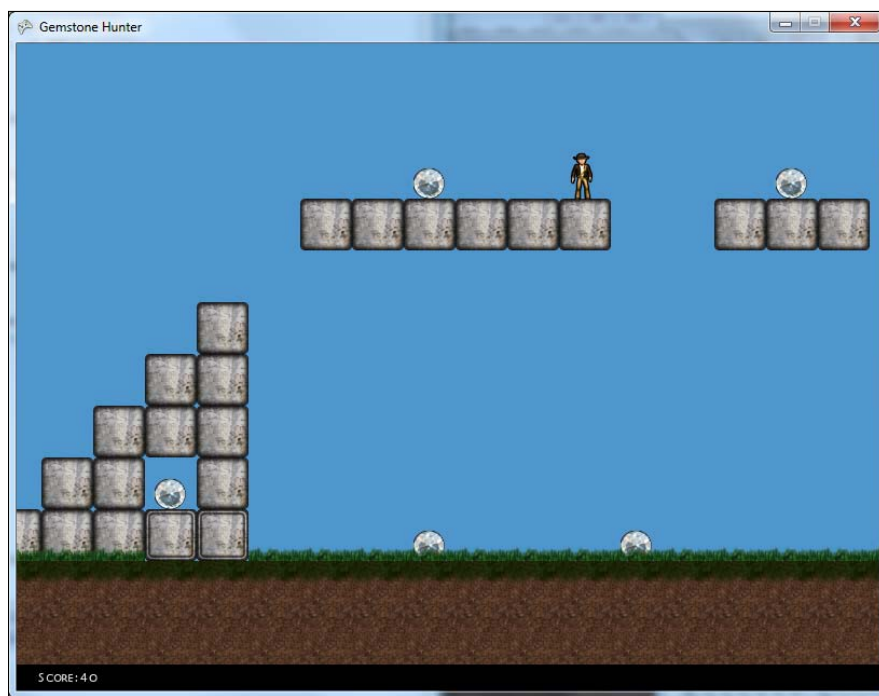
6. In the `Draw()` method of the `Game1` class, add a call to display the current score right before the `spriteBatch.End()` call:

```

spriteBatch.DrawString(
 pericles8,
 "Score: " + player.Score.ToString(),
 scorePosition,
 Color.White);

```

7. Execute the `Gemstone Hunter` application and collect a few gemstones:



## What just happened?

When called, the `LevelManager.Update()` method checks each of the gemstones that exist for the current level and compares their `CollisionRectangle` properties to the same property of the `player` object. If these rectangles intersect, the gemstone is removed from the `gemstones` list and the player's score is incremented. Remember that we need to use a reverse-running `for` loop here because we can potentially modify the contents of the list, and a `foreach` loop will throw an exception if the list is modified during the loop.

## Enemies

Gemstone Hunter is pretty easy right now. You are free to run around collecting all of the gems you want at your own pace with no chance of ever triggering the dreaded "die" animation. Not for much longer, though—time to add some zombies to the mix!

As with the `Player` and `Gemstone` classes, the `Enemy` class will be derived from the `GameObject` class, allowing it to exist within the world and, like the `Player` class, detect collisions with the tile map.

### Time for action – summoning up the zombies

1. Add a new class called `Enemy` to the Gemstone Hunter project.
2. Add the following `using` directives to the `Enemy` class:  

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Tile_Engine;
```
3. Modify the class declaration for the `Enemy` class to make it public and derive it from the `GameObject` class:

```
public class Enemy : GameObject
```

4. Add declarations for the `Enemy` class:  

```
private Vector2 fallSpeed = new Vector2(0, 20);
private float walkSpeed = 60.0f;
private bool facingLeft = true;
public bool Dead = false;
```

5. Add a constructor for the `Enemy` class:  

```
#region Constructor
public Enemy(ContentManager content, int cellX, int cellY)
{
 animations.Add("idle",
```

```

 new AnimationStrip(
 content.Load<Texture2D>(
 @"Textures\Sprites\MonsterC\Idle"),
 48,
 "idle"));
 animations["idle"].LoopAnimation = true;
 animations.Add("run",
 new AnimationStrip(
 content.Load<Texture2D>(
 @"Textures\Sprites\MonsterC\Run"),
 48,
 "run"));
 animations["run"].FrameLength = 0.1f;
 animations["run"].LoopAnimation = true;
 animations.Add("die",
 new AnimationStrip(
 content.Load<Texture2D>(
 @"Textures\Sprites\MonsterC\Die"),
 48,
 "die"));
 animations["die"].LoopAnimation = false;
 frameWidth = 48;
 frameHeight = 48;
 CollisionRectangle = new Rectangle(9, 1, 30, 46);
 worldLocation = new Vector2(
 cellX * TileMap.TileWidth,
 cellY * TileMap.TileHeight);
 enabled = true;
 codeBasedBlocks = true;
 PlayAnimation("run");
 }
#endregion

```

## 6. Override the Update() method in the Enemy class:

```

#region Public Methods
public override void Update(GameTime gameTime)
{
 Vector2 oldLocation = worldLocation;
 if (!Dead)
 {
 velocity = new Vector2(0, velocity.Y);
 Vector2 direction = new Vector2(1, 0);
 flipped = true;
 if (facingLeft)
 {
 direction = new Vector2(-1, 0);
 flipped = false;
 }
 }
}

```

```
 direction *= walkSpeed;
 velocity += direction;
 velocity += fallSpeed;
 }
 base.Update(gameTime);
 if (!Dead)
 {
 if (oldLocation == worldLocation)
 {
 facingLeft = !facingLeft;
 }
 }
 else
 {
 if (animations[currentAnimation].FinishedPlaying)
 {
 enabled = false;
 }
 }
}
#endregion
```

- 7.** Add a new list declaration to the LevelManager class to handle Enemy objects:

```
private static List<Enemy> enemies = new List<Enemy>();
```

- 8.** In the LoadLevel() method of the LevelManager class, clear the enemies list right after the call to gemstones.Clear():

```
enemies.Clear();
```

- 9.** In the LoadLevel() method of the LevelManager class, add a new if statement section to the loop that currently examines each square for START and GEM code values. Place this section after the check for GEM codes:

```
if (TileMap.CellCodeValue(x, y) == "ENEMY")
{
 enemies.Add(new Enemy(Content, x, y));
}
```

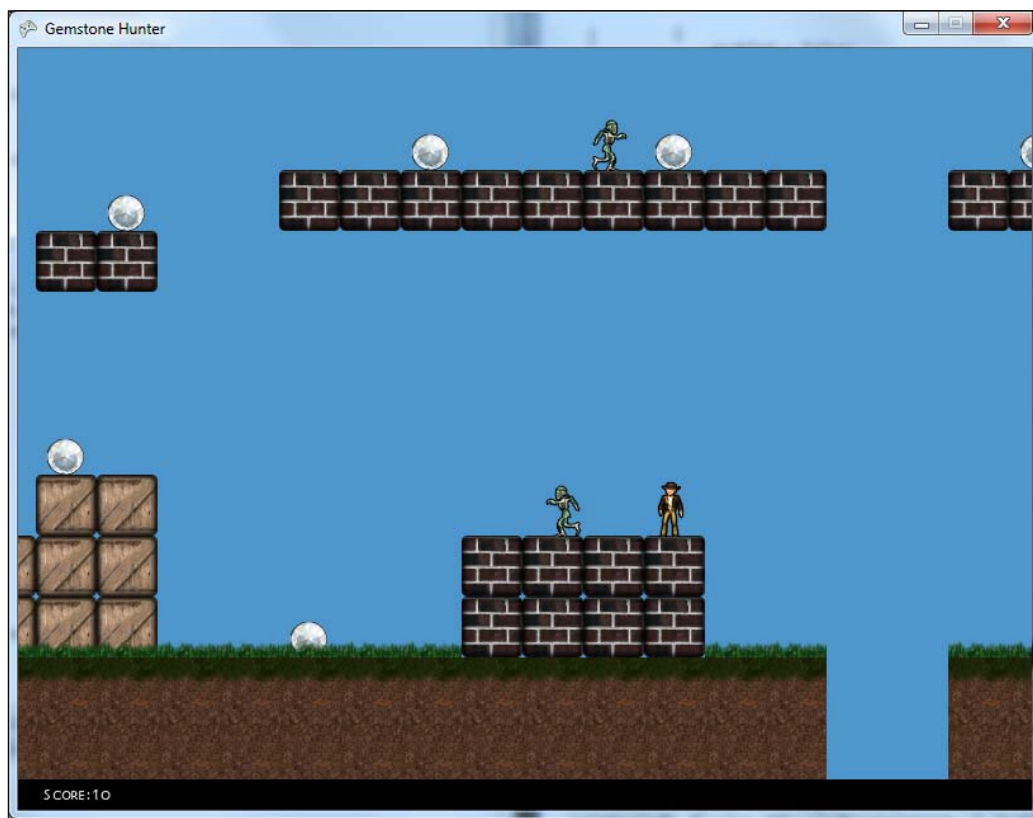
- 10.** Modify the Update() method of the LevelManager to update enemies. Add this code below the loop that updates the gemstones list, but inside the if (!player.Dead) code block:

```
foreach (Enemy enemy in enemies)
 enemy.Update(gameTime);
```

11. Modify the `Draw()` method of the `LevelManager` class by adding a loop to draw each enemy after the gemstones have been drawn:

```
foreach (Enemy enemy in enemies)
 enemy.Draw(spriteBatch);
```

12. Execute the Gemstone Hunter application and observe the enemies as they move through the level:



### ***What just happened?***

The construction of the `Enemy` class is very similar to the `Player` class we built earlier, except that we do not need to handle input for enemies. All of our enemies utilize a very simple artificial intelligence: walk in a direction until you bump into something, and then turn around.

During the `Update()` method, we store the enemy's current location before any movement takes place. Assuming the enemy is not dead, we then generate a vector pointing in the direction of the zombie's movement— $(1, 0)$  if facing right, and  $(-1, 0)$  if facing left.



We multiply the direction vector by the enemy's walk speed (slower than the player—it is a zombie, after all) and add their directional movement to their velocity. We then add the fall acceleration due to gravity to their vertical velocity.

The next step in the `Update()` method is to call `base.Update()` to allow the `GameObject`'s `Update()` code to process the enemy's movement. Again checking to make sure that the enemy is not dead, we compare the enemy's position after `base.Update()` to the position we stored at the beginning of the method. If the position is the same, we know that the enemy's movement was halted by running into a wall, and negate the `facingLeft` variable to turn the enemy around.

If the enemy happens to be dead, we check the currently playing animation to see if it has finished yet. If it has, we know that the "die" animation, which we will trigger when the player stomps on an enemy, has finished, and we can disable the enemy to remove it from the game. In order for this animation to ever be played, however, we need to be able to determine when an enemy is killed.

## Player-enemy interaction

Again drawing inspiration from classic platform games, we will decide that when a player collides with an enemy, if the player is landing on the enemy from above, then enemy will be killed. Otherwise, the enemy will kill the player.

### Time for action – interacting with zombies

1. In the `Update()` method of the `LevelManager` class, replace the `foreach` loop that updates the enemies list entries with the following loop:

```
for (int x = enemies.Count - 1; x >= 0; x--)
{
 enemies[x].Update(gameTime);
 if (!enemies[x].Dead)
 {
 if (player.CollisionRectangle.Intersects(
 enemies[x].CollisionRectangle))
 {
 if (player.WorldCenter.Y < enemies[x].WorldLocation.Y)
 {
 player.Jump();
 player.Score += 5;
 enemies[x].PlayAnimation("die");
 enemies[x].Dead = true; ;
 }
 else
```

```

 {
 player.Kill();
 }
 }
}
else
{
 if (!enemies[x].Enabled)
 {
 enemies.RemoveAt(x);
 }
}
}

```

- 2.** In the Player class, add a declaration to hold the lives the player has remaining:

```
private int livesRemaining = 3;
```

- 3.** Still in the Player class, add a property to access the livesRemaining value:

```

public int LivesRemaining
{
 get { return livesRemaining; }
 set { livesRemaining = value; }
}

```

- 4.** Add the Kill() method to the Player class to allow the player to be killed:

```

public void Kill()
{
 PlayAnimation("die");
 LivesRemaining--;
 velocity.X = 0;
 dead = true;
}

```

- 5.** Add the Revive() method to the Player class, which we will use to respawn the player after they have died. We will implement this functionality when we build the game state structure around Gemstone Hunter:

```

public void Revive()
{
 PlayAnimation("idle");
 dead = false;
}

```

- 6.** Execute the Gemstone Hunter application and kill some zombies! If you get killed by the zombies, you will need to end the application and restart it to continue playing.

## What just happened?

When we determine that the player has collided with an enemy, we check the `y` coordinate of the player's `WorldCenter` vector against the `y` coordinate of the enemy's `WorldLocation` vector. `WorldCenter` points to the middle of the object's display area, while `WorldLocation` points to the upper left corner. If the player's center is above the top of the enemy, we grant the player a kill of the zombie, and call `player.Jump()` to boost them back up into the air. The enemy's "die" animation is played, and the enemy is marked as dead.

Otherwise, the player's `Kill()` method is called, resulting in the player's "die" animation being played, and any current horizontal movement negated. Because the `Player` class's `Update()` method checks to make sure the player is not dead before accepting any input, the player will now be unable to move and lie lifeless on the floor.

## Level transitions

We have a couple of different door tiles in our tile set that can be used to allow the player to exit the current map and enter a new one. In Chapter 8, we indicated that marking a square with a code in the format `T_M_X_Y` would indicate that the square contained a transition to map number `M` at location `(X, Y)`. We will implement this functionality by modifying the `Player` class.

### Time for action – supporting map transitions

1. Add a new helper method to the Helper Methods region of the `Player` class:

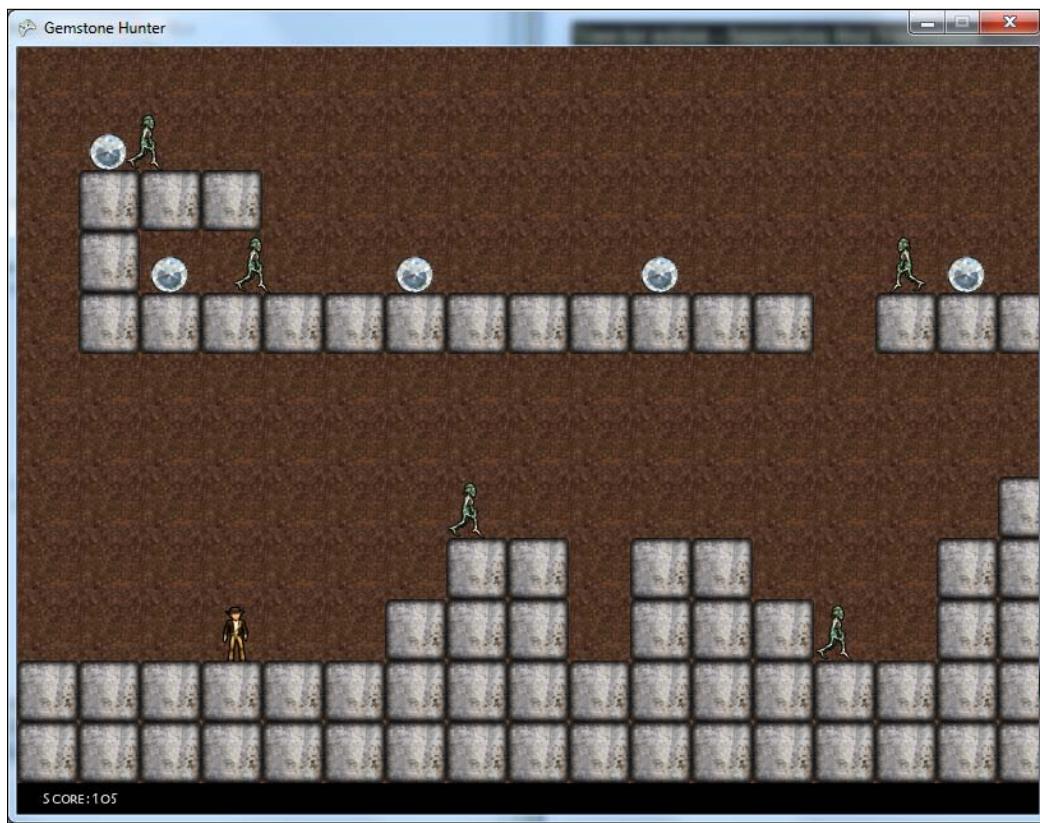
```
private void checkLevelTransition()
{
 Vector2 centerCell = TileMap.GetCellByPixel(WorldCenter);
 if (TileMap.CellCodeValue(centerCell).StartsWith("T_"))
 {
 string[] code = TileMap.CellCodeValue(centerCell).
 Split('_');

 if (code.Length != 4)
 return;
 LevelManager.LoadLevel(int.Parse(code[1]));
 WorldLocation = new Vector2(
 int.Parse(code[2]) * TileMap.TileWidth,
 int.Parse(code[3]) * TileMap.TileHeight);
 LevelManager.RespawnLocation = WorldLocation;
 velocity = Vector2.Zero;
 }
}
```

2. Modify the `Update()` method of the `Player` class to add a check for pressing the *Up* key or pressing *Up* on the gamepad to check the current square for an available transition. Place this after the check for pressing the *Space bar* or *A* button to jump:

```
if (keyState.IsKeyDown(Keys.Up) ||
 gamePad.ThumbSticks.Left.Y > 0.3f)
{
 checkLevelTransition();
}
```

3. Execute the `Gemstone Hunter` application and move right on the map until you reach the cave door. Stand on the door and press up on the keyboard or gamepad to transition to the next map. Try not to get killed by the zombies on the way! Once you reach the door and walk through it, the second map (`MAP001.MAP`), containing an underground level, will be loaded:



## What just happened?

The `checkLevelTransition()` method examines the code value in the cell containing the center of the player's sprite, looking for an entry starting with `T_` that indicates a transition square. If found, the text is split into pieces separated by an underscore using the `Split()` method. If the result is not four pieces (`T_`, the map number, an x coordinate, and a y Coordinate), the method simply returns and does nothing.

If the code is in the proper format, the `LevelManager` is asked to load the desired map. After the map has been loaded (which may contain `START` codes that position the player) the player is moved to the position indicated by the transition code value.

## Processing other codes

Now that we have implemented a method that allows the player to be killed, we can handle the final outstanding pre-defined code value—`DEAD`. This code makes entering the marked square lethal to the player. We will implement this code by building a helper method that will examine the current cell's code value and take appropriate action.

If you define additional code values, you can expand on this method to react to them appropriately.

## Time for action – handling codes

1. Add the `checkCurrentCellCode()` helper method to the `LevelManager` class:

```
#region Helper Methods
private static void checkCurrentCellCode()
{
 string code = TileMap.CellCodeValue(
 TileMap.GetCellByPixel(player.WorldCenter));

 if (code == "DEAD")
 {
 player.Kill();
 }
}
#endregion
```

2. Call `checkCurrentCellCode()` as the first task performed inside the `if (!player.Dead)` statement in the `Update()` method of the `LevelManager` class:  
`checkCurrentCellCode();`

3. Launch the game and run to the right until you reach an area where there are floor blocks missing, forming a pit. Jump into the pit, and your character will die when he reaches the lowest level of the pit.

### ***What just happened?***

We simply read the code value from the map for the cell where the player's `WorldCenter` point is located. This allows the player to get close to deadly things without actually dying, but one pixel too far will trigger the square.

## **Game structure**

Wrapping our game in a simple game state structure should be fairly familiar by now, and the process is straightforward. Let's put our game state structure together for the Gemstone Hunter game.

### **Time for action – implementing game states**

1. Add declarations to the `Game1` class for our `GameState` enum:
2. Still in the declarations section of the `Game1` class, add vectors for the display position of our text items, a texture to hold the title screen image, and the delay before respawn when the player dies:

```
Vector2 gameOverPosition = new Vector2(350, 300);
Vector2 livesPosition = new Vector2(600, 580);
```

```
Texture2D titleScreen;
```

```
float deathTimer = 0.0f;
float deathDelay = 5.0f;
```

3. We currently have temporary code in the `LoadContent()` method that loads straight into the first level of the game. Replace the current `LoadContent()` method with the following:

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

```
 Content.Load<Texture2D>(@"Textures\PlatformTiles"));
 TileMap.spriteFont =
 Content.Load<SpriteFont>(@"Fonts\Pericles8");

 pericles8 = Content.Load<SpriteFont>(@"Fonts\Pericles8");
 titleScreen = Content.Load<Texture2D>(@"Textures\TitleScreen");

 Camera.WorldRectangle = new Rectangle(0, 0, 160 * 48, 12 *
 48);
 Camera.Position = Vector2.Zero;
 Camera.ViewPortWidth = 800;
 Camera.ViewPortHeight = 600;

 player = new Player(Content);
 LevelManager.Initialize(Content, player);
}
```

**4.** Add the `StartNewGame()` helper method to the `Game1` class:

```
private void StartNewGame()
{
 player.Revive();
 player.LivesRemaining = 3;
 player.WorldLocation = Vector2.Zero;
 LevelManager.LoadLevel(0);
}
```

**5.** Replace the `Update()` method with the following to remove any temporary code and implement the game state logic:

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

 KeyboardState keyState = Keyboard.GetState();
 GamePadState gamepadState = GamePad.GetState(PlayerIndex.One);
 float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
 if (gameState == GameState.TitleScreen)
 {
 if (keyState.IsKeyDown(Keys.Space) ||
 gamepadState.Buttons.A == ButtonState.Pressed)
 {
 StartNewGame();
 }
 }
}
```

```
 gameState = GameState.Playing;
 }
}
if (gameState == GameState.Playing)
{
 player.Update(gameTime);
 LevelManager.Update(gameTime);
 if (player.Dead)
 {
 if (player.LivesRemaining > 0)
 {
 gameState = GameState.PlayerDead;
 deathTimer = 0.0f;
 }
 else
 {
 gameState = GameState.GameOver;
 deathTimer = 0.0f;
 }
 }
}
if (gameState == GameState.PlayerDead)
{
 player.Update(gameTime);
 LevelManager.Update(gameTime);
 deathTimer += elapsed;
 if (deathTimer > deathDelay)
 {
 player.WorldLocation = Vector2.Zero;
 LevelManager.ReloadLevel();
 player.Revive();
 gameState = GameState.Playing;
 }
}
if (gameState == GameState.GameOver)
{
 deathTimer += elapsed;
 if (deathTimer > deathDelay)
 {
 gameState = GameState.TitleScreen;
 }
}
base.Update(gameTime);
}
```



**6.** Replace the Draw() method with the following:

```
protected override void Draw(GameTime gameTime)
{
 GraphicsDevice.Clear(Color.Black);
 spriteBatch.Begin(
 SpriteSortMode.BackToFront,
 BlendState.AlphaBlend);
 if (gameState == GameState.TitleScreen)
 {
 spriteBatch.Draw(titleScreen, Vector2.Zero, Color.White);
 }
 if ((gameState == GameState.Playing) ||
 (gameState == GameState.PlayerDead) ||
 (gameState == GameState.GameOver))
 {
 TileMap.Draw(spriteBatch);
 player.Draw(spriteBatch);
 LevelManager.Draw(spriteBatch);
 spriteBatch.DrawString(
 pericles8,
 "Score: " + player.Score.ToString(),
 scorePosition,
 Color.White);
 spriteBatch.DrawString(
 pericles8,
 "Lives Remaining: " + player.LivesRemaining.ToString(),
 livesPosition,
 Color.White);
 }

 if (gameState == GameState.PlayerDead)
 {
 }
 if (gameState == GameState.GameOver)
 {
 spriteBatch.DrawString(
 pericles8,
 "G A M E O V E R !",
 gameOverPosition,
 Color.White);
 }
 spriteBatch.End();
 base.Draw(gameTime);
}
```

**7.** Execute the Gemstone Hunter game!

## ***What just happened?***

Much as we did in our previous games, we defined a limited set of game states, beginning the game in `TitleScreen` mode. The `Update()` and `Draw()` methods are segmented to take different actions based on the current `gameState` value.

Non-interactive game states (`PlayerDead` and `GameOver`) are controlled by our standard timing mechanism, allowing the game to advance after a preset period of time has elapsed (five seconds in this case).

## **Have a go hero**

Of all of the games presented in this book, *Gemstone Hunter* is perhaps the most open for customization and expansion. Here are just a handful of suggestions for implementing enhancements to *Gemstone Hunter* based on things you have learned building the games in this book:

- ◆ There are three other types of monster sprites included with the Platform Starter Kit. Modify the `Enemy` class to randomly select a monster type when a monster is spawned.
- ◆ Alternatively, modify the `Enemy` type to accept an additional parameter indicating which of the four types of monsters should be spawned, and create new codes in the Level Editor for each different type of monster. You can then specify exactly which type of monster you wish to appear in each location.
- ◆ As with *Robot Rampage*, *Gemstone Hunter* is currently silent. Add the sound system from *Asteroid Belt Assault* to *Gemstone Hunter* and generate appropriate sound effects for things like picking up gems and squishing zombies.
- ◆ Expand on the tile images provided with *Gemstone Hunter* to add items such as spiked pits and background details such as clouds, hills, and roots for underground areas.

## **Summary**

*Gemstone Hunter* implements the basics of a platform-style game, allowing the player to move around on a tile-based game world. Key concepts from the *Gemstone Hunter* game include:

- ◆ An alternative approach to sprite-based animation, using animation strips whose dimensions determine the number of frames in the animation instead of specifying individual frames from a larger sprite sheet
- ◆ Deriving multiple game object types from a base type and specializing the behavior of each of the child types while maintaining basic interaction with the game world from the base class

- ◆ Implementing gravity and tile map based collisions to allow the player to explore the game world
- ◆ Parsing the map codes we generated in the Level Editor to populate the game world with gemstones and enemies when a level is loaded
- ◆ Allowing the player to interact with hidden map codes for functionality such as level transitions and deadly areas

You've done it! With your first four XNA games completed, I hope that you now have not only ideas for the kinds of games you want to make, but also the knowledge to successfully implement them. The XNA Framework gives independent coders and small development teams a powerful set of tools to bring their creations to life, on the PC, the Xbox 360, and now the Windows Phone.

# Index

## Symbols

.NET CF 7  
.NET Compact Framework. *See* the .NET CF  
.NET Framework 7

## A

**AddComputerTerminal()** method 271  
**AddExplosion()** method 147, 226  
**AddLargeExplosion()** method 226  
**addNodeToOpenList()** method 252  
**AddShot()** method 229  
**AddSparkEffect()** method 226  
**AddWaypoint()** method 128  
**A\* implementation**  
    about 248  
    adjacent squares 256-259  
    beginning 251  
    path, finding 253  
    PathNode class, adding 248  
    requisites 244  
**alpha blending** 29  
**animated pieces**  
    about 58  
    classes 58  
    drawing 69-71  
    drawing, SpriteBatch overloads used 72  
    fading pieces 66  
    falling pieces 67  
    GameBoard, updating 63-66  
    managing 62  
    rotating pieces 68  
    updating 68

**AnimateWhenStopped** member 205

### Animation class

    building 350  
    constructor, adding 352  
    declaration, modifying 350  
    declarations, adding 350  
    properties, adding 351, 352  
    public methods, adding 352, 353  
    using directives, adding 350

**ArePiecesAnimating()** method 63

### Asteroid Belt Assault project

    about 9  
    asteroid field, creating 104  
    CollisionManager class 151  
    creating 89-91  
    EnemyManager class 126  
    explosion effects, implementing 141  
    game structure 161  
    Particle class, constructing 142  
    PlayerManager class 119  
    ShotManager class 116  
    sound effects 157  
    sprite-based star field, creating 100  
    Sprite class, building 91

### AsteroidManager class

    about 104  
    asteroids, colliding 111  
    asteroids, positioning 106  
    BounceAsteroids() method, adding 112  
    building 104  
    constructor, adding 105  
    declarations, adding 104  
    helper method, adding 104  
    second helper function, adding 105

## **asteroids**

- colliding 111
- drawing 110, 111
- position, checking 109, 110
- positioning 106-108
- updating 110, 111

## **B**

- baseAngle** vector 208
- BaseSprite** 205
- Begin()** pair 20
- BounceAsteroids()** method 112, 113
- BoundingBox** class 97
- BoundingBoxRect** property 97

## **C**

### **Camera** class

- adding, to title engine project 296
- creating 172-174
- declaration, modifying 296
- declarations, adding 296
- methods, adding 298
- properties, adding 297, 298
- using directive, adding 296

### **checkAsteroidToPlayerCollisions()** method 155

### **CheckChanged** event 337

### **CheckCollisions()** method 156

### **checkCurrentCellCode()** helper method 390

### **checkEnemyToPlayerCollisions()** method 154

### **checkLevelTransition()** method 390

### **checkPowerupPickups()** method 241

### **checkPowerupSpawns()** method 240

### **checkRocketSplashDamage()** method 285

### **CheckScoringChain()** method 66, 78, 85

### **CheckScoringRow()** method 53

### **checkShotEnemyImpacts()** method 285

### **checkShotToAsteroid()** method 153

### **checkShotToEnemy()** method 153

### **checkShotToPlayerCollision()** method 154

### **checkWeaponUpgradExpire()** method 234

### **Child** classes 58

### **CircleCollision()** method 154, 183

### **class** declaration area

- variables, adding 14

### **classes, for animated pieces**

- about 58
- creating 58

### **classes, for fading pieces**

- creating 61, 62

### **classes, for falling pieces**

- creating 60, 61

### **classes, for rotating pieces**

- creating 58, 59

### **ClearBoard()** helper method 41

### **ClearBoard()** member 49

### **Click** event handler 338

### **Code Folding** 174

### **collision** detection

- supporting 96, 97

### **collision** manager 151

### **CollisionManager** class

- about 151
- checkAsteroidToPlayerCollisions()** method, adding 155
- CheckCollisions()** method, adding 156
- checkEnemyToPlayerCollisions()** method, adding 154
- checkShotToAsteroid()** method, adding 153
- checkShotToEnemy()** method, adding 153
- checkShotToPlayerCollision()** method, adding 154
- constructor, adding 151
- creating 151
- declarations, adding 151

### **collisions**

- handling 152
- player collisions 154
- player shot collisions 153
- types 152

### **colors** 15

### **Common Language Runtime (CLR)** 7

### **ComputerTerminal** class

- adding, to robot rampage project 263
- constructor, adding 263
- declarations, adding 263
- public methods, adding 264
- using directives, adding 263

### **constants**

- vs. numeric literals 32

**Contains() method** 19

**Content Pipeline**

about 25, 26

textures, reading into memory 26

**createLargeExplosion() method** 237

**currentSquare** 15

## D

**Dequeue() method** 66

**Deserialize() method** 343

**Destination property** 183

**DetectShutdowns() method** 271, 285

**DetermineScore() method** 53

**Dictionary** 62

**difficulty levels**

adding, to game 84-86

**directives**

using 32

**DirectX API** 7

**DirectX Application Programming Interface (API).**

*See* **DirectX API**

**DirectX Software Development Kit (SDK)** 7

**Draw() method** 16, 19, 49, 50, 118, 119, 177, 182, 306, 307

## E

**EffectsManager class**

about 221

adding, to robot rampage project 221

declaration, modifying 221

declarations, adding 221

initialize() method, adding 221

RandomDirection() method, adding 222

updating 222

using directives, adding 221

**elastic collision**

bouncing asteroids 112-115

implementing 111

**End() pair** 20

**enemy AI methods** 275

**Enemy class**

about 126

AddWaypoint() method, adding 128

constructor, adding 127

creating 127

declarations, adding 127

Draw() method, adding 131

building 273

constructor, adding 273

declarations, adding 273

movement related methods, adding 275, 276

updating 274

using directive, adding 273

IsActive() method, adding 129

Update() method, adding 130

waypoint management 128

WaypointReached() method, adding 129

**EnemyManager class**

about 132

creating 132

declarations, adding 132

drawing 138

AddEnemy() method, adding 278

adding, to robot rampage project 277

declarations, adding 277

Initialize() method, adding 277

updating 278, 279

setting up 134

updating 137, 138

**enemy robots, robot rampage project**

adding 272

enemies, moving 275

enemy basics 272

EnemyManager class 277

WeaponManager class, updating 281, 282

**Enqueue() method** 66

**exitToolStripMenuItem\_Click() event handler** 320

**explosion effects, Asteroid Belt Assault project**

about 141

particle-based explosions, building 145

particle system, designing 142

**ExplosionManager class**

about 145

AddExplosion() method, adding 147

constructor, adding 146

declarations, adding 146

drawing 150

explosions, creating 147

RandomDirection() helper method, adding 147  
updating 149

## F

**FadingPiece class** 62

**fading pieces**

adding 67  
generating 66

**FallingPiece class** 61

**falling pieces**

about 67  
generating 67

**FillFromAbove() method** 67

**FillPiece() method** 45

**findAdjacentNodes() method** 256

**FindPath() method** 253, 256

**FireShot() helper method** 121

**FireShot() method** 118

**FireWeapon() method** 230

**FixScrollBarScales() helper method** 326

**flood**

displaying 82-84  
tracking 81, 82

**Flood Control game**

about 8  
animated pieces, adding 58  
building 47  
classes 29  
Content Pipeline 25  
difficulty levels, adding 84-86  
flood, tracking 81  
GameOver game state, adding 79, 80  
GamePiece class 30  
project, setting up 24, 25  
SpriteFonts, adding 73  
water filled pipes 44  
water, propogating 45

**Flood Control game, building**

Draw() method 49, 50  
Game1 declarations 47, 48  
Initialize() method, updating 48  
mouse input, handling 53, 54  
player, letting to play 55, 56  
playing 56  
scores 52, 53

scoring chains 52, 53  
screen, drawing 49-52

**FormClosed event**

handing 346

**frame-based animations**

about 98  
animation frames, adding 98  
Sprite, drawing 99, 100  
Sprite, updating 98, 99

## G

**game**

building 12  
Windows game project, creating 12, 13

**Game1 class**

checkPlayerDeath() method, adding 286  
updating 285-289  
resetGame() helper function, adding 163

**Game1 class, map editor project**

updating 331, 332

**Game1 Update method** 333

**GameBoard class**

about 39  
connection, making 45-47  
game board, creating 40  
GameBoard.cs class, creating 39, 40  
game board, initializing 40, 41  
game board, manipulating 41, 42  
GamePieces, updating 41  
gaps, filling in 42, 43  
new pieces, generating 43, 44  
water filled pipes 44, 45  
water, propogating 45

**gameBoardDisplayOrigin vector** 48

**gameForm\_VisibleChanged() method** 319

**GameManager class**

adding, to robot rampage project 283  
declaration, modifying 283  
declarations, adding 283  
public methods, adding 284

**GameObject class**

about 354  
building 354  
declaration, modifying 355  
declarations, adding 355

- map-based collision detection methods, adding 361, 362
- properties, adding 355, 356
- public methods, adding 358
- updateAnimation() helper method, adding 357
- using directives, adding 354
- GameOver game state**
  - adding 79, 80
- GamePiece class**
  - about 30, 58
  - building 30-32
  - connection, methods 36
  - constructors, building 33
  - creating 33
  - GetSourceRect 37, 38
  - identifying 30
  - methods, rotating 35, 36
  - methods, updating 33
  - pipe connectors 36
  - updating 33
- games**
  - Asteroid Belt Assault 9
  - Flood Control 8
  - Gemstone Hunter 9
  - overview 8
  - requisites 9
  - requisites, optional 10
  - Robot Rampage 9
- game structure, Asteroid Belt Assault project**
  - about 161-164
  - drawing 166, 167
- game structure, Gemstone Hunter project**
  - about 391
  - game states, implementing 391-394
- game structure, robot rampage project**
  - about 283
  - Game1, updating 285
  - score keeping 285
- Gemstone class**
  - building 377
  - constructor, adding 378
  - declaration, modifying 377
  - using directives, adding 377
- Gemstone Hunter project**
  - about 9, 291
- advanced tile engine, constructing 294
- animated game objects 354
- animation 357
- animation strips 349
- camera, rebuilding 296
- drawing 357
- enemies, creating 382
- game structure 391
- gemstones, building 377
- graphics, borrowing 292
- levels, loading 372
- level transitions 388, 389
- map codes, handling 390
- map editor project 310
- movement 357
- other codes, processing 390
- player, creating 364
- player-enemy interaction 386
- projects, creating 292
- property, adding 380
- score tracking, implementing 380
- tile engine project, constructing 294, 299
- tile map, drawing 306
- zombies, summoning up 382-384
- GenerateNewPieces() method 44, 67**
- GenerateRandomMap() method 199, 200**
- GetSourceRect() method 37, 45**
- GetWaterChain() method 46**
- GoalManager class**
  - adding, to robot rampage project 266
  - declaration, modifying 266
  - declarations, adding 266
  - DetectShutdowns () method, adding 285
  - Initialize() method, adding 266
  - public methods, adding 269-271
  - read-only property, adding 266
  - updating 270
  - using directives, adding 266
- graphics object 14**
- graphics\_PreparingDeviceSettings() event handler 315**

## H

- HandleGamepadInput() method 126**



**handleInput()** method 209, 211, 214, 230  
**HandleKeyboardInput()** helper method 122  
**HandleKeyboardInput()** method 126  
**Handle...()** methods 209  
**HandleMouseInput()** helper method 54  
**HasConnector()** method 53

## I

**imposeMovementLimits()** helper method 123  
**imposeMovementLimits()** method 125, 126  
**initialize()** method  
    about 16  
    customizing 16  
**Initialize()** method 158, 190, 365  
**installing**  
    XNA Game Studio 11, 12  
**Intersects()** method 97  
**IsBoxColliding()** method 97, 109  
**IsCircleColliding()** method 98  
**IsEqualToNode()** method 251  
**IsKeyDown()** method 123  
**IsMouseVisible** property 16  
**isOnScreen()** helper method 109  
**IsWallTileByPixel()** 195

## L

**Lerping** 145  
**LevelManager** class  
    building 372, 373  
    declaration, modifying 374  
    declarations, adding 374  
    Initialize() method, adding 374  
    LoadLevel() method, adding 375  
    properties, adding 374  
    ReloadLevel() method, adding 375  
    using directives, adding 373  
**level transitions, Gemstone Hunter project**  
    about 388  
    map transitions, supporting 388, 389  
**LinearCost()** method 251  
**LoadContent()** method 16, 74, 156, 170, 174, 227, 231  
**LoadImageList()** helper method 325  
**Load()** method 27, 74

## M

**map**  
    drawing 196  
    tile map, drawing 196, 197  
**map-based collision detection** 360  
**map codes** 345  
**MapEditor** form  
    connecting, to game 336  
    editor, completing 336-338  
    final controls 327-330  
    menu bar, creating 320  
    scroll bars 325, 326  
    scrolling delay, fixing 340  
    tile selection controls 321, 323  
**MapEditor\_FormClosed()** event 346  
**MapEditor\_Load()** event handler 324  
**MapEditor\_Load()** method 329  
**map editor project**  
    about 310  
    controls, adding to form 319  
    creating 310  
    event handlers, adding 318, 319  
    form, adding 313-315  
    Game1 class, updating 331, 332  
    Game1 Update method 333, 334  
    level editor project, creating 311-313  
    map codes 345  
    maps, loading 341, 342  
    maps, saving 341, 342  
    passability 344  
**MapEditor\_Resize()** method 327  
**MapSquare** class  
    adding, to Gemstone Hunter project 294  
    declaration, modifying 294  
    declarations, adding 294  
    TogglePassable() method, adding 295  
    using directives, adding 294  
**map squares**  
    dealing with 191-193  
**mapSquares** array 191  
**MathHelper** 19  
**MathHelper.Clamp()** method 175, 211  
**MathHelper.Max()** method 19  
**MouseState** class 54  
**Move()** method 175

## N

**Next() method** 19

**numeric literals**

vs constants 32

## O

**ObjectIsVisible() method** 176

## P

**particle-based explosions**

building 145

**Particle class**

constructing 142

constructor, adding 143

declarations, adding 142

drawing 144

constructor, adding 219

creating 218, 221-241

declarations, adding 218

properties, adding 218

updating 219, 220

using directives, adding 218

updating 144

**particle system, explosion effects**

designing 142

**passability** 344

**PathFinder class**

adding, to robot rampage project 251

addNodeToOpenList() method, adding 252

class declaration, modifying 251

declarations, adding 251

findAdjacentNodes() method, adding 256-259

FindPath() method, adding 253-255

using directive, adding 251

**path finding system implementation, robot  
rampage project**

about 244

A\* implementation 248

A\* path finding algorithm 244-248

power-ups, placing safely 261

**PathNode class**

adding, to robot rampage project 248

constructor, adding 249

declarations, adding 248

helper method, adding 250

properties, adding 249

using directive, adding 248

**pathWaypoints object** 133

**pieceType parameter** 61

**PlayAnimation() method** 358

**PlayEnemyShot() method** 159

**player**

adding 200

in bounds, staying 210, 211

input, handling 206-208

Player class, building 201-205

walls, accounting for 212-215

**Player class**

constructor, creating 364, 365

creating 364

Dead property, adding 364

declaration, modifying 364

declarations, adding 364

jumping animation 370

repositionCamera() method 371

running animation 367

update() method, overriding 367, 368

using directives, adding 364

**player-enemy interaction, Gemstone Hunter  
project**

about 386

zombies, interacting with 386, 387

**player goals implementation, robot rampage  
project**

about 262

computer terminals, building 263-265

computer terminals, spawning 265

**PlayerManager class**

constructor, adding 120

creating 119

declarations, adding 119

Draw() method, adding 124

FireShot() helper method, adding 121

HandleGamepadInput() helper method,  
adding 122

HandleKeyboardInput() helper method,  
adding 122

imposeMovementLimits() helper method,  
adding 123

- Update() method, adding 124
- user input, handling 121, 122
- playerScore 15**
- player shot collisions 153**
- player's ship**
  - drawing 123
  - updating 123
- PlayExplosion() method 159**
- PlayPlayerShot() method 159**
- position vector 175**
- power ups, robot rampage project**
  - about 238
  - spawning 239
- PreparingDeviceSettings event 317**
- PropagateWater() method 45, 47**
- public methods, GameObject class**
  - Draw() method, adding 359
  - PlayAnimation() method, adding 358
  - Update() method, adding 358
- puzzle game**
  - designing 24
  - Flood Control project, setting up 24, 25

## R

- rand 14**
- RandomDirection() helper method 147**
- random map**
  - generating 198
  - random wall placement 198, 200
- Random.Next() method 42**
- RandomPiece () method 42**
- randomVelocity() method 109**
- redistributable fonts package 12**
- RemoveSuffix() method 34**
- Replace() method 34**
- repositionCamera() 212**
- repositionCamera() helper method 210**
- resetGame() helper function 163**
- resetGame() method 165**
- ResetWater() method 45**
- robot rampage project**
  - about 9
  - creating 170
  - A\*, implementing 244

- enemy robots, adding 272
- game structure 283
- path finding system, implementing 244
- player goals, implementing 262
- visual effects 217
- weaponry, adding 228
- RotatePiece() method 35, 36**
- RotatingPiece class**
  - about 58
  - modifying 59
- rotating pieces**
  - about 68
  - generating 68
- rotationAmount property 60**
- rowLookup variable 43**
- running animation, Player class**
  - running animation, Player class about 367

## S

- score display 75**
- ScoreZoom class**
  - about 76
  - creating 76
  - displaying 77-79
  - updating 77, 78
- ScreenToWorld() method 299**
- SelectedIndexChanged event 336**
- Serialize() method 343**
- SetTileAtCell() method 305**
- SetTileAtSquare() 195**
- setUpWaypoints() method 135**
- ShotManager class**
  - about 117
  - adding 116
  - constructor, adding 116
  - declarations, adding 116
  - Draw() method, adding 118
  - FireShot() method, adding 117
  - Update() method, adding 118
- shots**
  - firing 117
- sound effects, Asteroid Belt Assault project**
  - about 157
  - finding 157

- generating 157
- sound effects manager**
  - building 158, 159
- SoundManager class**
  - about 157
  - declaration, modifying 158
  - implementing 160
  - Initialize() method, adding 158
  - PlayEnemyShot() method, adding 159
  - PlayExplosion() method, adding 159
  - PlayPlayerShot() method, adding 159
- SpawnEnemy() method 135**
- sprite-based star field 100**
- SpriteBatch.Draw() call 27**
- SpriteBatch.Draw() method 20, 144, 176**
- SpriteBatch.Draw() overload 72**
- SpriteBatch.DrawString() method 75**
- spriteBatch.End() 20**
- SpriteBatch overloads**
  - used, for drawing animated pieces 72
- Sprite class**
  - new Sprite class, building 177-182
- Sprite class, Asteroid Belt Assault project**
  - about 91
  - animation and drawing properties,
    - adding 94, 95
  - basic properties, adding 93, 94
  - building 91
  - collision detection, supporting 96, 97
  - constructor, adding 93
  - declarations 92
  - frame-based animation 98
  - Frame property 95
  - FrameTime property 95
  - Location property 94
  - Source property 95
  - TintColor property 94
  - Velocity property 94
- SpriteFonts**
  - adding, to game 73, 74
  - score display 75
  - ScoreZoom 76
- sprites**
  - and sprite sheets 27-29
  - vs textures 27

- sprite sheets**
  - and sprites 27-29
- SquareChase!**
  - drawing 20
  - playing 20
- SquareChase mini game, variables**
  - colors 15
  - currentSquare 15
  - playerScore 15
  - rand 14
  - squareTexture 14
  - TimePerSquare 15
  - timeRemaining 15
- SquareScreenRectangle() method 193**
- squareTexture**
  - about 14
  - creating 17
- SquareWorldRectangle() method 193**
- StarField class**
  - about 100
  - constructor, adding 101
  - creating 100, 101
  - declarations, adding 101
  - drawing 102
  - implementing 102
  - updating 102
- StarField.Draw() method 102**
- StartNewGame () method 284**
- StartNewLevel() method 85**

## T

- TargetElapsedTime property 17**
- TextChanged event handler 337**
- Texture2D object 20**
- this.Window.ClientBounds property 19**
- tile engine project**
  - constructing 294
  - MapSquare class, adding 294
- tile map**
  - adding, to game project 308, 309
  - drawing 306
- TileMap class**
  - declaration, modifying 299
  - declarations, adding 299

- Initialize() method, adding 300
- methods, adding 302-304
- using directives, adding 299
- Tile\_Sheet.png file 29**
- TimePerSquare 15**
- timeRemaining 15**
- timeRemaining counter 19**
- timerGameUpdate\_Tick() handler 341**
- titles**
  - about 188
  - dealing with 194
  - handling 194, 195
  - TileMap class, creating 189-191
  - title map 188
- Transform() method 176**
- tryToSpawnPowerup() method 239**
- turretAngle vector 208**

## U

- Update() 19**
- UpdateAnimatedPieces() method**
  - about 65
  - calling 68
- updateAnimation() helper method 357**
- UpdateFadingPieces() method 64**
- UpdateFallingPieces() method 64**
- Update(). Max() 19**
- Update() method 48, 182**
  - about 17, 118
  - coding, for SquareChase 18
- UpdatePiece() method 60**
- UpdateRotatingPieces() method 64**
- updateWaveSpawns() method 136, 137**
- user input, handling 121**
- using directives 177, 201**

## V

- verticalOffset parameter 61**
- view**
  - Camera classes, viewing 184, 186
  - sprite, viewing 184, 186
  - visualizing 184
- viewPortSize vector 175**

## visual effects, robot rampage project

- about 217
- EffectsManager class, adding 221
- explosions, building 223, 224
- particle class, adding 218-241
- particles, revisiting 218
- spark effects 226

## W

- wallChancePerSquare 200**
- waterHeight variable 84**
- WaterTracker list 47**
- waveSpawns dictionary 135**
- WaypointReached() method 132**
- waypoints, managing**
  - enemies, spawning 135, 136
  - EnemyManager class, setting up 134, 135
- WeaponManager class**
  - about 228, 233
  - AddShot() method, adding 229
  - beginning 228
  - checkPowerupPickups() method, adding 241
  - checkPowerupSpawns() method, adding 240
  - checkRocketSplashDamage() method,
    - adding 281, 282, 285
  - checkShotEnemyImpacts() method,
    - adding 281, 282, 285
  - checkShotWallImpacts() method,
    - adding 236, 237
  - checkWeaponUpgradExpire() method,
    - adding 234
  - createLargeExplosion() method, adding 237
  - declaration, modifying 228
  - declarations, adding 228, 239
  - FireWeapon() method, adding 230
  - FireWeapon() method, replacing 233
  - properties, adding 228
  - tryToSpawnPowerup() method,
    - adding 239, 240
  - Update() method, modifying 237
  - updating 230, 261, 281
  - using directives, adding 228
  - WeaponFireDelay property, replacing 233
- weapon upgrades, robot rampage project**
  - about 232

- power-ups 238
- shot to map collisions 236

#### **Windows game project**

- creating 12, 13

**WorldToScreen() method 299**

## **X**

### **XNA**

- about 8
- sound, implementing 157

#### **XNA game**

- about 13
- declarations area 14
- Draw() method 19
- Game1 class constructor 15

- initialize() method 16

- initialize() method, customizing 16

- LoadContent() method 16

- SquareChase!, drawing 20

- SquareChase!, playing 20, 21

- squareTexture, creating 17

- update(), coding for SquareChase 18

- update() method 17, 18

- variables, adding to class declaration area 14

#### **XNA Game Studio**

- installing 11, 12

- requisites 9

- requisites, optional 10

#### **XNA project templates 8**

**XNA's Not Acronymed.** *See* **XNA**





**Thank you for buying  
XNA 4.0 Game Development by Example:  
Beginner's Guide**

## About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

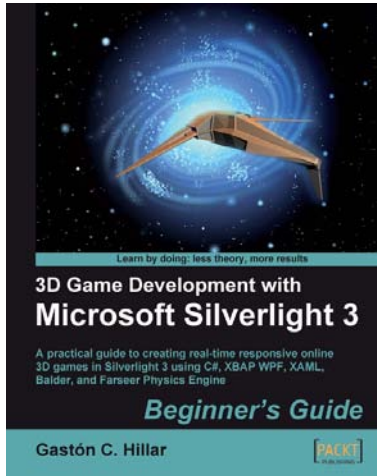
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.PacktPub.com](http://www.PacktPub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



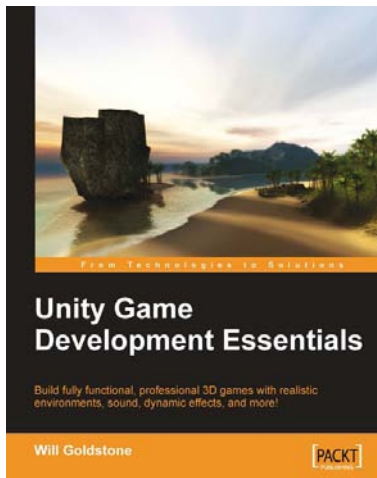


## 3D Game Development with Microsoft Silverlight 3: Beginner's Guide

ISBN: 978-1-847198-92-1      Paperback: 452 pages

A practical guide to creating real-time responsive online 3D games in Silverlight 3 using C#, XBAP WPF, XAML, Balder, and Farseer Physics Engine

1. Develop online interactive 3D games and scenes in Microsoft Silverlight 3 and XBAP WPF
2. Integrate Balder 3D engine 1.0, Farseer Physics Engine 2.1, and advanced object-oriented techniques to simplify the game development process
3. Enhance development with animated 3D characters, sounds, music, physics, stages, gauges, and backgrounds



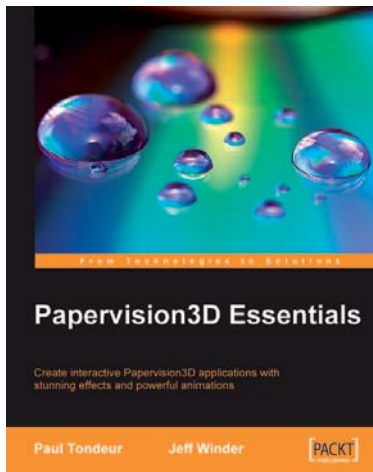
## Unity Game Development Essentials

ISBN: 978-1-847198-18-1      Paperback: 316 pages

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

1. Kick start game development, and build ready-to-play 3D games with ease
2. Understand key concepts in game design including scripting, physics, instantiation, particle effects, and more
3. Test & optimize your game to perfection with essential tips-and-tricks
4. Written in clear, plain English, this book is packed with working examples and innovative ideas
5. This book is based on Unity version 2.5 and uses JavaScript for scripting

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

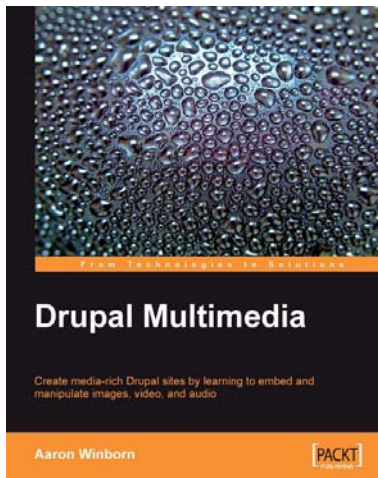


## Papervision3D Essentials

ISBN: 978-1-847195-72-2      Paperback: 428 pages

Create interactive Papervision 3D applications with stunning effects and powerful animations

1. Build stunning, interactive Papervision3D applications from scratch
2. Export and import 3D models from Autodesk 3ds Max, SketchUp and Blender to Papervision3D
3. In-depth coverage of important 3D concepts with demo applications, screenshots and example code.
4. Step-by-step guide for beginners and professionals with tips and tricks based on the authors' practical experience



## Drupal Multimedia

ISBN: 978-1-847194-60-2      Paperback: 264 pages

Create media-rich Drupal sites by learning to embed and manipulate images, video, and audio

1. Learn to integrate multimedia in your Drupal websites
2. Find your way round contributed modules for adding media to Drupal sites
3. Tackle media problems from all points of views: content editors, administrators, and developers

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles