CLASSIC GAME DESIGN

Second Edition

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher*. Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

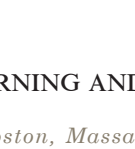
Companion files on the disc are also available by writing to the publisher at info@merclearning.com.

CLASSIC GAME DESIGN

From Pong to Pac-Man with Unity

Second Edition

Franz Lanzinger



MERCURY LEARNING AND INFORMATION

Dulles, Virginia | Boston, Massachusetts | New Delhi

Publisher: David Pallai

Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

This book is printed on acid-free paper.

Franz Lanzinger. *Classic Game Design. From Pong to Pac-Man with Unity, Second Edition.*

ISBN: 978-1-68392-385-5

Library of Congress Control Number: 2019939379

192021 321

Printed on acid-free paper in the United States of America.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

Who Are You? 1
What are Classic Arcade Video Games? 2
Unity, Blender, Gimp, and Audacity 4
How to Use This Book 6

CHAPTER 2: TOOLS OF THE TRADE 8

Installing Unity 8
Hello World! 10
Programming with C# 17
Using GIMP to Make an Image 22
Using Blender to Make a 3D Object 25
Using Audacity to Make a Sound Effect 28
Using Unity: My First Demo 31

CHAPTER 3: *PONG* 40

Before *Pong* 40
Pong, Atari (1972) 42
Coin-op, the Real Atari 44
Pong Sequels and Clones 44
Bitmasters, Day One 44
Pong at Forty 45

CHAPTER 4: CLASSIC PADDLE GAME 46

Getting Ready 46
Version 0.01: The Playfield 46
Version 0.02: The Paddles 52
Version 0.03: The Ball 57
Version 0.04: A Better Playfield 59
Version 0.05: Audio 61
Version 0.06: Scoring 64
Version 1.0: First Release! 67

Postmortem 68
Exercises 69

CHAPTER 5: *BREAKOUT* 71

Woz 71
Breakout, Atari (1976) 71
Breakout Sequels 74
Where Are They Now? 75

CHAPTER 6: CLASSIC BRICK GAME 76

Paddle Game for One 76
Version 0.01: The Playfield 77
Version 0.02: The Player 80
Version 0.03: Basic Ball Movement 83
Version 0.04: Collisions 86
Version 0.05: Bricks 91
Version 0.06: First Playable 95
Version 0.07: Scoring 96
Version 0.08: Title Screen 99
Version 1.0: First Release and Postmortem 102
Exercises 103

CHAPTER 7: *SPACE INVADERS* 104

Huge Money, Huge 104
The Design of *Space Invaders,* Taito (1978) 104
Score Equals Skill 105
Going Strong 41 Years Later 107

CHAPTER 8: CLASSIC GAME PROJECT THREE: VERTICAL SHOOTER 108

Designing a Shooter 108
Version 0.01: The Playfield 108
Version 0.02: The Spaceship 112
Version 0.03: Sprites 117
Version 0.04: Aliens 126
Version 0.05: Alien Shots 133

Version 0.06: Scoring and Lives 138
Version 0.07: Alien Death Sequence 149
Version 0.08: Sound 153
Version 0.09: Levels 156
Version 1.0: Release and Postmortem 162
Exercises 163

CHAPTER 9: *SCRAMBLE* 165

Scrolling Shooter 165
Experts Rule 166
Scramble Sequels 167

CHAPTER 10: CLASSIC GAME PROJECT FOUR: SCROLLING SHOOTER 169

Designing a Scrolling Shooter 169
Version 0.01: The Playfield 171
Version 0.02: Spaceship Part 1: Modeling 179
Version 0.03: Spaceship Part 2: Texturing 186
Version 0.04: Spaceship Control 191
Version 0.05: Level 1 196
Version 0.06: Rockets 200
Version 0.07: Flying Rockets 204
Version 0.08: Shots 212
Version 0.09: Flying Saucers 217
Version 0.10: Level Design 223
Version 0.11: Audio 229
Version 0.12: Scoring 231
Version 1.00: Release and Postmortem 233
Exercises 234

CHAPTER 11: *PAC-MAN* 236

The First Maze Game 236
Cutscenes 237
Pac-Man Fever 238
Ending Rule 238
Pac-Man AI 239
Pac-Man Sequels and Maze Games 240

CHAPTER 12: CLASSIC GAME PROJECT FIVE: MAZE GAME 244
Designing a Maze Game 244
Version 0.01: The Maze 245
Version 0.02: The Player 251
Version 0.03: Nasty Enemies 254
Version 0.04: Dots 258
Version 0.05: Audio 259
Version 0.06: Scoring and Levels 262
Version 0.07: Tuning 268
Version 1.00: Release and Postmortem 270
Exercises 271

EPILOGUE 272
So Many Games, So Few Pages 272
Novelty 273
How Modern Games are Influenced by the Classics 273

APPENDIX I: INTRODUCTION TO C# FOR BEGINNERS 276
Programming Is Easy 276
Interpreted or Compiled? 277
Numbers and Strings 278
Variables and Variable Names 279
Whitespace 281
Statements and Semicolons 281
Computations 282
Functions and Function Calls 282
Looping 283
Learning to Code 284
The Code in this Book 284

APPENDIX II: EIGHT RULES OF CLASSIC GAME DESIGN 286

APPENDIX III: ABOUT THE DVD 287

INDEX 288

Acknowledgments

I'd like to thank just some of the many people who made this book possible.

Most significantly, Eric Ginner and Robert Jenks read and worked through large portions of the book in draft form and gave voluminous feedback and suggestions. This book would be less without their valuable contributions.

Special thanks to Mark Robichek, Karl Anderson, Mark Alpiger, Sam Mehta, Brian McGhie, Desiree McCrorey, Joe Cain, Eugene Polonsky, Bob Jones, Aaron Hightower, Ed Logg, Dave O'Riva, Steve and Susan Woita, and everyone at Atari coin-op.

David Pallai, my fantastic publisher, helped every step of the way and is always kind and supportive.

A big word of thanks to my parents, Klaus and Aida Lanzinger, for their humanity and love.

And last, but not least, a giant hug and thank you to my wife, Susan Lanzinger, for helping throughout the years. I couldn't have done this without you.

About the Author

Franz Lanzinger is the owner of Lanzinger Studio, an independent game development and music studio in Sunnyvale, California. He began his career in game programming in 1982 at Atari Games Inc., where he designed and programmed the classic arcade game *Crystal Castles*. In 1989 he joined Tengen, where he was a programmer and designer for *Ms. Pac-Man* and *Toobin'* on the NES. Mr. Lanzinger co-founded Bitmasters, where he designed and coded games including *Rampart* and *Championship Pool* for the NES and SNES, and *NCAA Final Four Basketball* for the SNES and Sega Genesis. In 1996 Mr. Lanzinger founded Actual Entertainment, publisher and developer of the *Gubble* series for PC and IOS. Mr. Lanzinger has a B.Sc. in mathematics from the University of Notre Dame and attended graduate school at the University of California at Berkeley. In 1980 he started playing arcade games, and at one time held the arcade world record scores on *Centipede* and *Burgertime*. Franz Lanzinger is a professional accompanist, piano teacher, and avid golfer. He continues to design and code games.

CHAPTER 1

Introduction

IN THIS CHAPTER

This is a hands-on book about game design, and what better way to learn about game design than to study and emulate the classics. You're going to make some games, not just read about them. Assimilating the classics is a time-honored tradition. Pianists play Bach, writers read Shakespeare, and painters copy the *Mona Lisa*. It's not just about experiencing them; it's about creating something very similar and grasping the process of creation that brings the most benefit.

WHO ARE YOU?

This book is for everyone who loves to play and make games, preferably in that order. You should be somewhat computer literate, but it's OK if you've never written a line of code, never taken an art class, and you're tone deaf.

Maybe you're a student concentrating your studies on programming, art, or design. This book can teach you about the basics of classic game design and introduce you to the major classic games and design techniques that every game developer should know. It's these classics that led the way and showed future generations of designers how to make games.

Maybe you're a fan of the old games, the ones that started it all. It's just plain fun and relatively easy to recreate the old games using modern tools. Maybe you own a few classic arcade games and you like to change the option switches to see what happens. This book will allow you to do a lot more than that. Not only are the sample games fun by themselves, you'll learn how to make changes to them without being a

professional programmer. With a little bit of effort you'll learn the basics of how to add new features, change the way the scoring works, and replace the graphics and sounds with something entirely different.

Possibly you're just interested in learning Unity, Blender, GIMP, or Audacity. These development tools are the basis of this hands-on approach and enable you to get started making your own games right away. These tools are free to use. GIMP, Audacity, and Blender are open-source software, which means that you can use them any way you wish, no strings attached. As of 2019, Unity is available in three versions, two of which cost money. In this book you will be using the free version. See the Unity website for details. These four tools are extremely powerful. By using Unity, Blender, GIMP, and Audacity to make a few games you'll gain a good introductory understanding of the entire process. You'll then be able to more easily tackle a myriad of advanced topics in game design and development. Installation instructions are available later in this chapter.

Just as composers need to listen to music, artists should look at art, and writers had better read, so game designers ought to play games, especially their own games. There's a word for it, "dogfooding," which literally means that if you're making dog food, you need to eat it too. In the classic era of the '70s and '80s, it was possible to keep up with the industry and play all the top games. Nowadays you have to pick and choose, but that's no excuse for not playing at all. Whether you're a newbie designer or a forty-year veteran with dozens of credited titles, you need to also be a player.

WHAT ARE CLASSIC ARCADE VIDEO GAMES?

Arcade games are coin-operated machines, where players pay money in the form of coins or tokens to play a game. They are sometimes called *coin-op games* for short. The early arcade games were built and designed to be played in arcades and street locations such as restaurants, movie theaters, or airports. Arcade video games work as a business because they provide a game experience that's hard to duplicate at home. Ever since home console and PC video games became hugely popular in the late '80s arcade video games have been relegated to the few remaining arcades and

street locations. Thirty years later there has been renewed and often nostalgic interest in arcade games by collectors and hobbyists, but the days of manufacturing tens of thousands of arcade cabinets for the latest arcade hit are over.

From a game design perspective, arcade video games are no different from the console games of today, for the most part. Some coin-specific features such as "add-a-coin" or dealing with a ticket dispenser only apply to arcade games, but the basics of controlling a character on a rectangular screen haven't really changed since 1972 and apply to arcade games, computer games, console games, and even mobile games.

The heyday of these types of games started in 1972 and ended in about 1984 when the arcade game industry collapsed in the United States. New arcade games are still manufactured today, but in much lower numbers than in the '80s.

What is meant by a *classic*? A classic should be of high quality, timeless, and influential. These are somewhat subjective criteria, but they'll have to do. Amazingly, a very large proportion of the top-selling arcade video games from the heyday of arcade video games fit this description. This almost seemed inevitable. There weren't that many arcade games made when compared to the huge number of new games released in the following decades. In the '80s this art form was so new and resulted in such huge growth that almost any reasonable idea would get reused countless times in the coming years. It was much easier to create an influential game back then compared to the present day. The high quality and timeless aspects were more difficult to achieve, especially because the technology was new and often cumbersome. Still, the '70s and early '80s were nothing less than the golden age of video game design.

In this book you will take a detailed look at five classic arcade video games: *Pong, Breakout, Space Invaders®, Scramble™,* and *Pacman™*. In the interest of learning the basics well rather than doing a comprehensive survey, the scope of this book is limited to those five featured games. There are probably several dozen other classic arcade video games that are similarly influential and important, games such as *Asteroids, Missile Command, Galaxian, Defender, Joust, Frogger®,* and *Pole Position,* just to

name a few. It is left up to you to look at, play, and learn from the many other classic arcade video games out there.

Each of the featured games is responsible for countless imitators. They pioneered some of the most important game categories. The games you will be creating come from five categories: a paddle game, a brick game, a vertical shooter, a scrolling shooter, and a maze game. Furthermore, you'll be looking at some of the methods and design decisions that go into making these types of games.

This is a book about design, so it won't dwell too much on the now outdated technologies used to make the original featured games. Rather, it'll try to answer some very basic design questions that every game designer needs to tackle: What do the players do? What's their motivation? What are the strategies and tactics? What are the basic design elements?

The influence of classic arcade video games on modern games is undeniable. When you see a player getting points for running into something, or dying when colliding with something, it's because some arcade video game in the '70s or '80s pioneered it. Much of the history of these early days is lost, so it's difficult to give proper credit to the people and companies who are responsible.

The real fun comes when you try to reconstruct some of the game elements from the classics. The paddles in *Pong*, the bombs in *Scramble*, the shots in *Space Invaders*: these are basic game elements that every video game designer needs to understand. There's no better way to gain this understanding than by building some simple games that use these elements.

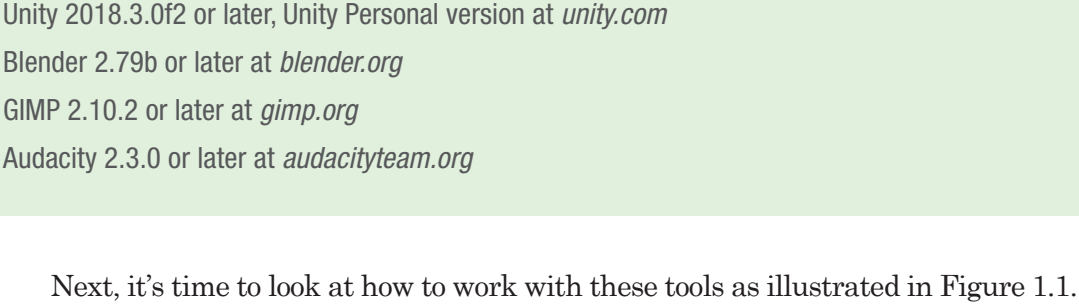
In the next section, you'll be taking a closer look at the tools you'll be using to make your own classic games.

UNITY, BLENDER, GIMP, AND AUDACITY

Yes, you'll dive right in and use professional tools to make your games. Here are free to use, modern, professional game development tools: Blender for 3D graphics, GIMP for 2D graphics, Audacity for sound, and Unity for creating the logic for the

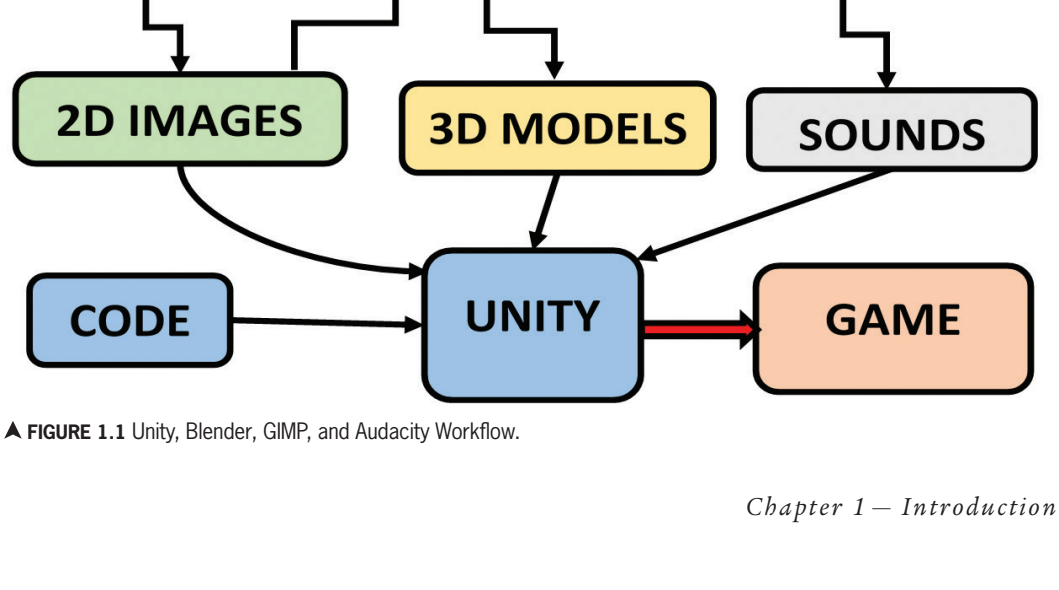
games and putting it all together. These are some of the same tools that many professional game developers use when making commercial games. Just a few years ago it would have cost many thousands of dollars to get access to game development tools of this caliber. Via the generous efforts of these open-source projects and the free version of Unity, even the smallest of budgets is sufficient to make a good-looking, good-sounding, and high-quality video game.

In the next chapter, you'll install the following software on your Mac or Windows PC.
Unity 2018.3.0f2 or later, Unity Personal version at *unity.com*
Blender 2.79b or later at *blender.org*
GIMP 2.10.2 or later at *gimp.org*
Audacity 2.3.0 or later at *audacityteam.org*



Next, it's time to look at how to work with these tools as illustrated in Figure 1.1.

You'll be using GIMP to make 2D graphics assets for your projects. You paint something in GIMP, export to the asset directory used by your Unity project, and



▲ **FIGURE 1.1** Unity, Blender, GIMP, and Audacity Workflow.

then you're immediately ready to use the graphics in your Unity project. Unity automatically imports the graphics. This allows you to make changes to your graphics in GIMP, export, and immediately test the changes in Unity. If you prefer you may use another paint program of your choosing. You merely need to export to one of the many supported graphics file formats such as .png, .tif, or .jpg. Alternatively, you could use a camera or scanner to make graphics files. Do you want to put a picture of yourself or your pet into the game? No problem. Just take a digital photo and put it into our Images directory. You'll also have the option of using GIMP to add effects to your images.

When making 3D art, the pipeline is a little more complex. You will start with making some 2D images, use them to make textured 3D models in Blender, and then use them in Unity. Blender allows you to create 3D models to be used directly in Unity. You'll use the textures from your Images directory to make the 3D models look better. Strictly speaking, you don't really have to have textures, especially for really simple games. But to make your games look more realistic you'll want to use textures.

Each graphic asset gets imported into Unity. Whether it's a spaceship, an alien, or an elaborate scene, it all goes through the art pipeline in order to be usable by Unity. Unity will be described in much more detail in the next chapter.

But hold on, don't forget about sound! You'll be adding some simple sound effects to your classic games. You'll be using the open-source tool Audacity to help make the sound effects. It would be easy to use a bunch of sound effects from an effects library. However, in the spirit of classic gaming, you'll be creating your sound effects from scratch.

HOW TO USE THIS BOOK

This book is intended to be read cover to cover, all the while carefully following the hands-on step-by-step instructions. This will give you a good foundation in the basics of classic game design and development. Along the way you'll learn some of the history and gain an appreciation of the pioneering arcade games that launched the video game industry.

If you're an experienced programmer and game developer, you might be able to dive right into the projects in the later chapters. You could also just load the projects as starting points for your own experiments.

The projects are all available on the companion DVD for this book.

If you're just interested in the programming aspects of these projects, you can just use the art and sound assets and follow along with the programming steps. Similarly, if you want to learn how we made the graphics, you can skip our programming and sound discussions.

You can just read the chapters about the featured classic games and learn about design without worrying about the technical implementations. However, it's recommended that you work through one or two of the classic game projects. The best way to learn anything is "learning by doing." This is especially true for the daunting task of learning how to be a game designer.

In the next chapter, you'll get started by taking the software out for a quick spin.

CHAPTER 2

Tools of the Trade

IN THIS CHAPTER

The goal for this chapter is to create a small demo application with 3D graphics and sound. You'll install Unity, Audacity, Blender, and GIMP. You'll create assets in GIMP, Blender, and Audacity. Finally, you'll see how these applications fit together by importing assets into Unity and setting up the demo.

INSTALLING UNITY

If you haven't done so already, go ahead and install Unity on your computer. None of the software tools in this book require a high-end system, but it's a good idea to use your fastest system with the best monitor setup. In order to run the tools in this book, you need access to a relatively recent PC or Mac. You're probably OK with a system built after 2015. If your computer is older, it may still be compatible depending on the particular capabilities of the system. To find out, go to the Systems Requirements webpage on the Unity website for details. Once you've determined that your system meets or exceeds those requirements, the next step is to install Unity version 2018.3.0f or later at *www.unity.com*. If you qualify for it, select the personal edition, which is free to use. This book is also compatible with the paid versions of Unity.

If you are using a somewhat later version of Unity there's a good chance that you can follow the steps in this book, with possibly some minor adjustments along the way. Be aware that the screenshots in this book may not match your screen if you do that.

If you are reading this book several years after 2019, then you'll be best off installing the exact version recommended above, or look at *www.classicgamedesign.com* for updated information on compatibility with the latest version of Unity.

This book supports both PCs and Macs. The projects in this book were originally developed on a PC and then tested on a Mac. The step-by-step instructions are designed to work on both PCs and Macs. The screen shots used to generate the illustrations in this book were captured on a PC, so if you're using a Mac you might notice some cosmetic differences between your screen and the PC screen shots in the book. When necessary, the book explains differences in performing the steps on a PC vs. on a Mac.

It is highly recommended that you use an HD monitor, preferably with a resolution of 1920x1080 or better. A dual or even triple monitor setup is definitely a plus and well worth it, considering the relatively low cost. Another possible setup is a laptop with an external second monitor to be used when convenient. It is also recommended that you connect a three-button mouse with a scroll wheel. Using a touchpad on a laptop is very cumbersome. For Blender, a full keyboard with a numeric keypad is much better than smaller keyboards. There is a workaround by configuring Blender for laptop use. Instructions on how to do that can be found online by searching for "configuring Blender for laptop."

If you're using a high DPI (dots per inch) monitor, you might find that the Unity fonts are too small to read. There is currently no setting inside of Unity to increase the font size, but you can do this in Windows 10 by going to the display settings and adjusting "Change the size of text, apps, and other items" to something higher than 100%. This screenshots for this book were produced on a Windows 10 machine with a 4K monitor and a 250% text size setting. You may need to make a similar font adjustment if you're using a Mac with a high-end 4K or better monitor.

The next section describes how to make a minimal coding application in Unity, which traditionally is called a "Hello World" program.

HELLO WORLD!

Before you create the demo application it's a good idea to make something even simpler. The phrase "Hello World" has taken on a special meaning for programmers. It is customary for programmers to make a "Hello World" application when first encountering a new programming language or development environment. A "Hello World" application simply displays the words "Hello" and "World." This is about as simple as it gets, and yet it can take quite a bit of time and effort to get this done. It's not intended to be a true test of the power of Unity. Rather, it's a simple exercise to make sure you can do something very basic.

This is the first step-by-step process in the book. There are a bunch of them, so get ready to follow along on your own computer. It's easy to get lost, skip a step, or to not quite follow the instructions exactly as written, so please read each step very carefully before trying the step on your own.

After the initial description of a step, there often follows a more detailed explanation. You may wish to read the explanations before doing the associated steps. It is often a good idea to read ahead by several steps to get a sense of where you're headed.

Step 1: Start Unity.

Make sure you've successfully installed Unity 2018.3.0f2 or later on either your PC or your Mac as described in the previous section. Later, or even slightly earlier versions of Unity will probably work as well, though you might need to make some adjustments.

Step 2: Click on **New**

This icon is near the upper right corner of the window.

Step 3: Under Project name replace "New Unity Project" with "**HelloWorld**."

You don't need to enter the quotes, just the letters. There's no space between Hello and World.

Step 4: For Template, select **2D**, for Location a **directory of your choice**.

For the location it's a good idea to use a newly created empty directory called CGD, short for Classic Game Design. The idea is that you aim to put all projects from this book together into this directory for easy reference. In the location box you can click on the three dots to browse for a good location on your computer.

Step 5: Click on **Create project**.

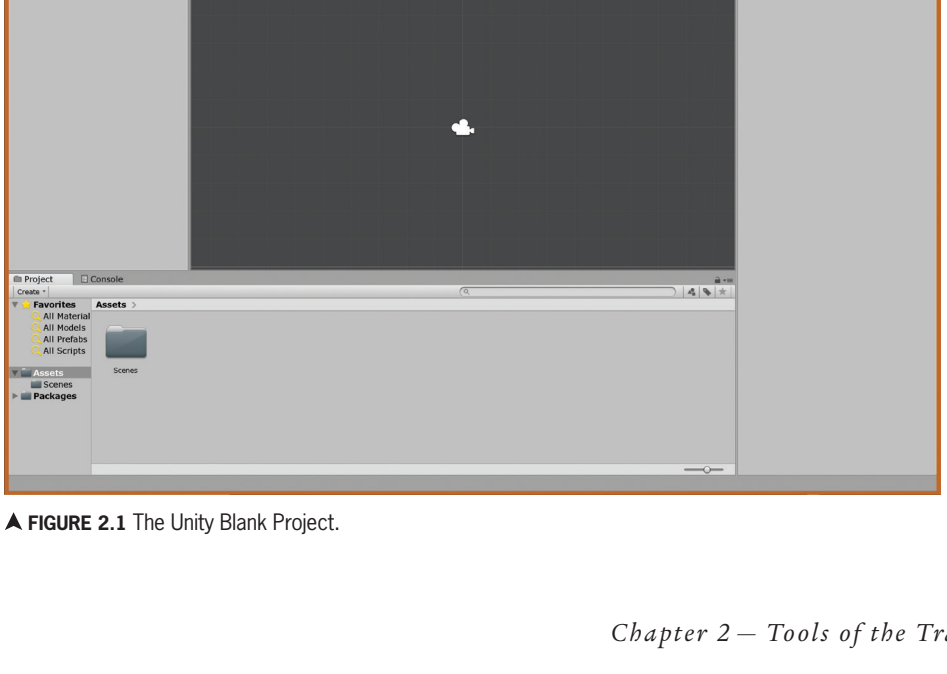
Depending on the speed of your computer, you'll now need to wait a minute or two while Unity creates the project. You'll see several messages while this is happening.

Step 6a: Click on **Layout** and select **Revert Factory Settings…**

If you're an experienced Unity user and wish to keep your existing settings intact, simply select the Project and Scene tabs instead.

Step 7: In the Scene panel, look for the 2D icon and click on it if it's not highlighted.

Your screen should look similar to the screenshot in Figure 2.1.



▲ **FIGURE 2.1** The Unity Blank Project.

Step 8: Click on the Layout drop-down menu and select the **Default** layout, if not already selected.

If you wish, go ahead and try out the other layouts, but switch back to the Default layout before continuing on. Notice the text on the top middle or top left of the window. It should say something similar to:

```
Unity 2018.3.0f2 Personal – SampleScene.unity - HelloWorld – PC,
Mac & Linux Standalone <DX11>
```

This text on the title bar of the window gives you some basic info such as the name of the current project and the version of Unity. The <DX11> text is the version of DirectX used by Unity in this project. DirectX is only used on Windows computers. On a Mac you'll likely see <Metal> instead of <DX11>.

In Steps 9 through 13, you will create the "Hello World" text object.

Step 9: Click on **GameObject – UI – Text**.

Step 10: In the Hierarchy panel, highlight **Text**.

Step 11: In the Inspector Panel, Set **Pos X** and **Pos Y** to **0**.

You do this by clicking the number entry boxes and typing 0 for each. This moves the text object to the center of the camera view.

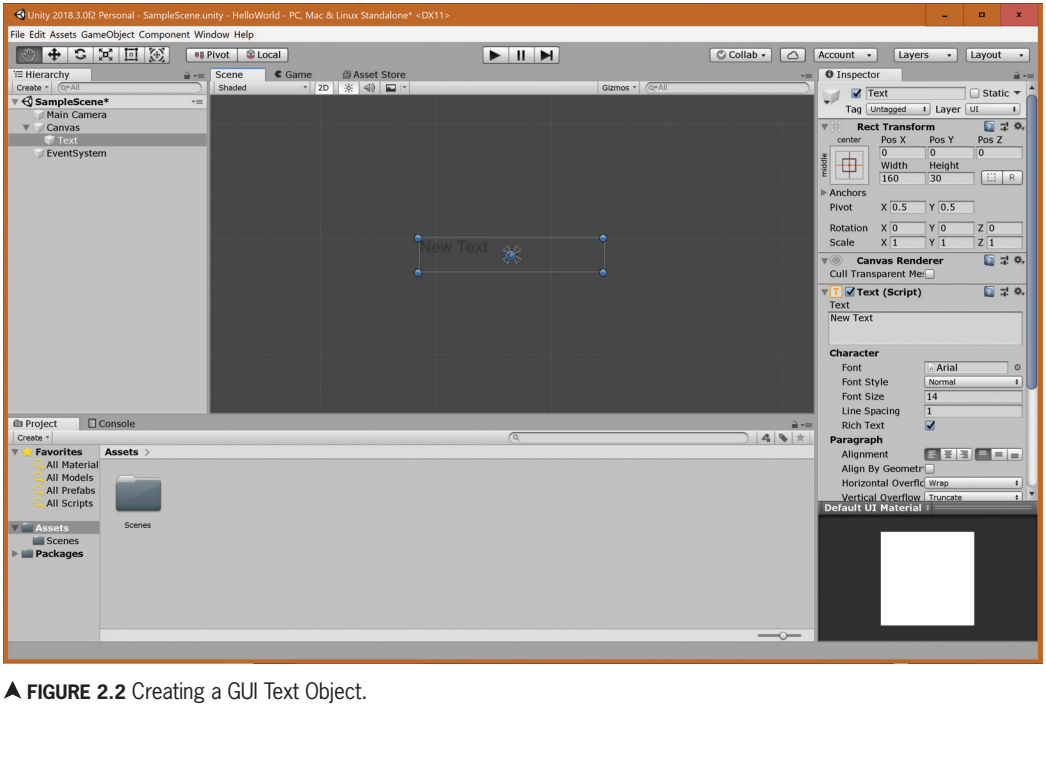
Step 12: Move the mouse to the Scene panel, and press the **f key** to focus on the text.

Your screen now looks like Figure 2.2.

If the Scene panel is blank, try clicking on "Text" in the Hierarchy, and repeat Step 12.

Before you move on and create the Hello World text, take a closer look at the Unity window. These last steps caused a great many changes to the screen. The next several paragraphs explain the layout of the Unity editor in more detail.

First, the window title now has a star at the end of it (PC version only). This is an indication that something has changed since you last did a save. Earlier in Step 8, you were asked to select the Default layout. If for some reason you have a different layout



▲ **FIGURE 2.2** Creating a GUI Text Object.

selected, please change it back to "Default" so your screen more closely matches the screen shots in the book.

There are four panels displayed currently: Hierarchy, Scene, Inspector, and Project. There are also unselected tabs for the Game, the Console, and the Asset Store panels. Each panel is displayed if you click on the corresponding tab.

On the top left of the Unity editor window is the Hierarchy panel. It displays four game objects in the current scene: "Main Camera," "Canvas," "Text," and "Eventsystem." The Text object is a subobject of the Canvas and is highlighted because it is currently selected. The Scene panel is the big, dark grey panel in the middle. It contains a graphical view of the current scene from the developer's perspective. There are four blue dots surrounding the Text game object and a string of characters with the current value of "New Text." In the next step you will change the text to "Hello World!".

The Game panel is at the same location as the Scene panel depending on the tab selection. It shows the game as it appears to the players. The players don't need to see the blue dots. They just see the text in the middle of the screen.

Next, on the right side of the window you have the Inspector panel. It shows the properties of the currently selected game object. Here is where you change the properties of your game objects by pointing, clicking, and typing.

The bottom section of the window shows the Project panel. It consists of a list of Favorites, Assets, and Packages. The Assets are shown in more detail in the subpanel to the right. This panel works much like the Windows File Explorer built into Windows. Try clicking on Favorites, then Assets, and then Packages. Now go back and click on Assets again. You will be using the Assets view throughout this book.

The Project panel shows items that make up the project. Currently, there's just one item in the project, the Scenes folder. You can double-click on the Scenes folder to reveal the one and only scene in the project, called SampleScene. Notice that there's a slider at the bottom right of the Project panel. Sliding this makes the asset names appear larger or smaller. Also, notice that the Inspector panel changes when you select items in the Project panel. To see the Text properties again you need to click on Text in the Hierarchy panel.

The Console panel is currently hidden behind the Project panel. To view it, click on the Console tab to select it. This is a panel used to display system messages. Unity prints error messages here, as well as debug messages generated by scripts. Click on the Project tab again to go back to displaying the Project panel.

You're now ready to change the text in the Text object. Click on Text in the Hierarchy panel if necessary. In the Inspector panel near the middle you'll see the property named "Text" with the value "New Text" in an editable box nearby.

Step 13: In the Inspector panel, click on "New Text" and change it to **Hello World!**

Notice that when you're making edits there, the display of the Text object changes in real time in the Scene panel and that the line that you're editing is highlighted in blue.

You could stop right now, but it's just too tempting to experiment with some other properties in the Inspector panel.

You want to make the text larger, so that means you change the font size.

Step 14: In the Text section look for **Font Size** and change it from 14 to **60**.

This has the effect of making the text disappear! To get it back change the overflow settings as follows:

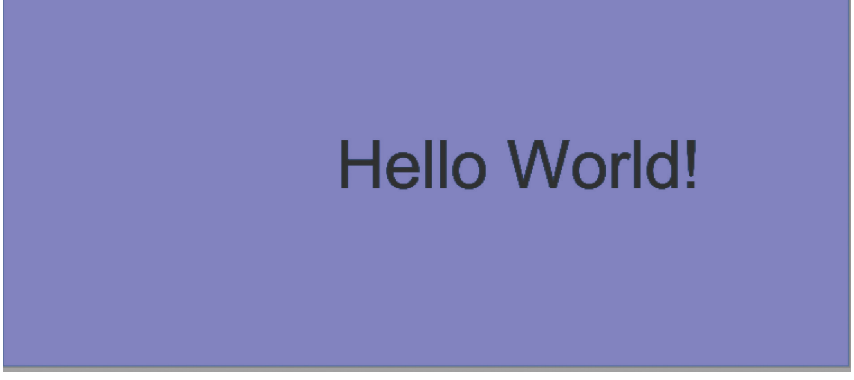
Step 15: Change Horizontal and Vertical Overflow to **Overflow**. Click on the **Game** tab.

You see that the text is much larger. The Game panel should look like Figure 2.3.

Next you will "run" the application inside of Unity. This application doesn't do much, but that's OK. Recall that the goal is to test running a very simple application in Unity.

Step 16: Select the Scene panel, then click on the **play button** ▶ above the Scene panel.

Unity automatically switches to the Game panel when the game is running. The game itself doesn't do anything yet other than display your Hello World! text. Currently,



▲ **FIGURE 2.3** Big Hello World! in the Game panel.

it looks identical to the view of the Game panel when the game isn't running. You can tell that the game is running by seeing that the play button is highlighted.

Step 17: While the game is running, go to the Inspector panel and change **Pos X** position to **100**.

You can change the properties of objects while the game is running. This is one of the amazing and useful features of Unity that make game development quick and interactive, but watch out! These changes are only temporary.

Step 18: To stop running the application, click on the **play button** again.

The Scene panel just came back, but notice that the Pos X is back at 0, which is where it was before you started running the game.

Step 19: Run the application again by clicking on the **play button**.

You can see that when you run the game now, the text is again the middle of the screen, just as before.

Step 20: Click the **play button** yet again to stop the application.

Step 21: Click on **File – Save Project**.

If you should forget to save before exiting, Unity will remind you to save.

Step 22: Exit Unity.

Unity will remind you to save your scene if you forget to save. It's a good idea to save your scenes and your project frequently. The most recent scene that you worked on will be automatically loaded next time you start up Unity with the same project.

This HelloWorld project is a test to see if you can do the very basics of Unity. Of course, Unity can do much more than display text, but you took your first steps. This philosophy of trying out the simple features first before moving on to more complex and difficult ones fits in with a recurring theme in software development: Test everything early and often.

You might have noticed that there was no coding necessary yet. It's time to try some programming.

PROGRAMMING WITH C#

Before you get started with programming in Unity, it's time for a quick history lesson. Unity initially supported three programming languages: C# (pronounced C sharp), JavaScript, and Boo. The first edition of this book used only JavaScript. Recently, Unity has stopped supporting JavaScript and Boo. Thus, you will be coding in C#. C# is a simple, modern, powerful, general-purpose, object-oriented language. Learning all of C# is a daunting task, but it's fine to use just the basic features of C# when getting started.

You're going to dive right in and write a small program. Your first goal is to add 2 and 2 and to display the result on the Hello World game panel.

Step 1: Look at **Appendix I, Introduction to C# for Beginners**, near the back of this book.

If you're very new to programming, it's recommended that you carefully read that appendix before continuing. If you're an experienced C# programmer, you can safely skim through that appendix and move on to the next step.

Step 2: Open Unity and load the **HelloWorld** project.

Step 3: Click on the **triangle next to Canvas** and then on **Text** in the Hierarchy panel.

The Inspector will once again show the properties of the Text object. At the bottom of the Inspector panel, there's an "Add Component" box. You'll be using that box to add a small C# program to this object. You may need to scroll the Inspector panel down to see the "Add Component" box, depending on the size and resolution of the Unity window.

Step 5: Click on the **Add Component** box.

Step 6: Click on **New Script** at the bottom, and then change the name of the script to **Testing123**, and click on **Create and Add**.

Step 8: Click on **Testing123** in the Assets panel.

Depending on your window resolution, you might notice that the script name is truncated to something like "Testing1…". The ellipsis (the three dots) appears if the asset name is too long to fit underneath the asset icon. To see the full asset name, slide the icon slider at the bottom right of the Assets panel all the way to the left, or all the way to the right.

There is now a default script in the Inspector right below the label "Imported Object," consisting of the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Testing123 : MonoBehaviour {
    // Start is called before the first frame update
    Void Start()
    {

    }
    // Update is called once per frame
    void Update()
    {

    }
}
```

If you are reading this book in color, rather than on a black-and-white device, you will notice that there are blue, green, teal, and black words in the code as printed in this book. This is called color coding and it can be very helpful when editing code. For example, keywords are blue, class names are teal, comments are green, numbers and operators are black. The colors may be different on your computer. There is no color coding in the Inspector panel.

This is the code where you'll insert additional code to compute 2+2. You'll be editing this script in the following steps.

Step 9: Double click on **Testing123** in the Assets panel.

After a delay of several seconds, the Visual Studio window should appear. It contains the same code you saw earlier, but now it's editable and color coded. If Visual Studio doesn't start up, try reinstalling Visual Studio Community Edition. It should have been installed when you installed Unity.

It may be helpful for you to know that in earlier versions of Unity an application called Monodevelop was used instead of Visual Studio. This is the reason for the rather esoteric term "MonoBehaviour." It is still possible for you to use Monodevelop instead of Visual Studio if you prefer. This book uses Visual Studio for code editing.

Step 10: Edit the script so it looks like the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Testing123 : MonoBehaviour {
   // Use this for initialization
   void Start()
   {

   }

   // Update is called once per frame
   void Update () {

   }

   // Testing out C# in Unity
```

```
    private void OnGUI()
    {
        int result = 2 + 2;
        GUI.Box(new Rect(10, 10, 240, 40), "Result is " + result);
    }
}
```

This is the first time you're editing text into Unity, so it's time to do some experimentation. In the following steps you'll test the code to see if it's working.

Step 11: Save the code! Use **File – Save** or the keyboard shortcut Ctrl-S on the PC, Command-S on the Mac. This is a truly important step and you can easily get bitten if you forget to do this. This is a good keyboard shortcut to learn because you'll need to save frequently.

Step 12: Run the game by clicking on the play arrow.

If you're somewhat lucky, you'll see the message telling you that "Result is 4." If you're not so lucky, you'll get some kind of error message. Carefully check all of your code and make sure that it exactly matches the text in the book, and that you saved your file in Visual Studio.

If nothing is happening at all, check the function name `OnGUI` for typos. If, for example, you typed `OnGui` instead, you'll get no error messages and no output from the function.

Now you'll carefully go through the code and attempt to fully understand it. The `using` statements are automatically generated by Unity when editing a new C# file. They can be safely ignored for now, except that they need to be there in order for the code to work. The `public class` statement sets up the `Testing123` class. The name in the code needs to match the filename. This is important to know when you copy a file, change the name of the file, but then forget to change the name of the class. The `Start` and `Update` functions are blank for now. This is where you would insert code for initialization and updating of animations and game logic.

The `OnGUI` function is a built-in feature of Unity and gets executed periodically to display user interface items. As stated above, if you have a typo in that name you won't get an error message, but the function then just sits there and doesn't get executed. Try temporarily changing the name to `OnGui`, for instance, and watch how suddenly the GUI message is no longer displayed when you run your code.

The variable name `result` is one that you made up. It could have easily been something else, such as `WeirdValue`. The only thing that matters is that the two occurrences of the name in your code match each other exactly. And, of course, there are rules for which characters can or cannot be part of a variable name. These rules differ among programming languages. The rules for C# are the same as for C and C++: variable names may have letters, digits, and the underscore character (_). The first character must be a letter. Case matters, and C# keywords such as `class` may not be used as variable names. Notice that it's not OK to use spaces. This is why `WeirdValue` is fine but `Weird Value` is not a valid variable name.

The `GUI.Box` line creates the output that you see in your window. The numbers `10, 10, 240, 40` are coordinates of the rectangle that contains your output. The `"Result is "` string gets displayed together with the value of the `result` variable. The plus sign before `result` is not addition but string concatenation, an operation that takes two strings and merges them together one after the other. Notice the space at the end of the `"Result is "` string. Without it the output would be `Result is4`.

The `result` variable contains an integer but automatically gets converted to a string when concatenated with another string. The last line contains a single curly bracket. It closes an earlier open curly bracket. The two brackets contain the code for the OnGUI function.

Step 12: Experiment with the code by changing some things and seeing what happens.

This is a somewhat free-form step. Keep your changes small and make sure to undo them when you're done. Some possible examples: multiplication with the star

character, such as 12*12, or change the string from `"Result is "` to `"I can put any old string in here "`.

If you've been lucky enough to avoid compiler error messages, this would be a good time to try out what happens when you do something wrong, such as forgetting a semicolon or a bracket.

Step 13: Save the scene and project.

It's time to move on and create some graphics.

USING GIMP TO MAKE AN IMAGE

GIMP is a fully featured, open-source, free program for creating and manipulating graphic images. If you haven't done so already, go to www.gimp.org and install GIMP on your system. If you have GIMP installed already, please verify that you have version 2.10 or later. If you have a later version, or a slightly older version, your installation should still be compatible, but that can't be guaranteed. In the following steps, you'll take GIMP for a quick spin and make an image.

Step 1: Start up GIMP.

Take a look at the main window. The title of the window is GNU Image Manipulation Program. GIMP is an abbreviation of that. It looks like Figure 2.4. Your colors and icon sizes may be different.

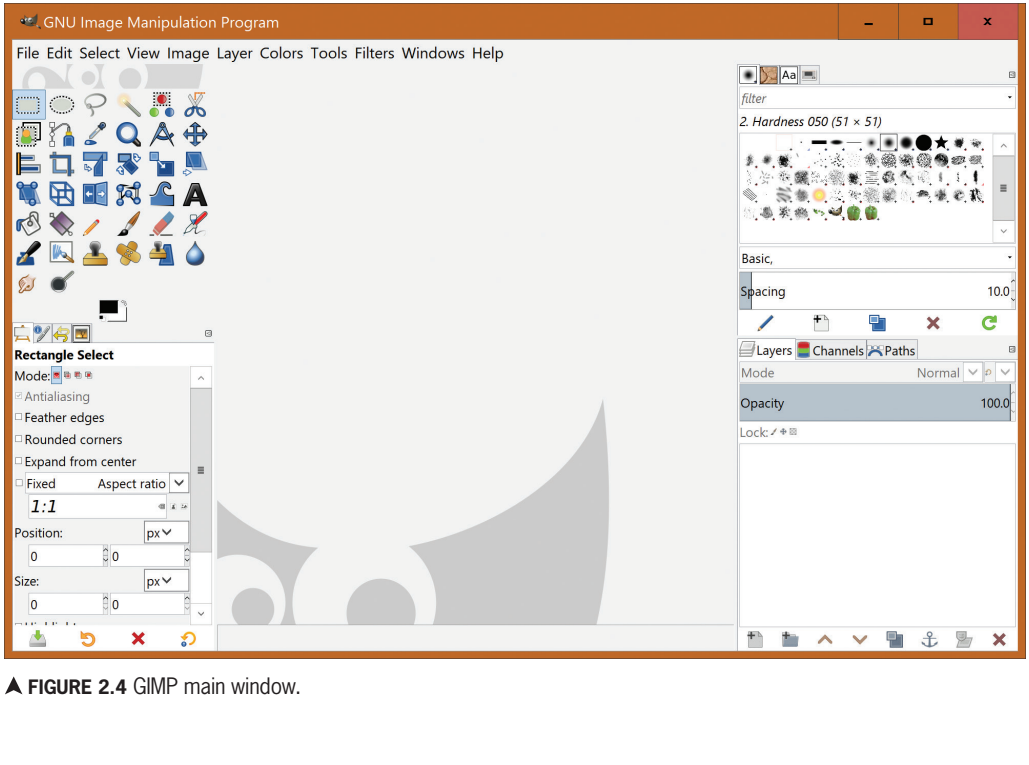
Depending on your previous usage of GIMP you may see additional windows. These additional windows are called "Dockable Dialogs." In this book, Single-Window mode will be used, a mode where these dialogs become panels on the right and left side of the main window.

Step 2a: Select **Windows – Single-Window Mode** from the main window.

Before you move on, you'll adjust the visual appearance of GIMP.

Step 2b: Adjust **Preferences** as explained below.

The Preferences window is found by clicking on the drop-down menu under `Edit` (`GIMP-2.10` on a Mac). Find the `Interface` section and click on `Theme`. If you want



▲ **FIGURE 2.4** GIMP main window.

to match the theme used when producing this book, choose the `Default` theme (`Light` on a Mac). You may wish to try out the other themes and choose one to your liking. The icons on the left and right of the GIMP window may possibly be too small for you, depending on which monitor you're using. In that case you can increase their size by clicking on the `Icon Theme` and select `Custom Icon Size` with an icon size setting of `Large` or `Huge`. This book uses the Color Icon Theme.

You're now ready to start using GIMP.

Step 3: Select the **File** menu and select **New…**.

Step 4: Set the **Width** to **1024**, **Height** to **768**, Units to **px** (pixels).

Step 5: Click **OK**.

A solid white image appears. That's where you'll create the image. Next, you'll need the Tool Options dialog. The next step is only necessary if you don't have that dialog visible.

Step 6: Select **Windows – Dockable Dialogs – Tool Options**.

If you have the Tool Options already visible, they will flash twice to show you where they are.

Now comes the fun part. You're going to fill your image with dried mud.

Step 7: Select **Bucket Fill Tool** in the Tool box.

It's the icon that looks like a paint bucket. Once you have it selected, the Tool Options area will change to show the options for the Bucket Fill tool.

Step 8: Select a **Fill Type** of **Pattern Fill**.

There are three fill types, FG color fill, BG color fill, and Pattern fill. FG stands for foreground, BG is background.

Step 9: Select the **Dried Mud** built-in pattern.

To select the pattern, first clear the text entry box immediately below the Pattern Fill text. Then start typing the name of the pattern (this is case sensitive) and hit return once the autocomplete shows the words "Dried mud." Move the mouse into the image area and notice that the mouse now has a paint bucket icon to let you know that you're about to do a bucket fill.

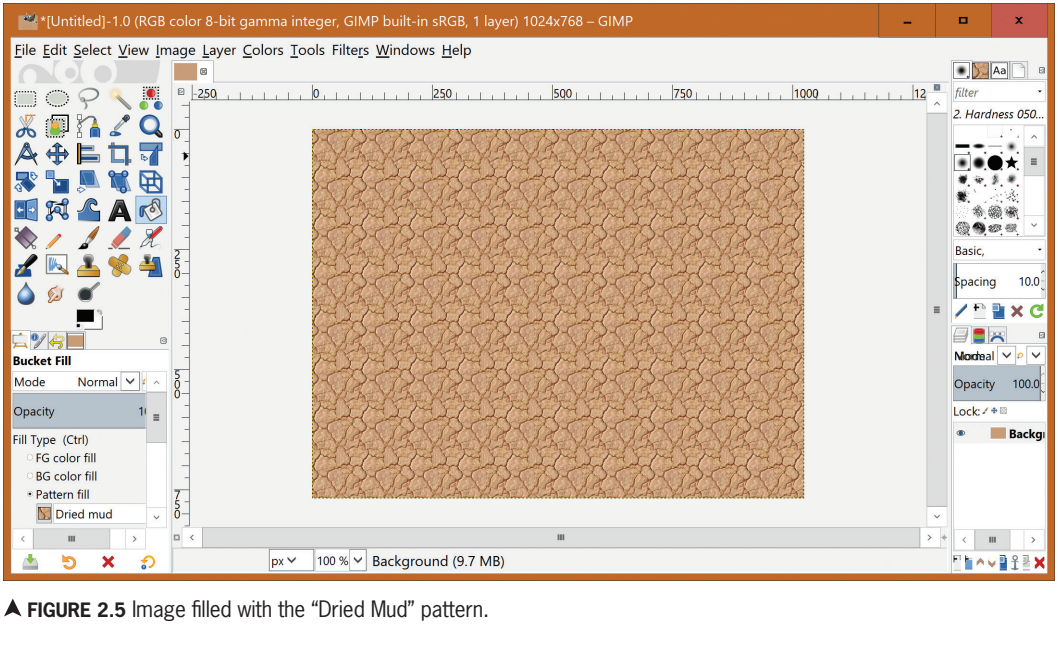
Step 10: Click anywhere inside of the image.

Your image fills with dried mud and should look like Figure 2.5.

This is going to be your texture image for your demo application. In this book, textures are saved in *png* format. PNG stands for Portable Network Graphics. This is a good format for textures because it has alpha channel support. It was designed to be an improved and unpatented replacement for GIF.

Step 11: Do **File – Export…** with name **MudBackground.png**.

Use the default export settings and use the Assets folder in your Unity project for HelloWorld. You just created a new image containing a repeating dried mud texture. Later on, you'll use GIMP to do some basic drawing with brushes and to manipulate existing images.



▲ **FIGURE 2.5** Image filled with the "Dried Mud" pattern.

Step 12: Do **File – Quit** to exit GIMP and **Discard Changes**.

You did an export, so there's no need to save the project as well.

In the next section, you'll get introduced to yet another tool, Blender, and use it to make a 3D object for your demo.

USING BLENDER TO MAKE A 3D OBJECT

Blender is an incredibly powerful, useful, and free program that allows you to make 3D objects for your games. If you haven't done so yet, install the latest stable version of Blender. It can be downloaded at www.blender.org. If you already have Blender installed, make sure that you have version 2.79b or later. You need to be aware that Blender made a very significant upgrade at version 2.8. This book was created using version 2.79b because at the time, 2.8 was still in beta and not fully compatible with Unity. If you are using version 2.8 or later go to www.classicgamedesign.com for the latest compatibility information. Version 2.8 has some fantastic user interface improvements, but if you are new to Blender you may wish to use version 2.79b in order to more easily follow the instructions in this book.

Step 1: Run **Blender** and **left-click** the mouse to dismiss the splash screen.

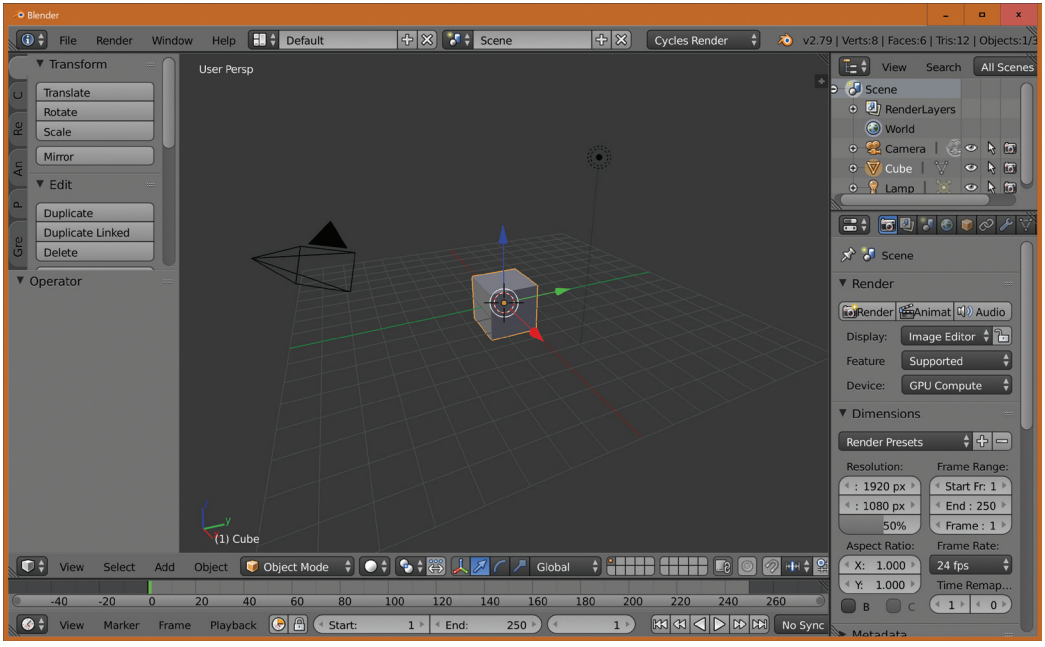
If you're an advanced Blender user, you may skip the following step.

Step 2: Click on **File – Load Factory Settings**.

Your screen should look similar to Figure 2.6.

The loading of the factory settings isn't really necessary if you just installed Blender, but if you've used this particular installation of Blender before it's a good idea to go back to the factory settings so your settings match the ones used by the book.

Before you start to use Blender, please be aware that the instructions in this book assume that you're using a three-button mouse with a scroll wheel and an extended keyboard with a numeric keypad. If you're using a trackpad or a smaller keyboard, for example, you can still follow along by learning how to change the user preferences. Instructions for this can be found on the internet. Mac users sometimes need to use different keyboard shortcuts because Mac keyboards are different from Windows keyboards.



▲ **FIGURE 2.6** Blender initial screen.

Step 3: Delete the cube by moving the mouse to the center of the window, press the **x** key and hit the **Enter** key.

Blender has this default cube set up every time you start the program. You don't need the cube this time, so you can start by deleting it from your 3D world.

Step 4: Click on **Add – Mesh – Monkey**.

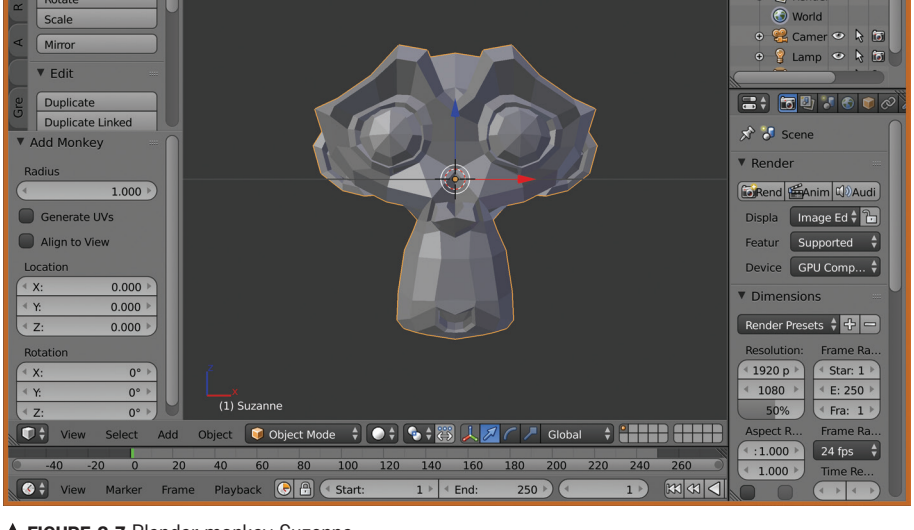
Yes, Blender has a monkey object built in. Her name is Suzanne. You'll put this monkey into your application.

Step 5: Move the mouse cursor back into the 3D panel, then hit the **1** key on your numeric keypad.

Step 6: Use the mouse scroll wheel to enlarge the view of the monkey head.

Your screen should now display Suzanne, as shown in Figure 2.7.

Blender is an enormous program. This book only scratches the surface. You can try out one of the more popular and powerful features of Blender by doing the following step.

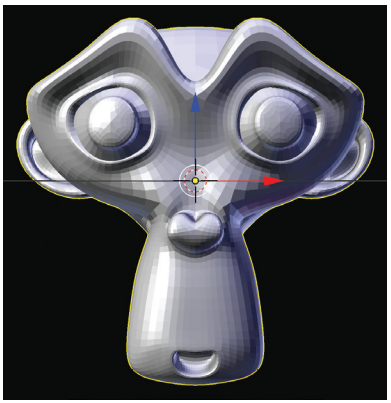


▲ **FIGURE 2.7** Blender monkey Suzanne.

Step 7, PC only: Hit **Control-2** to add a Subdivision Surface Modifier. Another way to get the same effect is this:

Step 7: Click on the **wrench icon** (Object Modifiers), click on **Add Modifier – Subdivision Surface**.

The monkey head magically looks much smoother. It's not really magic, but just a mathematical algorithm that inserts more faces for a smoother appearance. The smooth monkey head is shown in Figure 2.8.



▲ **FIGURE 2.8** Smooth monkey head.

Step 8: Save the file and name it **monkey.blend**. Make a note of where the file is on your system. You'll be using the monkey.blend file later in this chapter when you bring it into Unity.

Step 9: Exit Blender

Now take a look at Audacity, the free audio editor and use it to make a cool sound effect.

USING AUDACITY TO MAKE A SOUND EFFECT

Audacity is free, open-source, cross-platform audio software. Audacity is a multitrack audio-editor and recorder for Windows, Mac OS X, GNU/Linux, and other

operating systems. Audacity can be downloaded at www.audacityteam.org. If you already have Audacity installed on your system, please verify that you have version 2.3.0 or later.

It's surprisingly easy to make sound effects with Audacity.

Step 1: Start Audacity.

Step 2: Click on **Generate – Pluck…**.

If there are two Plucks in your version of Audacity, select the upper one.

You will be using one of the built-in sound effects, a synthetic plucking sound.

Step 3: Set the **Pluck MIDI pitch** to **32**.

Step 4: Set the **Fade-out type** to **gradual**.

Step 5: Set the **Duration** to **4.0 seconds** and click **OK**.

You just made a very cool looking audio waveform. Make sure that you have speakers or headphones attached to your computer and that the volume is set to a medium level.

Step 6: Click on **Play**  .

You should hear four seconds of a distorted sound effect, just what you want. If you don't hear anything, test out your computer with some other sound source, such as an online video.

This sound is a good start, but it's too long.

Step 7: Select the **right two seconds** of the waveform with the mouse.

You select portions of the waveform by clicking and holding the left mouse button, then dragging the mouse, then letting go of the left mouse button. The selected portion of the waveform is now highlighted. It's OK to overshoot and select past the four-second mark.

Step 8: Press the **delete** key on your keyboard.

The remaining waveform should be about two seconds long.

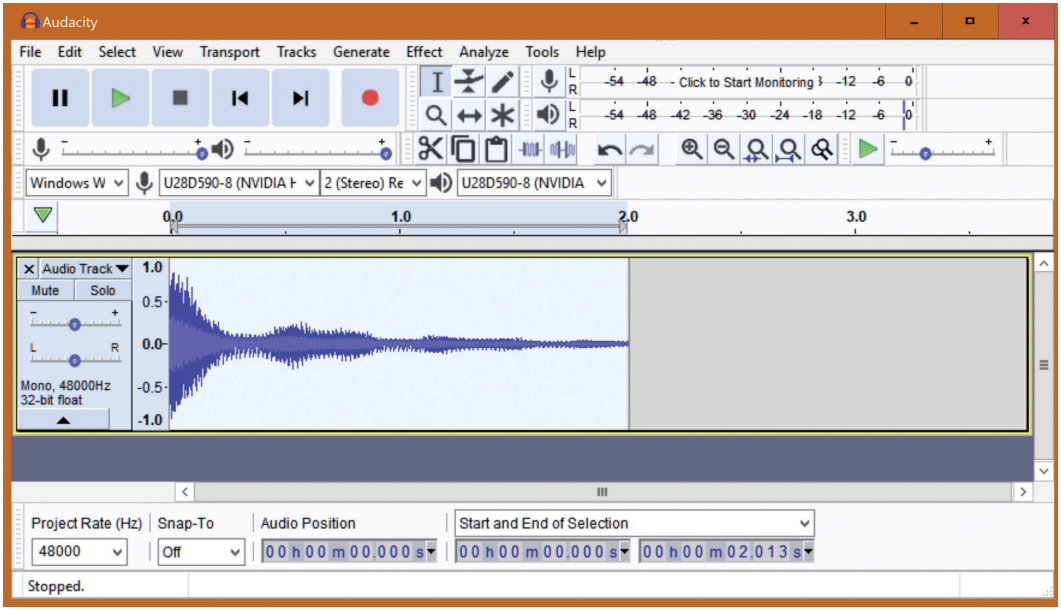
Step 9: Click on **Skip to Start**  , then **Play**  .

Make the sound more interesting.

Step 10: Click on **Select – All**, **Effect – Wahwah…** and **Apply, Close**, then click on the **Play** arrow to listen to it.

Your window now shows the new waveform as illustrated by Figure 2.9. If your window looks much darker than that, you have the dark theme installed. To use the light theme, go to `Edit – Preferences` (`Audacity – Preferences` on a Mac), click on `Interface`, and choose the `Light` theme.

Step 11: Click on **File – Export – Export as WAV** and use the filename **monkeysound.wav.** Use the Assets directory from the Unity HelloWorld project. You will see a Metadata entry window while saving the file. You may safely ignore that, leave the fields blank, and click on OK.



▲ **FIGURE 2.9** Audacity window showing Sound Effect Waveform.

Step 12: Exit Audacity.

The program will ask you if you want to save your work. That may seem strange because you just exported the sound effect. You may save now if you wish, but strictly speaking it's not necessary because the exported .wav file is all you need.

As you saw when you selected the Effect menu, Audacity has a large number of effects. Don't be afraid to try a few of them to see what they do. You're always just a few clicks away from creating something weird and brand new. It's also fun to record sound effects with a microphone and to then apply some effects on the recordings. More details on how to use Audacity for making realistic and strange sound effects will be discussed later on in this book.

Next, you're going to put the texture image, 3D object, and sound effect together in a demo Unity project.

USING UNITY: MY FIRST DEMO

In this section, you'll be making a demo application in Unity. The plan is to have the monkey head from Blender bouncing up and down on a textured playfield. You'll insert the sound effect and have it looping just because it's very easy to do that. This isn't a game yet, but simply a demo of how the development tools interact with each other.

Step 1: Start up **Unity**.

Unity shows a list of recent projects.

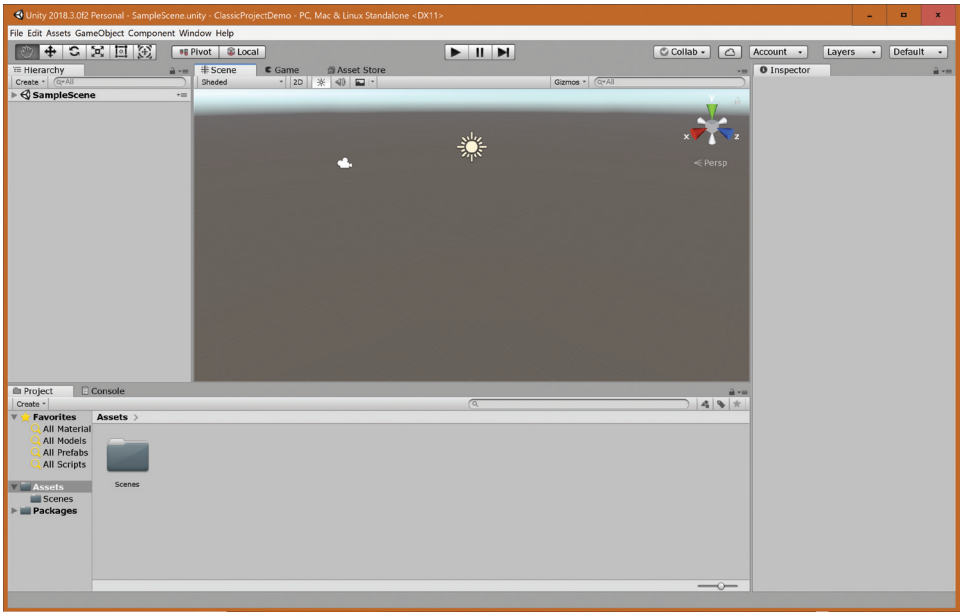
Step 2: Click on **New**

Step 3: Enter **ClassicProjectDemo** as the Project Name, **3D** Template

The HelloWorld project was 2D, this one is 3D.

Step 4: Click on **Create Project**.

It may take some time for Unity to create your project. When it's done, your screen should look like Figure 2.10.



▲ **FIGURE 2.10** Getting started with Unity demo project.

If your screen looks different from this, it might be because you have a different layout selected. Make sure that you are using the "Default" layout. If the top right box in your Unity window doesn't say Default, click there and select Default from the drop-down menu.

First, you'll create your playfield. It starts out as a cube and then you'll stretch it and rename it.

Step 6: Click on **GameObject – 3D Object – Cube**.

Step 7: In the Inspector panel, change the **Position X** value to **0**, **Y** value to **0**, and **Z** value also to **0**, if necessary.

Those positions are probably at 0 already, but if they're not, change them to 0.

Step 8a: Change the **Scale X** to **10**, leave **Y** at **1**, and change **Z** to **10**.

Step 8b: Rename the Cube to **Playfield** in the Inspector.

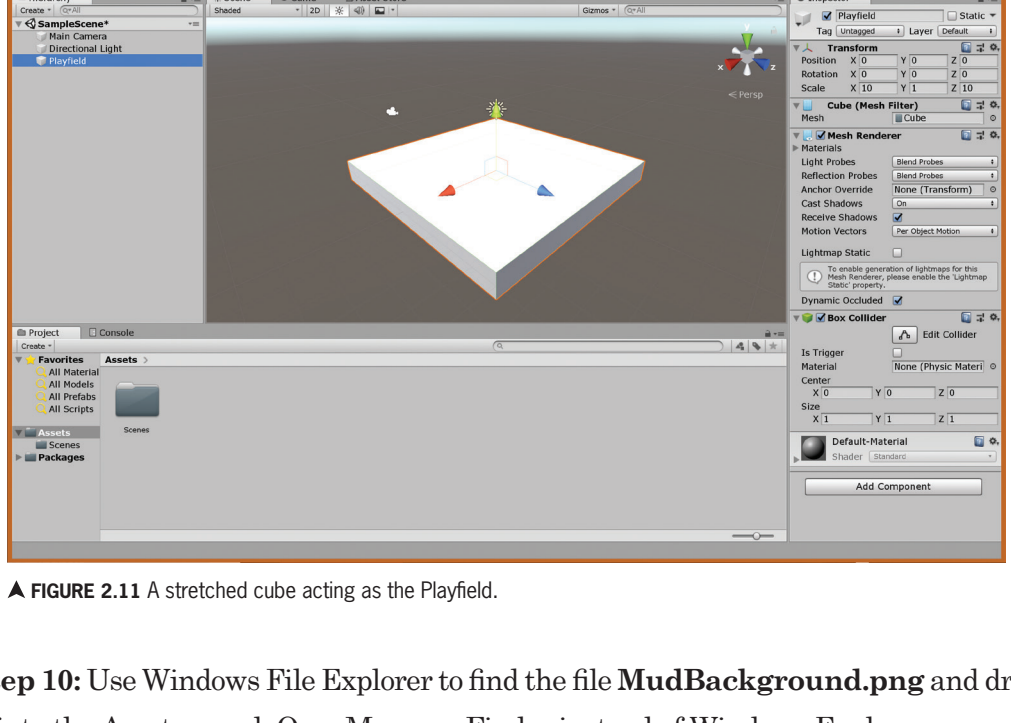
Giving your objects meaningful names is an important habit. Don't get lazy and skip this step! Yes, your game will still run with default names, but you'll quickly get

lost and confused with too many names like cube and sphere! By the way, the name for this object in the first edition of this book is Cube…

Step 9: Hover the mouse over the Scene panel. Press the **f key**, and **scroll** the mouse scroll wheel up to make the playfield larger.

Your Unity window should look similar to Figure 2.11.

You're now ready to start importing *assets*. The word "asset" refers to pieces of graphics, code, or sound that might be used in your scenes. The assets are all listed in the Project panel in the Assets subpanel. It's easy to add assets to a project. Just drag them into the Project panel from another window using the mouse.



▲ **FIGURE 2.11** A stretched cube acting as the Playfield.

Step 10: Use Windows File Explorer to find the file **MudBackground.png** and drag it into the Assets panel. On a Mac, use Finder instead of Windows Explorer.

This is the texture file that you created using GIMP a couple of sections ago. The Assets panel should now list two items, the Scenes folder and MudBackground.

Step 11: Drag the **MudBackground** asset on top of the **Playfield** object in the Hierarchy.

Now the playfield is textured with dried mud! Notice that you now have a Materials folder in the Assets panel. This folder was created automatically when the mudbackground texture was assigned to the Playfield.

The texture is hard to see, so you will change the tiling factor for the texture in the next step.

Step 12: Select the **Playfield, expand MudBackground** at the bottom of the Inspector panel and change the **Tiling** to **0.5** for both **X** and **Y** in the Inspector panel.

To expand the MudBackground, click on the triangle below the brown sphere in the Inspector panel. You can use the Tab key to move from the X text entry to the Y text entry.

Step 13: Select the **Playfield** again, move the mouse into the Scene panel, and scroll the mouse wheel to adjust the zoom level on the playfield so that you see all of the playfield.

After all that, your screen should look similar to Figure 2.12.

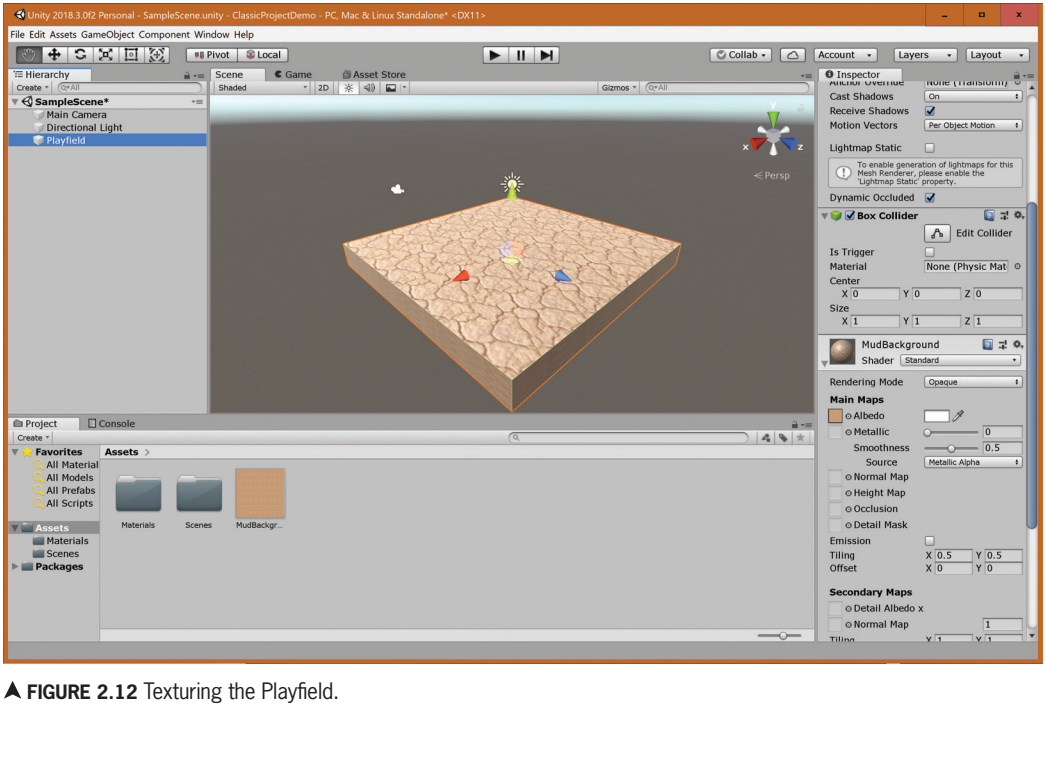
Your next asset is the monkey. Even though this is a very different file from the mud texture file, the importing of this asset works the same way.

Step 14: Drag the file **monkey.blend** into the Assets panel.

Step 15: Drag the **monkey** asset from the project panel to the Scene and drop it near the center of the playfield.

Step 16: With the **monkey** still highlighted, change the **X Position** to **0**, **Y Position** to **5**, and the **Z Position** to **0**. Also, change the **Y Rotation** to **180**, and leave the **X Rotation** at **–90**.

You can use the Tab key to quickly move through these text entry fields. It's possibly to click on the X Position text entry first, the enter the rest of the numbers using just the keyboard.



▲ **FIGURE 2.12** Texturing the Playfield.

Step 17: Right Click on the **scene gizmo**  and select **Back Perspective. Press f** to focus on the monkey.

The scene gizmo is in the upper right corner of the Scene panel. Try out the different built-in views just for fun, but use the Back Perspective view when you're done. Next, you'll change the color of the monkey to green.

Step 18a: **Create** a new **Material** in the Assets panel and rename it to **MonkeyMaterial**.

In the project panel there's a Create dropdown menu. Use it to select Material.

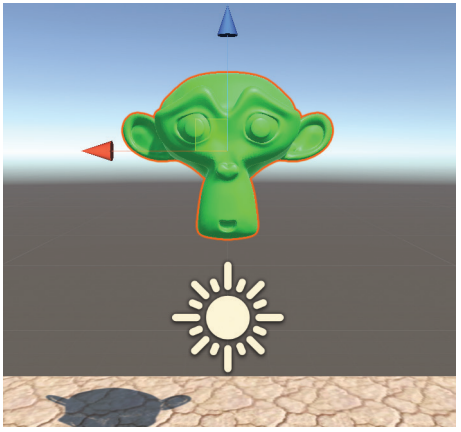
Step 18b: In the Inspector, change the color of **Albedo** to **green**

When you click on the Albedo color, a color dialog allows you to change the color. When you exit the color dialog, the MonkeyMaterial icon in the Assets panel should appear to be green.

Step 18c: Drag the **MonkeyMaterial** on top of the **monkey** in the Scene panel. Press **f** again and scroll the scroll wheel on your mouse to get a good view of the monkey.

When you do this, the monkey turns green even before you let go of the mouse button.

You should now see a green monkey head in the Scene panel as shown in Figure 2.13.



▲ **FIGURE 2.13** Green monkey.

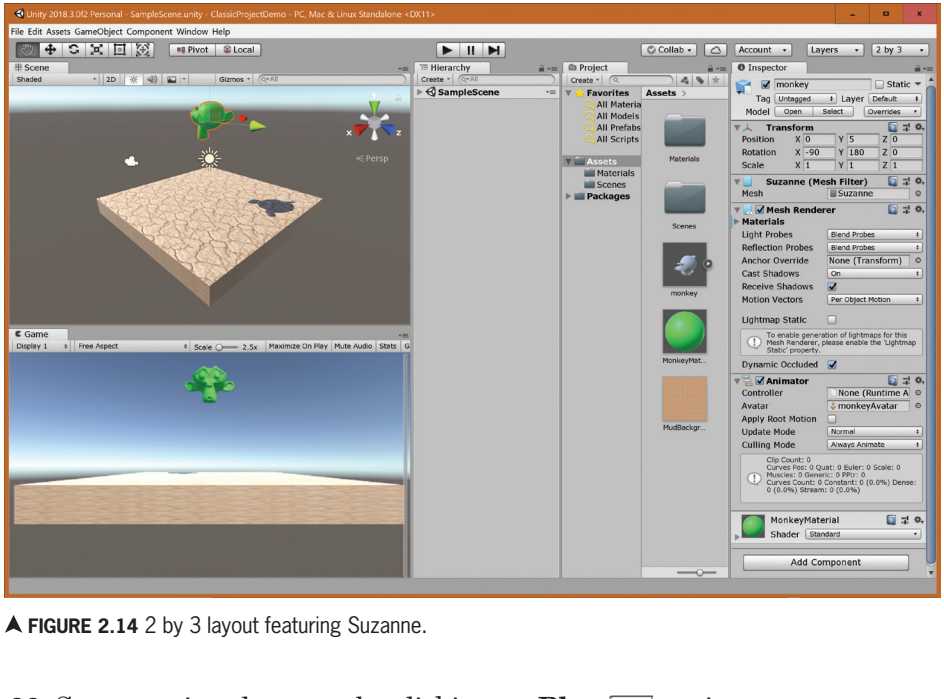
Step 19: Click on the layout drop-down menu in the upper-right corner of the window, currently at **Default**, and select **2 by 3**.

The 2 by 3 layout with two views of the monkey is shown in Figure 2.14. Your view of the Scene panel may be zoomed in more than in the Figure. You can adjust this view with the mouse scroll wheel as in the previous step.

Step 20: In the Game panel, select **Maximize on Play**.

Step 21: Click on **Play** ▶.

Running the demo right now causes the Game panel to cover the entire screen. The monkey sits there and doesn't do anything. The next few steps will animate the monkey by enabling the built-in physics engine of Unity.



▲ **FIGURE 2.14** 2 by 3 layout featuring Suzanne.

Step 22: Stop running the game by clicking on **Play** ▶ again.

This next step removes the animation that Unity automatically imported from Blender. You don't need this animation as it doesn't do anything.

Step 23: Expand SampleScene in the Hierarchy and highlight the **monkey**. Click on the **small star** ⚙ on the right side of the **Animator** component and select **Remove Component**.

Next you will add a Rigidbody component to the monkey to give it gravity and have Unity's built-in physics engine move the monkey for you.

Step 24: Select **Component – Physics – Rigidbody**.

Now when you play the demo, the monkey falls right through your playfield into a bottomless pit. That's better, but the monkey should collide with the playfield.

Step 25: Click on **Component – Physics – Sphere Collider**.

Now the monkey falls to the playfield and gets stuck slightly above it. Your goal is to make it bounce.

First, you need to create a *Physic material*. This unusual terminology refers to a set of physical properties.

Step 26: Click on **Create** in the Project panel and select **Physic Material**.

Step 27: Rename the new Physic material to **Bounce** in the Assets panel.

When you created the Physic Material, its name was highlighted in blue. You can immediately rename it at that point by just typing the new name. This method of renaming items in Unity works anytime you create something. To rename something later on you need to click on the name, wait a second or more, then click on the name again and type the new name.

Step 28: Find the **Bounciness** property in the Inspector and change it to **1**.

You might as well remove the friction as well, so do this:

Step 29: Change the **Dynamic Friction** and **Static Friction** to **0**.

Step 30: Drag the **Bounce** material from the Assets panel on top of the **monkey**.

You can drag it to the green monkey in the Scene panel, or the monkey in the Hierarchy. Note that the Sphere Collider now lists Bounce as the Material. To see this in the Inspector, you'll need to select the monkey.

Now the monkey bounces but the bouncing is very damped, and the bouncing stops very quickly. This is OK, but it would be more fun to have the playfield be bouncy as well.

Step 31: Drag the **Bounce** material onto the **Playfield** in the Hierarchy panel.

Now when you run the demo the monkey bounces and keeps on bouncing. The monkey doesn't quite reach the playfield when it bounces, but it's close enough for this demo.

Finally, you'll add some sound. Again, you're going to simply drag your sound asset, monkeysound.wav, into the Project panel.

Step 32: Find the file **monkeysound.wav** in the Assets folder of HelloWorld and drag it into the Assets panel of this project.

Having an asset in your project doesn't actually do anything. To activate it do the following:

Step 33: Select the **Playfield** object and click on **Component – Audio – Audio Source**.

Step 34: Drag the **monkeysound** asset on top of the **Playfield** in the hierarchy panel.

Step 35: Click on Playfield, then **Check** the **Loop** property in the Inspector.

The loop property is part of the Audio Source section in the Inspector panel. You may need to scroll down to see it. The loop property makes your sound repeat over and over. Notice that the "Play on Awake" property is already checked. That means that the sound will start looping as soon as you press play. Later on in this book you'll learn how to trigger sounds when objects collide with each other.

Step 36: Play ▶ the program and admire your handiwork.

Step 37: Save your scene and project, then **exit** Unity.

Now you know the very basics of running the tools and creating some assets with them. In the next chapter, you'll take a quick look at the early history of video games and a closer look at the arcade video game that started it all, *Pong*.

CHAPTER

3 Pong

IN THIS CHAPTER

Pong is generally considered to be the first successful commercial video game. Released in 1972 by a then unknown company, Atari, it had a great name and was an instant hit. *Pong* is truly the game that launched the commercial video game industry. In this chapter, you'll look at the history and design of *Pong* from various perspectives. You'll also get introduced to our first two classic game design rules.

BEFORE *PONG*

Before *Pong* there were tennis and table tennis, also known as Ping Pong. Both are Olympic sports with hundreds of years of history. More importantly, in the '70s both tennis and table tennis enjoyed great popularity around the world. Back then a majority of the US population knew the basic rules and had at least some experience with trying to play these games.

Pong wasn't the first video game. There's some debate on which one was in fact first, but the first commercial *arcade* video game was *Computer Space*, shown in Figure 3.1.

Amazingly, eight years before *Asteroids*, this game had Asteroids controls! You're flying a space ship with a thrust and two rotation buttons and you shoot at flying saucers.

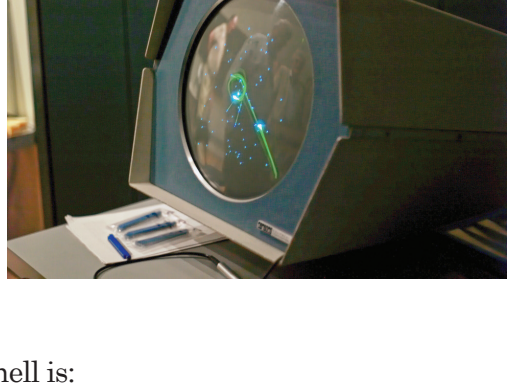
▼ **FIGURE 3.1** *Computer Space* screenshot.

Computer Space was created in 1971 by Nolan Bushnell and Ted Dabney, who would soon found Atari. *Computer Space* was not a big commercial success, probably because it was too difficult to learn. The screen also looks a bit busy, in great contrast to its much simpler and better-looking successors.

According to Nolan Bushnell, "Sure, I loved it, and all my friends loved it, but all my friends were engineers. It was a little too complicated for the guy with the beer in the bar."

Going farther back in time, *Spacewar!* (see Figure 3.2) is very similar to *Computer Space*. It was developed on a PDP-1 mainframe computer at MIT in 1962 by Steve Russell and others. DEC distributed this game with all PDP-1's and consequently it ended up at a large number of universities. Even more amazing, this game also had Asteroids controls.

FIGURE 3.2 *Spacewar!* PDP-1 (1962).



Another well-worn quote by Nolan Bushnell is:

All the best games are easy to learn and difficult to master. They should reward the first quarter and the hundredth.

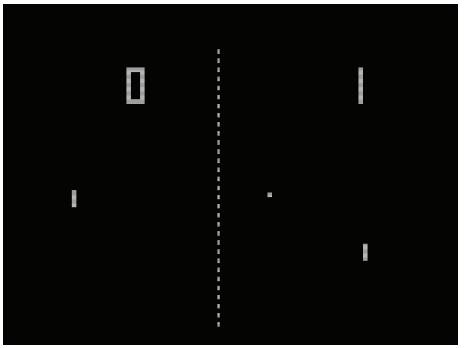
"Easy to learn and hard to master" has become a mantra for many game designers, especially arcade game designers in the '70s and '80s. Arcade games times average three minutes, so there just isn't much time for potential players to learn the games. Ideally the players would watch someone else play the game for a minute or two and would immediately feel that they, too, could do that.

All this led up to *Pong*. If *Pong* isn't easy to learn nothing is.

PONG, ATARI (1972)

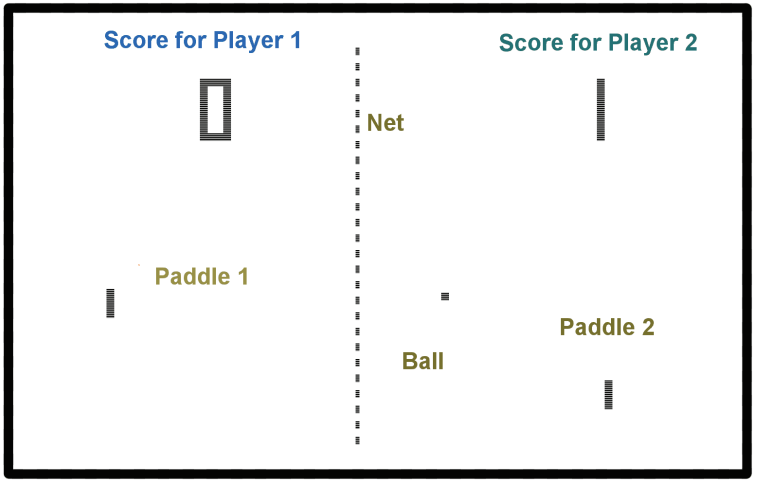
The gameplay for *Pong* is incredibly simple, even for 1972. Two players each control a paddle with a knob and try to keep the bouncing ball in play. *Pong*'s only differences from Ping Pong are that it takes place on a TV screen, the physics are simplified, and the players control their paddles with knobs instead of holding a physical paddle.

▼ **FIGURE 3.3** *Pong*, Atari 1972.



When you first encounter the screen, it looks like Figure 3.3.

The design elements consist of just six items: two scores for the players, two paddles, a ball, and a net. Figure 3.4 illustrates the design elements.



▲ **FIGURE 3.4** *Pong* design elements.

The players control the paddles and they have nothing else to do but to move the paddles up and down and try to make contact with the ball. The scoring is very familiar and self-explanatory.

The physics are simple and a bit unrealistic. There is no gravity, no friction, and no spin effect. Basically it's like Ping Pong in space. None of that matters though. In fact, it's partly because of the clean look and feel that the game was so successful.

This book introduces eight classic game design rules. Here is the first one:

Classic Game Design Rule 1: The Simple Rule: Keep it simple.

Simplicity is the hallmark of great design, and not just in games. The iPhone, *Pong*, the four-note theme for Beethoven's fifth symphony, Ernie Els's golf swing, and the pyramids of Egypt: all have a startlingly similar elegance. Designers often arrive at this simplicity via an arduous and complex path. Only rarely does the final design appear fully formed. Rather, years of development are needed to get there.

The enemy of Rule 1 is *featuritis*, a disease that can afflict even the best designers. Looking at the sequels of hit games, it's often apparent that the addition of features merely dilutes and spoils the original game. There are exceptions, of course, but great care must be taken when trying to improve upon a successful product.

Examples of why Rule 1 is important are everywhere. Consider Apple's iPod, iPhone, and iPad. Their phenomenal success is often attributed to their optimally simple user interface.

Here's the second rule:

Classic Game Design Rule 2: Immediate Gameplay Rule: Start gameplay immediately.

All too many modern games break this rule. People are impatient. They don't want to wait around, or read a bunch of rules. They want to start playing the game right away.

It takes some judgment to deal with this rule. A good way to look at it is this: Estimate the duration of the playing session, and allow for about 5% of that time for

instructions, cut-scenes, or the traversing of menus before starting with the actual gameplay. In the coin-op days of the '80s, 3-minute game times were the norm, which is 180 seconds; thus, the games wouldn't go over 9 seconds of introduction or instruction before allowing people to play.

Ideally, as in *Pong* for instance, the players would insert a coin and start playing just a few seconds afterwards.

It's tempting to write more rules now, but that would be a violation of Rule 1!

COIN-OP, THE REAL ATARI

Pong was the first product made by Atari. The people who made the games were hardware engineers. There were no programmers because the game was made entirely in hardware. It would take several more years until commercial games were programmed by game programmers rather than designed by hardware engineers.

In the mid-'70s Atari split into two groups: coin-op and consumer. The coin-op group always considered itself the "Real Atari" because most of the big hits originated as coin-op games. The consumer group, however, would soon be responsible for the vast majority of revenues.

PONG SEQUELS AND CLONES

Predictably, *Pong* led to a whole slew of arcade sequels and clones including *Pong Doubles* (1973), *Super Pong* (1974), and *Quadrapong* (1974), all by Atari. *Pong Doubles* added two more paddles so that four players could play. Atari also got into the home video game business with the *Home Pong* console. If you haven't done so already, this would be a good time for you surf the web and look at some images and videos of *Pong* and its sequels.

BITMASTERS, DAY ONE

Over 20 years later, in 1994, Bitmasters got a development contract to do a basketball game for Mindscape on the Genesis and SNES home video game systems. Bitmasters was a small game development company located in Sunnyvale, California,

just a few miles from the old Atari buildings. This was no coincidence, because several of the people at Bitmasters were ex-Atari employees, including Eric Ginner, Dave O'Riva, and the author of this book, Franz Lanzinger.

Day One of the basketball project was also Day One for several new programmers. None of them had ever seen a SNES system, much less programmed for it. So what would be the best way to teach them the basics? They all spent the day programming *Pong* using 65816 assembly language and proprietary Bitmasters software tools developed for previous SNES games. Amazingly, it took just one day for the new programmers to get a very good version of *Pong* up and running on their development systems.

PONG AT FORTY

Is *Pong* still a viable game forty years later? Yes! In 2012, Atari held a high publicity contest called the Pong Indie Developer Challenge. The winning entry was *Pong World*, published in November of 2012, four decades after the first *Pong* hit the arcades. This modern sequel is a much more complex game than the original, but the basic ideas behind *Pong* are still there.

What can you learn from this? Just as good literature, music, and art continue to thrive tens or even hundreds of years after their creation, so do the great classic video games. All game developers should keep this in mind when negotiating contracts with publishers.

It's also good to consider the far future when designing games. Can you imagine what a game console will look like in fifty years? Chances are the resolution will be higher, the processors faster, the storage larger. The controls will be different, maybe even unrecognizable. The constants are the rules, the product trademarks, the characters, the stories, and to some extent the basic game mechanics. A reasonable attempt to future-proof your game would include the following: stay away from fad controls, avoid cultural references to current events, and develop your art assets in a resolution-independent way.

In the next chapter, you'll start by developing your very own paddle game inspired by *Pong*.

CHAPTER 4

Classic Paddle Game

IN THIS CHAPTER

In this chapter, you're going to build your first game, a two-player paddle game similar to Pong. It's an exercise in building a prototype from scratch using Unity.

GETTING READY

As you can see, the title is Classic Paddle Game. This is a working title, intended to be replaced by the real title as some point. It is up to you to create a better title. Working titles are often chosen to be intentionally unusable for a commercial product, and this one's no different. You're going to do a game that's a very abstract version of Ping Pong. There are two players, and all they do is control their respective paddles to hit a ball back and forth across the screen. If a player misses hitting the ball, the other player gets a point. The first player to get to 11 points wins. That's the game in a nutshell, and this description is a rough guideline. You're perfectly free to change some things along the way. This game, unlike *Pong*, will use a physics engine and, just for fun, it'll be in color with 3D lighting effects.

VERSION 0.01: THE PLAYFIELD

Your first goal is to display the playfield. This is a very common first step in making games. Whether it's a detailed world in *Skyrim*™ or a blank canvas in *Pong*, you always need some kind of background. Your background in this game will be a green, rectangular shape with borders on the top and bottom.

Step 1: Create a new Unity project with the name "ClassicPaddleGame" in your Unity projects directory. Keep the default 3D Template.

Step 2: Use the **2 by 3** layout.

Upon startup, there is a blank workspace as shown in Figure 4.2. You should see the text "2 by 3" in the upper right-hand corner of the window. If you don't, activate the layout drop-down menu and select "2 by 3." You're now ready to create your game.

Step 3: Click on **GameObject – 3D Object – Cube** from the main menu and rename the Cube to **Playfield**.

Renaming can be done either in the Hierarchy or in the Inspector.

Step 4: Set the **Position** of the **Playfield** to (**0**, **0**, **0**) in the Inspector panel.

If it's at (**0**, **0**, **0**) already you don't have to do anything.

Step 5: Use the **Top Isometric** view.

To do this, right-click on the Scene Gizmo in the upper right corner of the Scene panel. Select Top and turn off Perspective.

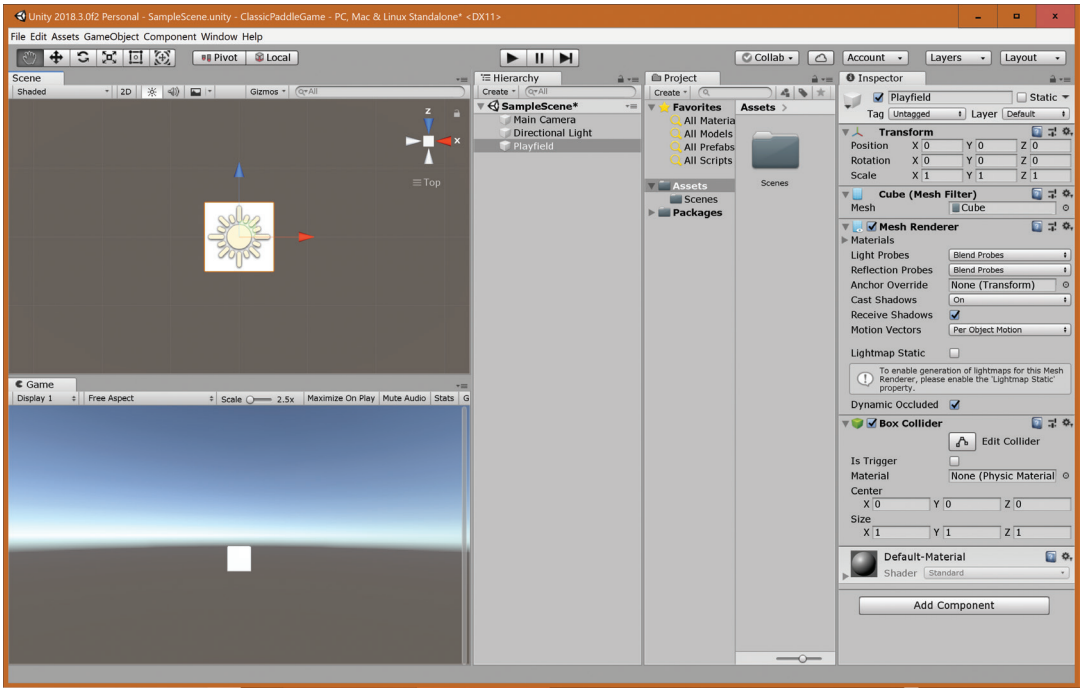
Step 6: Select the **Playfield** object, hover the mouse over the Scene panel, and then press the **f** key.

You should now see a white square in the Scene Panel, shown in Figure 4.1. The "f" key focuses the view in the Scene panel on the currently selected object. It's very useful for finding your current game object when it's gotten lost off-screen someplace, or if the zoom level is much too large or too small.

This is your starting point for the playfield. Make the Playfield larger by changing the scale.

Step 6: Set the **Scale** of the **Playfield** to (**30**, **30**, **1**).

This is done by clicking on X, Y, and Z in the Scale section of the Inspector window and entering the new values for X and Y. The Z Scale is already set to 1. You can speed this up by using the Tab key, as explained earlier in the book.



▲ **FIGURE 4.1** Creating a cube in Unity.

Step 7: Use the **Front** view and **focus** on the **Playfield**.

Just as you did with the top view, right-click on the Scene Gizmo and select front.

Next, you'll change the color of the playfield object. You'll do this by creating a material, assigning it to the object, and adjusting the color of the material.

Step 8: Click on **Create** in the Project panel and select **Material** and give it the name **Mat Playfield**.

Rather than renaming the material later, it's possible to immediately type the new name after creating the material with the default name "New Material."

Step 9: Change the **Albedo** of **Mat Playfield** to a slightly dark shade of **green**.

As you might recall from Chapter 2, to change color, click on the rectangle to the right of "Albedo" in the Inspector. Then use the pop-up Color Dialog to select a slightly dark green color. To set this color, first select green in the rainbow circle, and

then select a dark green color shade from the main square as shown in Figure 4.2.

Step 10: Assign the **Mat Playfield** material to the **Playfield**.

This is done by dragging the material with your mouse from the Assets panel to the Playfield in the Hierarchy panel or alternately in the Scene panel.

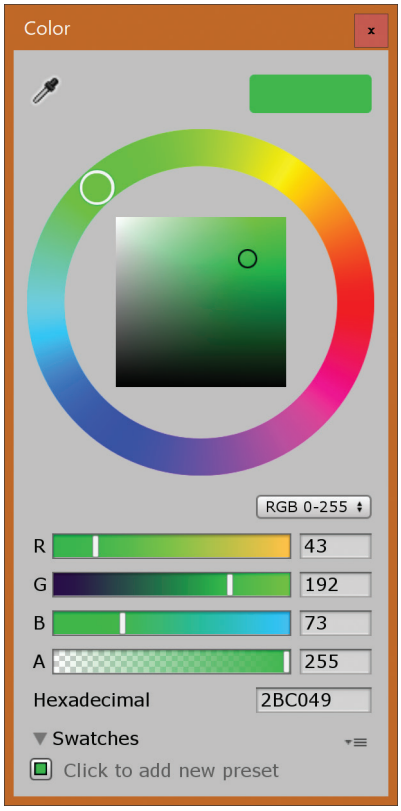
Step 11: Click on **Main Camera** and move it to (**0**, **0**, **-30**).

Step 12: Change the Type of the Directional Light to **Point Light**, move it to (**0**, **0**, **-10**), and change the **Range** to **100**. Then rename it to "Main Light."

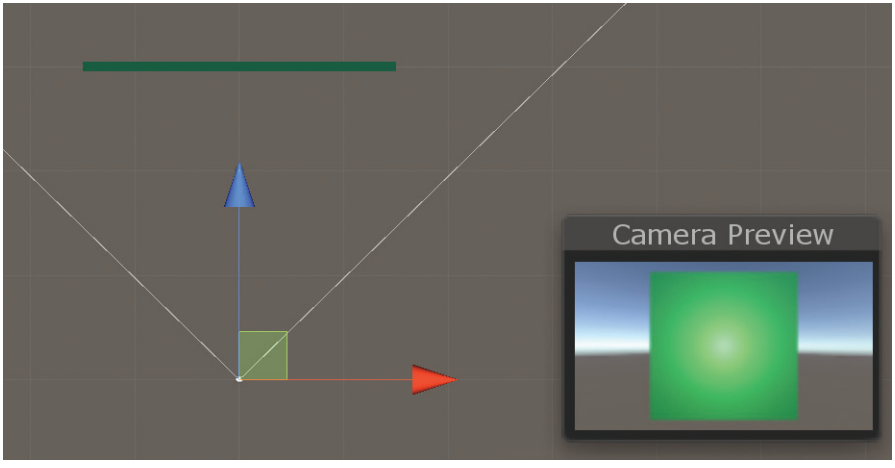
Step 13: Select the **Top** view in the Scene panel.

Step 14: Select the **Main Camera** object by clicking on it.

You should now get a good view of what's happening as shown in Figure 4.3.



▲ **FIGURE 4.2** Setting the color for the Playfield.



▲ **FIGURE 4.3** Camera moved back to reveal the entire playfield.

Step 15: Experiment with the Field of View slider.

Be sure that the "Main Camera" is still selected. The top view shows that the camera is in front of the playfield and the view lines emerging from the camera encompass the entire playfield. You can move the "Field of View" slider with your mouse to see the effect of changing the field of view. When you're done playing with the slider, put it back at 60 degrees.

You've done quite a bit of work and all you have is a green square! Still, this is a good point to test the game, just to test if you can see the green square when you hit the play button.

Step 16: Turn on **Maximize on Play** in the Game panel, if it's not already on.

Step 17: Click on **Play**. Then click on it again to stop play mode.

Step 18: Save and **Exit** Unity.

Your next goal is to create boundaries at the top and bottom of the playfield. Also, please note that from here on the instructions are slightly less detailed, now that you're getting more familiar with the Unity interface.

Step 19: Launch Unity. You should see the scene just as you left it when you saved it.

Exiting and starting up again is a good way to ensure that the project is saved properly. If you need to take a break it's a good idea to save and exit rather than to just let the computer sit there. This brings up a related issue, version control. Version control is a way to automatically keep track of multiple versions of your projects. This book keeps things simple by avoiding the added complications brought about by installing and using version control. With very small projects it's unnecessary to use version control. As your project grows you may wish to periodically save your project with a new version number appended to the filename. This allows you to restart your development at an older version if and when something goes wrong.

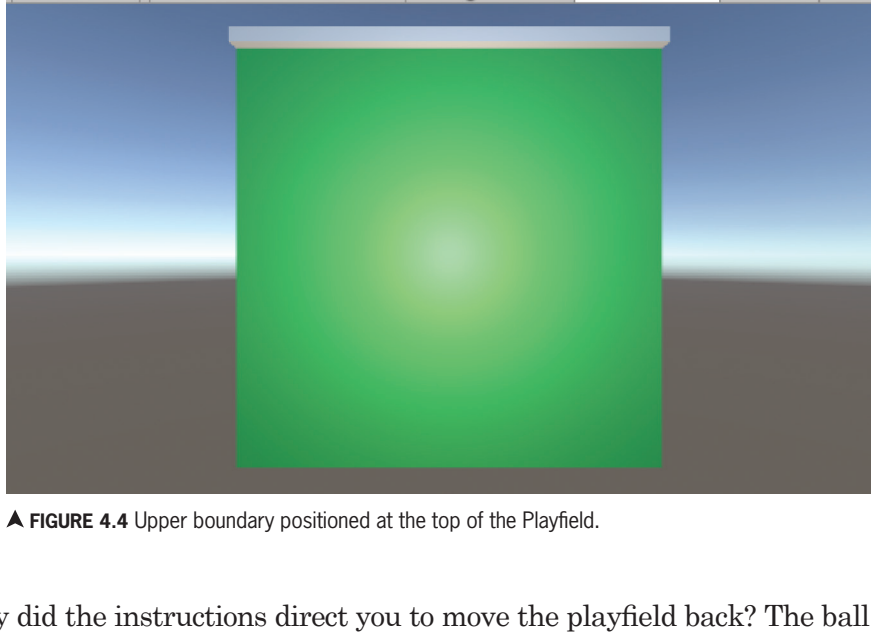
Step 20: Create a Cube, Position (**0**, **15**, **0**), **Scale** (**30**, **1**, **1**) with name **BoundaryUpper.**

There are two Create menus, one below the Project tab, and one below the Hierarchy tab. In this case you'll use the Hierarchy Create, or you could use the GameObject menu instead.

Step 21: Select the **Back** view in the Scene panel.

Step 22: Select **Playfield** and give it a new **Z Position** of **1.1** instead of 0.

The playfield just got a cool 3D quality to it. Your screen should look like Figure 4.4.



▲ **FIGURE 4.4** Upper boundary positioned at the top of the Playfield.

Why did the instructions direct you to move the playfield back? The ball is going to have a z-coordinate of 0, so you want the playfield behind it rather than at the same position. Notice the subtle 3D effect of the boundary because it is no longer overlapping with the playfield.

Step 23: Select **BoundaryUpper**, then right-click and select **Duplicate**.

Step 24: Rename the newly created duplicate to **BoundaryLower**.

Step 25: Select **BoundaryLower** and change the **Y Position** to **-15**.

Step 26: Save your work then **exit** Unity.

This is about as simple a playfield as you can have in a game. Commercial game projects spend millions of dollars developing just the playfields for their large worlds, but essentially, they are all just the stage and background for the true stars of the games, the animated characters. While it's certainly possible to skip making the playfield entirely, it's usually a good idea to have a simple playfield in place before doing anything else.

For the next version, you'll add the paddles for your paddle game and control them with your computer keyboard.

VERSION 0.02: THE PADDLES

The paddles are the player characters in this two-player game. They will be created using our usual technique of starting with cubes and scaling them.

Step 1: Launch Unity and load the project.

Step 2: Create a **Cube** and name it **PaddleLeft**.

Step 3: Change the **Position** of **PaddleLeft** to (**-14**, **0**, **0**) and the **Scale** to (**1**, **4**, **1**).

In case you're wondering, the 14 was determined by trial and error. The playfield is 30 units wide, so you'd think that -15 would be the correct x position, but you want the paddle to be offset a little bit away from the edge, so -14 seems about right.

Next, let's make the paddle red.

Step 4: Create a **Material** in the Project panel, name it **Mat Paddle**.

Step 5: Change the **Albedo** of **Mat Paddle** to **red**.

Step 6: Drag **Mat Paddle** onto **PaddleLeft**.

The paddle should now be red instead of grey. Next, you need to make the other paddle.

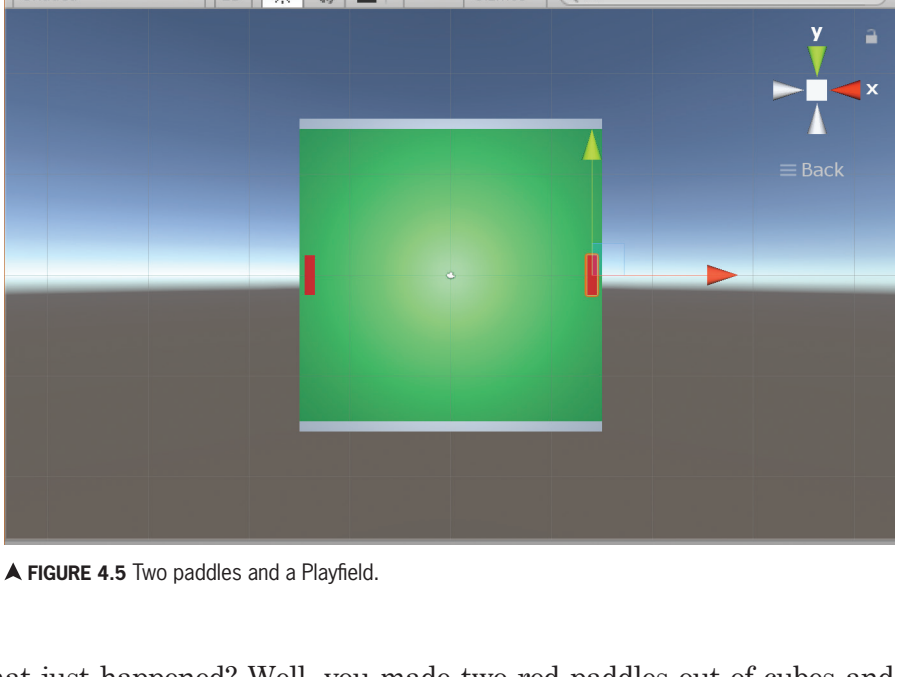
Step 7: Duplicate PaddleLeft.

This is done by selecting it, right clicking, and selecting "Duplicate" from the menu.

Step 8: Rename the duplicate to **PaddleRight**.

Step 9: Move **PaddleRight** to (**14**, **0**, **0**).

The Scene panel now shows the two new paddles, ready for action, as shown in Figure 4.5. You can use the Hand Tool icon in the top left corner to center the view in the Scene panel if necessary.



▲ **FIGURE 4.5** Two paddles and a Playfield.

What just happened? Well, you made two red paddles out of cubes and placed them on the playfield. You're now ready to make the paddles move in response to player inputs.

Step 10: Save.

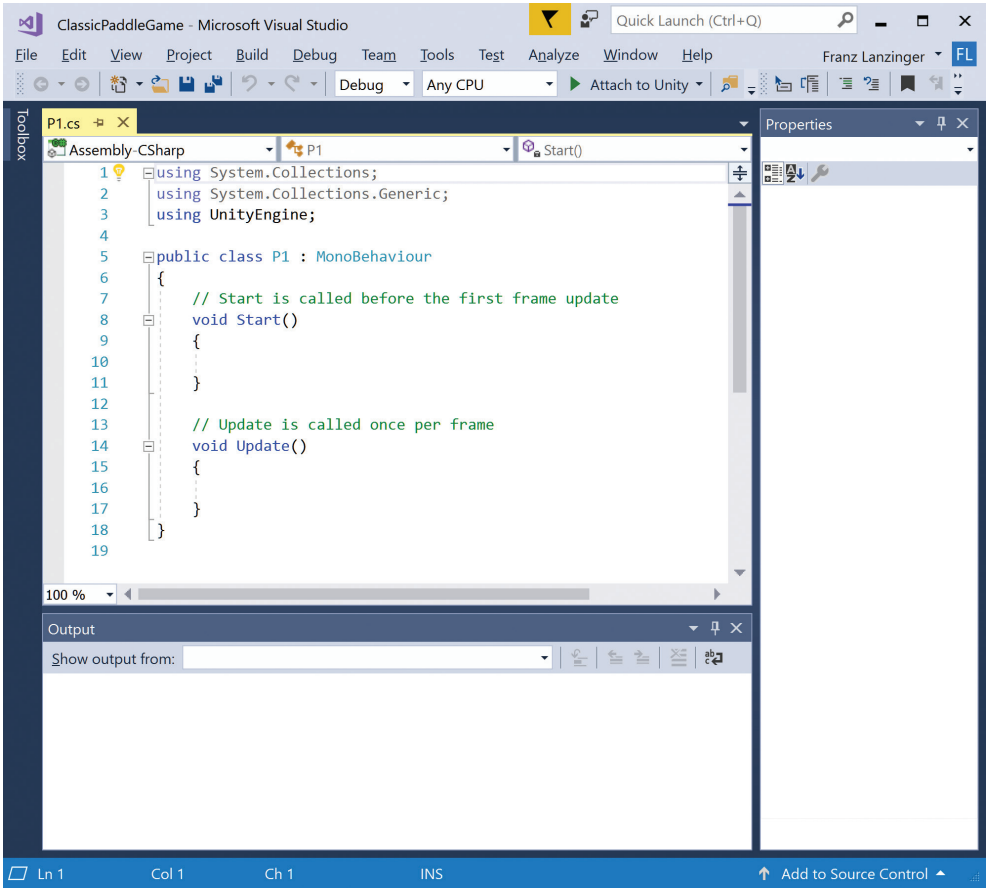
This step isn't really necessary, but it's a good habit to periodically save your work in case something goes wrong.

Step 11: In the Project Panel, click on **Create – C# Script** and rename it **P1** instead of NewBehaviour.

This is where you start putting in some code to make the left paddle move up and down.

Step 12: Select **P1** and click on **Open…** in the Inspector

A new window opens up. This is Microsoft Visual Studio, your chosen code editor for Unity. You'll be editing your code in this window, as shown in Figure 4.6.



▲ **FIGURE 4.6** Microsoft Visual Studio code editor.

You should see 19 lines of code. There are two functions, `Start` and `Update`, and they are empty. These are placeholder functions to help you get started. You will be editing the file P1.cs. You can see the name of the file in the top left tab above the code.

Step 13: Enter the following code to replace the `Update` function:

```
// Update is called once per frame
void Update ()
{
    if (Input.GetKey("w"))
    {
        transform.Translate (0, 20 * Time.deltaTime, 0);
    }
    if (Input.GetKey("s"))
    {
        transform.Translate(0, -20 * Time.deltaTime, 0);
    }
}
```

The code editor does a lot of work for you, but it takes some getting used to. Watch the screen as you type, and you'll see that Visual Studio balances parentheses, does smart indentation, and guesses keywords for you. It also warns you with red underlines if it thinks you made a mistake. These features can best be learned by diving in and typing code.

Step 14: Click on **File – Save Assets\P1.cs** in the Visual Studio Window. This saves your editing work in Visual Studio.

Always save your code right away. Notice that the filename P1.cs has a star next to it whenever there are unsaved changes present. On a Mac it's not a star but a small circle.

This `Update` function periodically checks the keyboard. When the "w" key is pressed down, it moves the current object by a few units of distance. In this case, the "w" key makes the object move up. The "s" key makes it go down.

Step 15: Drag the **P1** script onto **PaddleLeft**.

You'll need to click on the Unity window to make it the active window before doing this.

Step 16: Click on **Play** and press the **w** and **s** keys. If you did everything correctly the keys should move the left paddle up and down.

There's a chance that you made a typo or some other mistake along the way. If so, you'll probably get an error message. To fix the error, go back to Visual Studio, fix the problem using the error message as a guide, save your changes, and try again. You may need to do this several times. This is normal, even for experienced programmers, so don't give up if it doesn't work for you right away.

Step 17: Exit Play mode.

It's *very important* that you leave play mode by deselecting the play arrow before you make changes that you wish to be permanent. If you forget to do this, everything you do during play mode will be lost when you finally remember to stop play mode! This is a nasty surprise waiting to happen. As long as you have "Maximize on Play" selected it's much less of a problem, so be sure to continue to use Maximize on Play when possible.

Now do this all again for the other paddle in the following steps.

Step 18: Create another C# Script, call it **P2**, and open it.

You'll be back in Visual Studio and see two tabs for the two script files, P1.cs and P2.cs.

Step 19: Select the P1.cs tab, do **Edit – Select All**, then copy it using **Edit – Copy**.

Step 20: Select the P2.cs tab, again select all the code, then do **Edit – Paste**.

Both P1.cs and P2.cs should now contain the same code.

Step 21: Change "w" to "up" and "s" to "down" in P2.cs. Also change P1 to P2 on line 5.

The "up" and "down" refer to the up arrow and down arrow keys on your keyboard. The class name needs to match the file name for this to work. This is why you need to change P1 to P2 on line 5.

Step 22: Save the file in Visual Studio.

Step 23: In Unity, drag **P2** from the Project window on top of **PaddleRight**.

Step 24: Play the game and try out the new controls.

You can now control both paddles.

Step 25: Save and **exit** Unity.

VERSION 0.03: THE BALL

It's time to create a ball to knock around with your paddles. Fortunately, this is really easy to do in Unity.

Step 1: Start up Unity and load your project.

Step 2: Create a **Sphere** in the Hierarchy panel and name it **Ball**.

Step 3: Select **Ball** and change the **Position** to (**0**, **0**, **0**), if necessary.

Step 4: Make it yellow by creating the material **Mat Ball**, making it **yellow**, and dragging it onto the **Ball** game object.

The code for the ball is a little tricky. You'll launch the ball from the middle of the playfield in a somewhat random direction.

Step 5: Create a new C# Script, call it BallScript, and assign it to the Ball object. Then type in the following code:

```
void Start()
{
    Rigidbody rigidb = GetComponent<Rigidbody>();
    if (rigidb)
    {
        rigidb.freezeRotation = true;
    }
    StartCoroutine("Waitforit");
}
// Update is called once per frame
void Update()
{

}
```

```
IEnumerator Waitforit()
{
   Rigidbody rigidb = GetComponent<Rigidbody>();
   yield return new WaitForSeconds(3);
   if(rigidb)
   {
       rigidb.AddForce(Random.Range(6, 8), Random.Range(-4, -3), 0);
   }
}
```

This code needs some explanation. The idea is to freeze the rotation of the ball in the `Start` function to make the physics behave the way you want. Then you wait for 3 seconds, followed by launching the ball in a randomized direction. You can look at the Unity documentation for more details on Coroutines. To adjust the launch direction vector, you can experiment with the numbers in the `Random.Range` function calls.

If you were to try and run the code right now, the ball wouldn't launch because the ball doesn't have a Rigidbody component yet. Here's how to do that:

Step 6: Select **Ball** in the Hierarchy. Click on **Component – Physics – Rigidbody**.

Step 7: In the Inspector **uncheck Use Gravity** and set the **Mass** to **0.01**.

It's critical that you enter the mass correctly, or the ball will behave strangely. For example, with a mass of 0.1, the ball would move much too slowly in response to the `AddForce` function call.

Step 8: Create a Bouncy Physic Material in the Asset panel just as you did in Chapter 2. Both Frictions are set to 0 and the Bounciness is 1.

Step 9: In the **Sphere Collider** of Ball, click on the small circle next to the Material box. A new window will pop up. Assign the **Bouncy** material to the Sphere Collider. This makes the ball bouncy.

Step 10: Drag the **Bouncy** material onto **both paddles** and **both boundaries**.

Step 11: Test your game!

You should be able to play the game now, with the ball bouncing back and forth. You might try this with a friend. If you're on your own, you can use your left hand on the w and s keys with the right hand on the arrow keys.

Step 12: Save and **Exit**

You have reached a major milestone. The game is now playable! There's still quite a bit of work left to do, but you've made a good start. The ball is bouncing off the walls and the paddles as long as you keep the ball in play. You do have a problem in that if the ball gets by one of the players, you have to restart the game if you want to play again. You'll fix this in the next section.

VERSION 0.04: A BETTER PLAYFIELD

You've reached your first major milestone, but there are still missing elements. You also have some problems with the game. There are two separate philosophies on how to proceed in such a situation. Do you fix what you have, or do you add more features and fix the problems later?

It's usually best to fix your problems early. This has the main advantage that it's easier to fix problems while your project is still small. It's just a better feeling to have a working game rather than a broken game. This also allows you to do more early testing. A large, broken game is difficult or impossible to test.

So, rather than adding scoring or audio, you're going to first fix this problem of the ball flying off into space when a player misses it.

Step 1: Start up Unity and load the **ClassicPaddleGame** project. Make sure you're still using the 2 by 3 layout.

When you change layouts, the view in the Scene panel might get changed as well. If necessary, reset the view to Back Perspective and focus on the Playfield.

You're now going to create an empty object and manually add a box collider to it.

Step 2: GameObject – Create Empty.

Step 3: Rename to **BoundaryLeft**.

Step 4: Move **BoundaryLeft** to position (**-15**, **0**, **0**).

Step 5: Component – Physics – Box Collider.

Step 6: Change the **Size** of the **Box Collider** to (**1**, **35**, **1**).

You now see a green outline of a skinny vertical box on the left border of the Playfield in the Scene panel. You didn't change the Y Scale in the Transform section, although that would have had the same effect as scaling the Box collider. The height of 35 was chosen to extend the box somewhat beyond the upper and lower boundaries.

Step 7: Check the **Is Trigger** box in the Inspector.

You'll see the effect of the trigger checkbox later, when you code your collision script. You now have a box much like the upper and lower boundaries on the left side of the playfield, except that it's invisible! You'll be using this invisible box as a way to detect when the ball is out of bounds.

Step 8: Create a **C# Script**, call it **BallRelaunch**. Then insert the following code after the `Update` function:

```
private void OnTriggerEnter(Collider other)
{
    other.transform.position = new Vector3(0, 0, 0);
}
```

This code is a private function of the class `BallRelaunch`. In order to reset the position of the colliding ball you created a new 3D vector with coordinates set to 0, then you assigned the position of the `other` object to this new vector.

Step 9: Save the code and drag the **BallRelaunch** script to the **BoundaryLeft** Object.

Step 10: Duplicate BoundaryLeft, rename it **BoundaryRight**, and move it to (**15**, **0**, **0**).

Now, if you test the game (and you should), you'll see that the ball gets magically transported to the middle of the screen whenever it gets by one of the players.

Step 11: Save and **Exit** Unity.

Feel free to keep the Visual Studio application running or not, but make sure that you don't have any unsaved editing left there.

You've just made the game quite a bit better but you're still missing a couple of major features: audio and scoring.

VERSION 0.05: AUDIO

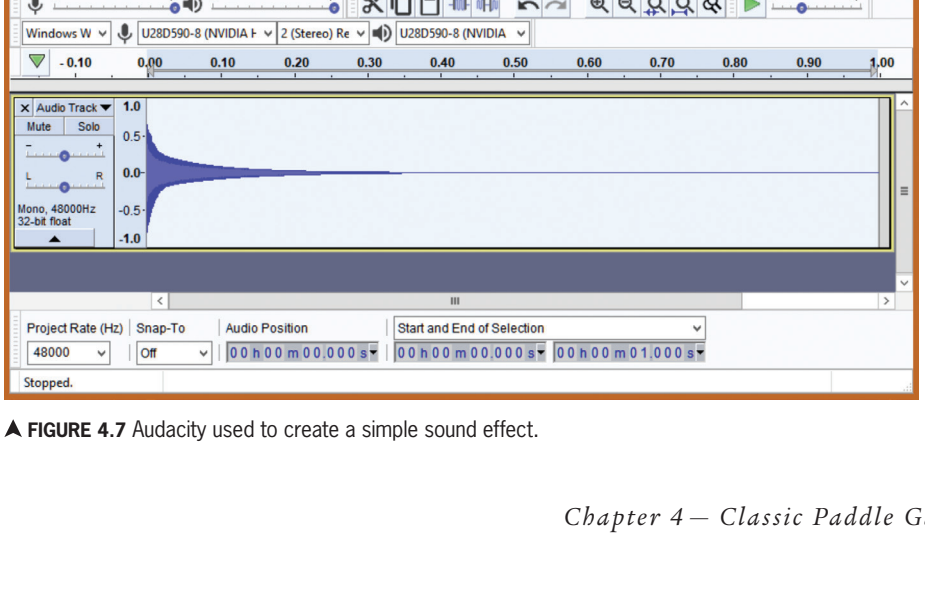
In this section, you'll add a simple sound effect to your game using Audacity.

Step 1: Open Audacity.

Step 2: Select **Generate – Pluck…** from the drop-down menu.

If there are two Plucks in this menu, choose the first one.

Step 3: Select a **Pluck MIDI pitch** of **80, Fade-out type Abrupt, Duration 1 second**, and click on **OK**.



▲ **FIGURE 4.7** Audacity used to create a simple sound effect.

Compare your Audacity window with Figure 4.7. Your blue waveform should look the same and the duration should be one second. Other panels and icons may be different depending on your computer.

If you play this sound, you'll hear that it can work as a collision sound in your game, which is what you want.

Step 4: Select **File – Export – Export as WAV**, give the file the name **pluck.wav**, and save it to the Assets folder of the ClassicPaddleGame Unity project.

Just as you did earlier in the book, the metadata can be safely left blank. One of the convenient features of Unity is that by saving your externally generated files into the Assets folder, they automatically get imported into Unity. Try out this feature right now.

Step 5: Exit Audacity and don't bother saving.

When you exit Audacity, it asks if you want to save. You may do that, if you wish. This will generate an .aup file which you can later load to get back the current state of Audacity. You likely won't need to do that for this very simple sound effect, so you can skip the save step.

Step 6: Open the Unity project and look for pluck.wav in the Assets panel.

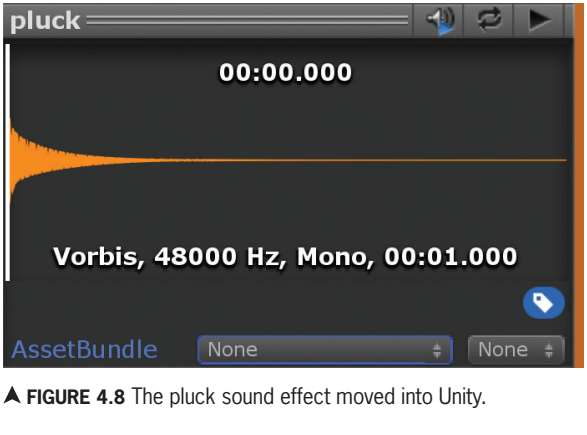
If everything went according to plan, the pluck audio file should be in the Assets panel, ready for inspection. You may need to scroll down in the Assets panel to see the pluck.wav file near the bottom.

Step 7: Select the **pluck** asset, then click on pluck at the very bottom of the Inspector panel.

This will display a small preview pane as shown in Figure 4.8.

You can now test the sound effect in Unity by clicking on the play triangle above in the upper right corner of the preview pane for pluck. That was pretty easy so far. Of course, you could have used any other short .wav file instead of pluck.wav.

Your next goal is to have your sound effect play when the ball collides with something. This takes a few steps:



▲ **FIGURE 4.8** The pluck sound effect moved into Unity.

Step 8: Select **Ball** in the Hierarchy. Then do **Component – Audio – Audio Source**.

This makes the ball a source of audio by adding an Audio Source component to it.

Step 9: Drag the **pluck** sound from the Assets Panel on top of the **Ball** object.

The pluck sound appears as the AudioClip in the Inspector panel. Next, you need to change the code for the BallScript.

Step 10: Add the following line of code at the beginning of **BallScript**. Insert after the three "using" statements at the top of the file as follows:

```
[RequireComponent(typeof(AudioSource))]
```

This statement tells the Unity system that the object associated with this script must have an AudioSource component. You just added this component two steps back, so no problem. This step isn't really necessary, but it help diagnose potential errors if you mistakenly associate this script with an object without an AudioSource component.

Step 11: Add the following function at the end of **BallScript**:

```
private void OnCollisionEnter(Collision collision)
{
    AudioSource audio = GetComponent<AudioSource>();
    audio.Play();
}
```

Step 12: Save the BallScript.cs file in Visual Studio.

Step 13: Save your work, then **test**.

During testing, you discover that the sound plays for no apparent reason on startup. To fix this unwanted behavior, do the following:

Step 14: Select the **Ball** object in the Hierarchy panel and **uncheck Play On Awake** in the Inspector.

Step 15: Save, Test, and **Exit**.

Audio for the older classic arcade games is famous for being extremely primitive. The basic formula for sound design was to have a few simple sound effects when something collided with something else. Recorded music and speech didn't become common until later. In this book, you'll stick with very simple sound effects in keeping with the spirit of early classic gaming.

In the next section, you'll finally add scoring to your game.

VERSION 0.06: SCORING

Your paddle game isn't really a game unless you add scoring. In the classic era, all games had numbers as scores. As games migrated to home systems, numerical scoring became less important, so it was either made irrelevant or dropped altogether, like in many of today's first person shooters.

Step 1: Start Unity and load ClassicPaddleGame.

Step 2: GameObject – Create Empty, name it **Score**.

Step 3: Add a new script component to Score with the name **Scoring** and enter the following code for the Scoring class after the three using declarations:

```
public class Scoring : MonoBehaviour
{
    public static int scorep1;
    public static int scorep2;
```

```
    // Use this for initialization
    void Start()
    {
        scorep1 = 0;
        scorep2 = 0;
    }

    // Update is called once per frame
    void Update()
    {

    }

    private void OnGUI()
    {
        GUI.Box(new Rect(10, 10, 200, 30), "Player 1 Score:    " +
                                                          scorep1);
        GUI.Box(new Rect(Screen.width - 250, 10, 200, 30),
            "Player 2 Score:     " + scorep2);
    }
}
```

A few words of explanation are in order. "scorep1" and "scorep2" are integer variables that store the score for the two players. Our "Start" function automatically gets called at the beginning of the game, so that's a good place to initialize the scores to zero.

The OnGUI function is similar to the code you used for the HelloWorld project. This is where you display the two scores plus labels.

If you now run the game, you'll see that scores are always 0! To make them update according to the gameplay, you also need to add some code to do this. Change the "BallRelaunch" script as follows:

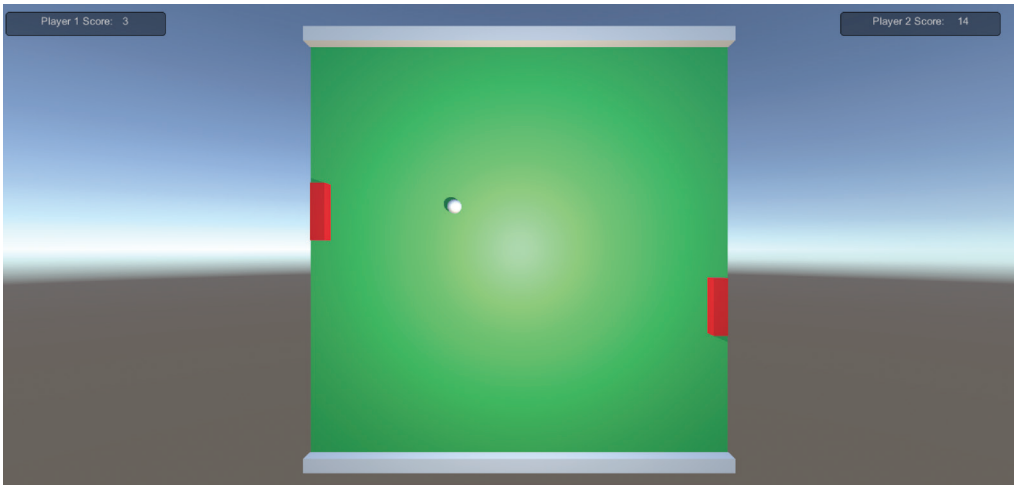
Step 4: Edit the **OnTriggerEnter** function in **BallRelaunch** to look like this:

```
private void OnTriggerEnter(Collider other)
{
    if (other.transform.position.x > 0)
        Scoring.scorep1++;
    else
        Scoring.scorep2++;

    other.transform.position = new Vector3(0, 0, 0);
}
```

The "++" operator in C# increments the preceding variable by 1. This code runs every time the box collider on the left and right boundaries collides with something, presumably the ball. The code then tests the x-coordinate of the ball. If it's greater than 0, it's on the right side of the screen, which means that the ball was missed by player 2. With this somewhat convoluted logic we conclude that in this case Player 1 gets a point, else Player 2 gets the point.

Step 5: Play the game to test it out. **Stop** running it, **Save**, then **exit Unity**.

Figure 4.9 shows the game in action.



▲ **FIGURE 4.9** Gameplay Screenshot of Classic Paddle Game.

VERSION 1.0: FIRST RELEASE!

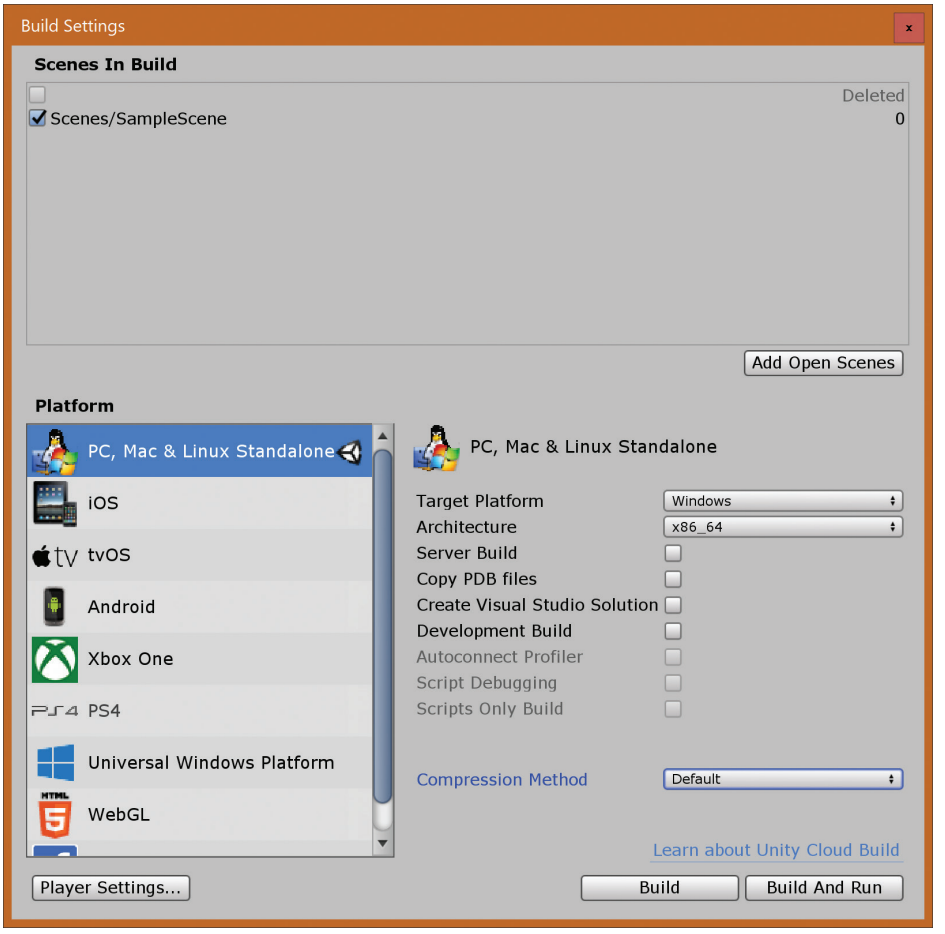
Is this game ready for release? Probably not, but you're going to release it anyway! It's an old joke among developers to say "ship it" right after fixing a bug or adding a feature. For you, in this book, *release* means something a little different. It merely means that you're done with the main development of the game and ready to move on to other things. Of course, if your game is fun and people like it, then the first release is just the beginning of development.

The classic arcade video games that you're studying in this book were developed in an environment where games were tested extensively before release. This testing would happen in focus groups and field tests. For this first game in your book you're going to be lazy with the testing because this game wasn't really intended to be a real product. It's basically a prototype and an exercise to get you started. Your testing consists of making sure the game is playable and runs as expected when you build it. In subsequent games, you'll be tougher during testing.

So how do you release this game? This is really easy in Unity. You simply select **File – Build Settings...** and you will see a new Window as shown in Figure 4.10.

Keep the PC and Mac & Linux Standalone Platform, click on Add Open Scenes to add the current scene, click on Build and Run, and select a folder. It's best to create a new, empty folder for this. You will then have to wait a while for the game to build, though this should take less than a minute, depending on the speed of your computer.

You'll then be able to try out the game in various environments, different graphics settings, and different window sizes. It might also be interesting to take the resulting game and move it to a different computer to see if or how it runs. You'll need to copy the entire build directory, not just the .exe file. On a Mac it works to just copy the .app file. When you run the game with the Very Low graphics quality the lighting of the Playfield is flat, but the game still plays. Apparently point lights don't work in this setting. This is all part of testing the release. You need to test what the game looks like and plays like in different environments. So far you can conclude that your game is still playable even in the lowest quality setting.



▲ **FIGURE 4.10** Building the game.

It's a good idea to test your game on several different computers before releasing it to a large audience. You'd be amazed how frequently additional problems surface during this process. This would also a good time to show the game to some fresh players and to get their feedback.

POSTMORTEM

A *postmortem* is a medical term having to do with studying a medical case after the patient has died. Literally, it is Latin for "after death." This word is also used

in the game development community to take a look back at a project after it's been released and to try to learn what went wrong and what went right. You're now going to do that for your first classic game project.

Here's what went right. You designed, built, and tested a prototype for a very simple paddle game. It works. It compiles and doesn't crash. It looks way better than the original paddle games from the seventies, but that's not really saying much. All you had to do is add color and use a 3D engine. The game feels pretty good and it took very little code.

What went wrong? Well the obvious problem is that you're not really done yet. There's no game over, no title screen, and there's no "Player x wins" message. This is somewhat deliberate. After all, this is just a prototype. You do have a worse problem, though: the physics aren't quite right. When the ball hits the paddle, the player doesn't really feel like he can control where he wants the ball to go. In real table tennis you can control how fast and where the ball goes. You'd like to have that in this game. Oh well, that's what sequels are for.

EXERCISES

There's always more that can be done to a game. Here are a few exercises for readers who would like to reinforce what they just learned in this chapter.

1. Adjust the speed of the ball to make it faster, thus making the game more difficult. Hint: adjust the mass.
2. Adjust the speed of the paddles to make them slower. Experiment with slower and faster speeds. What is the effect of a very fast paddle speed? A very slow speed?
3. Add two more paddles to make it a four-player game.
4. Create a new and different sound effect using Audacity and use it in the game.
5. Add a circular obstacle in the middle of the playfield. Try adding several obstacles. Is the game better or worse when you do that? Explain.

6. Implement two ball speeds. Have the speed depend on a button press by one of the players.

   Which player should control the ball speed? Why?

Advanced Exercises for experienced Unity users:

7. Make it a one-player game by adding AI to player 2.
8. After completing the previous exercise, add a menu to select one-player or two-player.
9. Add graphics for the net in the middle by adding a texture to the Playfield object. Create the texture using GIMP.
10. Make the score display look more like the original Pong with large segmented digits.

11*. This one is tough, good luck! Build the project again, from scratch, without looking at the book. Change some things along the way, just for fun. For example, make the ball bigger, change the colors, change the lighting, make the paddles a different shape. Save your work often and create different save files as you progress.

CHAPTER

5 Breakout

IN THIS CHAPTER

Breakout is the first successful single-player arcade video game. It's a good example of a simple game with the kind of addictive quality that foreshadowed the golden age of arcade gaming in the early '80s. With *Breakout*, it's just you against the machine, an experience similar to golf or bowling, where you try to outdo your own past efforts.

WOZ

Steve Wozniak, aka Woz, built a working prototype of this game in four days, together with Steve Jobs. Jobs was a brand new technician at Atari and had been assigned the task of making this game. He immediately enlisted his friend and hardware guru, Woz. Getting little sleep, the duo worked nonstop to pull off this stunning feat. Nowadays, games such as *Breakout* can be implemented in software in just a few hours, but this was 1976. In order to keep costs reasonable, the game needed to be built using custom hardware.

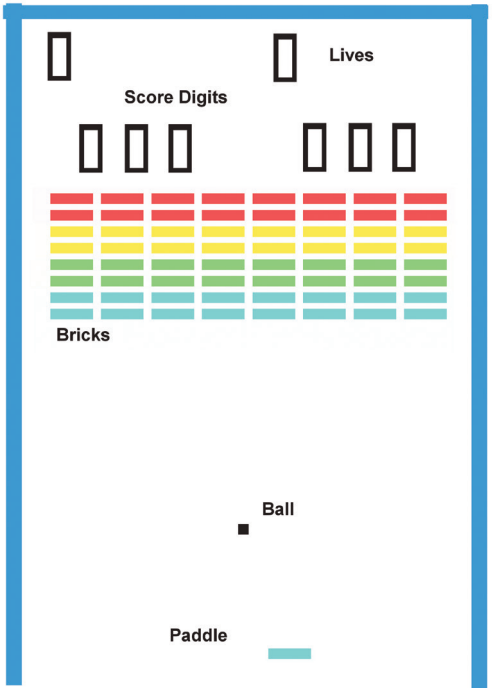
This is one of the earliest anecdotes of people working crazy overtime hours to make a video game. Things haven't changed, and it's still very common for game developers to work late into the night. On the other hand, there are plenty of successful and even famous designers who work normal hours and have a life.

BREAKOUT, ATARI (1976)

Just like *Pong*, *Breakout* is a ball and paddle game, but with the goal of breaking the bricks in a wall. Every time the ball hits a brick it would magically disappear, and

you would score points. You would start with three balls and you'd lose a ball every time you'd miss hitting it with the paddle. *Breakout* gets more difficult as the game progresses by speeding up the ball and making the paddle smaller. You can see a game design diagram of the original coin-op *Breakout* in Figure 5.1.

FIGURE 5.1 *Breakout*, Atari, 1976.



The text in the diagram isn't in the original. The colors were changed to make the diagram easier to see in print. The original background color was black and the digits were white. The number of bricks in the original *Breakout* coin-op version was 14 columns across and 8 rows high.

The details of the difficulty progression are interesting. The paddle gets smaller when the ball hits the top boundary for the first time. The ball speeds up after it hits the paddle 4 times and again after it hits the paddle 16 times. Also, the angle of the ball path changes after it hits the paddle 4, 8, 12, and 16 times.

The reason those numbers are powers of two, or related to powers of two (i.e., 12 = 8 + 4) has to do with the efficiency and cost of the hardware, not just because the game designers liked powers of two.

The scoring is fairly simple, rewarding the player with 1, 3, or 5 points depending on the color of the brick. In the original arcade game, the color was faked by putting a colored overlay on top of the black-and-white monitor.

The difficulty resets every time the player loses a ball, giving a moment of relief to the player. On the other hand, there's a slow difficulty progression happening in the background because every time a brick gets destroyed, the difficulty increases

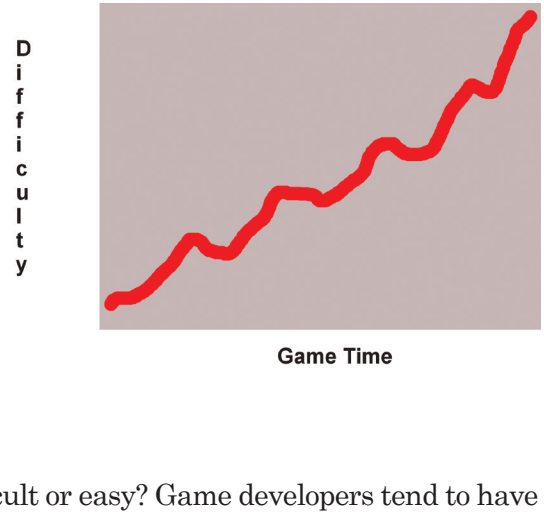
slightly. When there are fewer bricks out there, it's more likely that the ball will hit the back border, which dramatically increases the difficulty due to the much smaller paddle.

Difficulty ramping is a central concept of classic game design and deserves its own design rule:

Classic Game Design Rule 3: Difficulty Ramping Rule: Ramp difficulty from easy to hard.

Almost all games need to deal with the issue of how fast to ramp difficulty, if at all. *Pong* had no ramping, at least not explicitly. In multiplayer games, the ramping usually happens by having your opponents getting better with practice. Successful single-player games almost always ramp difficulty at a steady pace, with periods of relief thrown in to reward the players. Look at Figure 5.2 for a way to visualize this.

▼ **FIGURE 5.2** Typical Difficulty Ramping in Single-Player Games.



How are you to know if a game is difficult or easy? Game developers tend to have tunnel vision when it comes to their own games. They may think it's easy when it's actually hard, especially for novices. The next rule:

Classic Game Rule 4: Test Rule: Test the game to make sure it's fun.

It's just that easy. You need to test the game. In the days of custom hardware, testing was an expensive proposition. Now that video games are developed using

incredibly powerful software tools, it's much easier and cheaper to test. You need to test early and often. People have careers consisting of testing video games. Mostly, the career video game testers look for bugs and try to figure out how to duplicate them. But even more important than bug testing is testing for fun and suitability. Is the game too easy or too hard? Or even both? These questions can only be answered by extensive testing, preferably by representatives of the target customers.

It's critically important to test new video games first on yourself, but then with children, expert players, casual players, players of all sizes, ages, and abilities. In the old days of arcade game development, this was a common practice. Arcade game companies put prototype games into arcades and street locations. They carefully measured how many quarters each game earned when compared to the other popular arcade games. If a game didn't have top earnings, the project would be cancelled.

In these days of internet distribution, testing is very easy, but it still takes some effort. Just release the game as a beta, or in a limited geographic region, and gather statistics on how long people play the games, where they have trouble, and how far they get into the game. It's also a good idea to talk to the players.

BREAKOUT SEQUELS

It's no surprise that *Breakout* inspired sequels, including *Super Breakout*™, Atari (1978) and *Arkanoid*™, Taito (1986).

In the sequels, the basic control stays the same. You still try to break the bricks, and if you break all of them, you move on to the next screen.

In *Super Breakout*, you have multiple balls and a progressive mode where the bricks shift down the screen at you.

In *Arkanoid*, various powerups are introduced which make the game much more interesting. For example, you get a capture powerup which, when enabled, lets you catch the ball and have it stick to the paddle. When you're ready, you can then release the ball with the launch button.

In 2017, the major app stores started to carry dozens of brick and ball games. If you search the internet for "bricks balls game" videos, you'll get a quick sense of what's out there. These games can definitely be viewed as *Breakout* sequels.

WHERE ARE THEY NOW?

Right after their *Breakout* adventure, Steve Jobs and Steve Wozniak founded Apple Computer, Inc. Steve Jobs was largely responsible for growing Apple Computer into one of the most valuable companies on the planet. Apple Computer dropped the "Computer" in its name and is called just Apple, Inc. as of 2007. Woz continues to be an iconic presence in Silicon Valley.

CHAPTER

6 Classic Brick Game

PADDLE GAME FOR ONE

This is going to be your first one-player game, building upon what you learned in the classic paddle game project. You will be starting "from scratch," even though it's tempting to reuse the framework from your first project. The two projects are different enough that it's better to just start over. This is a good lesson, by the way. When in doubt, just make a fresh start. Often the baggage from an old project is more of a hindrance than a help.

This game is a combination of *Pong* and pinball. There will be a paddle at the bottom of the screen and a wall of bricks. There's the familiar bouncing ball, and the goal is to keep the ball bouncing, much like in a pinball game. You start with 3 balls, and if you lose all of them, it's game over. In a way, this is a very simple pinball game without the gravity.

This project has another big difference with the paddle project. You'll be using "fake" physics rather than the physics engine built into Unity. It's good to remember that all the classic games of the '70s and '80s didn't use physics engines, but instead used custom code to move and animate game objects. This was due to the very limited computer resources available at the time. It wasn't until the '90s that floating point computations were commonly used in games, and even then, there was a large speed cost associated with them. Today, floating point operations are about the same speed as their integer counterparts, so the world has really changed in this regard. In the classic era, integers were king, and the whole idea of using a floating point physics engine was a distant dream.

In your project, you will use the classic technique of updating your object positions using explicit code, and you'll do collision reaction explicitly as well. The object positions will be represented using floating point numbers, just because it's easier to do it that way in Unity.

VERSION 0.01: THE PLAYFIELD

Once again, your first goal is to design and display the playfield. This is an easy step for you now because you just built something similar in the previous project.

Step 1: Run Unity and create a new **3D** project with the name **ClassicBrickGame**.

Step 2: Use the **2 by 3** layout.

Step 2: GameObject – 3D Object – Cube, rename it **Playfield**.

Step 3: Select the **Playfield**, hover the mouse over the **Scene Panel**, then type **f**.

You should now see a cube in the Scene panel. This is your starting point for the playfield. Next, you'll make the playfield larger:

Step 4: Change the **Scale** of **Playfield** to (**30**, **30**, **1**). The **Position** should already be (**0**, **0**, **0**).

Step 5: Use the Scene Gizmo to select the **Front iso** view and refocus with the **f** key.

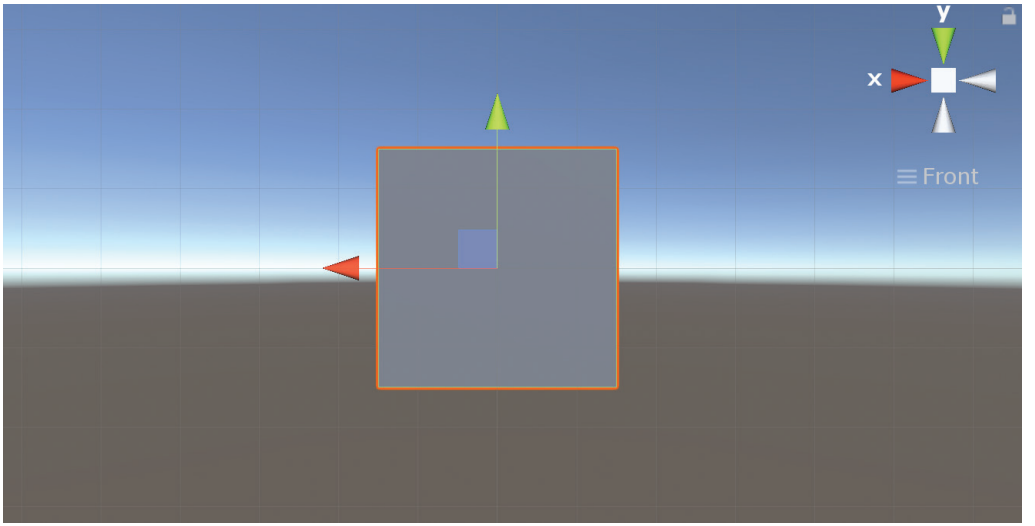
Right-click on the Scene Gizmo and uncheck perspective, if necessary, to enable the isometric view. The Scene panel should now look like Figure 6.1.

Next, you'll change the color of the playfield. Just as in ClassicPaddleGame, you'll do this by creating a material, assigning it to the object, and adjusting the color of the material.

Step 6: Click on **Create** in the Project panel, click on **Material**, and immediately type the new name for it: **Mat Playfield**.

The text entry is highlighted in blue to tell you that you can type a new material name, if you wish.

Step 7: In the Inspector panel change **Albedo** to a **dark green color**.



▲ **FIGURE 6.1** The Playfield rescaled.

Step 8: Drag the **Mat Playfield** material onto the **Playfield** game object.

The Playfield game object now appears to be dark green in the Scene panel. The Game panel is a lighter shade of green.

Step 9: Move **Main Camera** to (**0**, **0**, **-30**).

The Game panel now shows the entire playfield.

Step 10: Delete the **Directional Light**. In the Hierarchy Create a **Point Light** at (**0**, **0**, **-10**) and change its **Range** to **100** in the Inspector panel.

Just as in the paddle game, this is a good time to do your first test.

Step 11: Select **Maximize on Play** in the Game panel. Then hit the **Play** arrow.

Just as in the Paddle game there's no animation yet, just a still view of the Playfield. You might have noticed that these initial steps are almost exactly the same as in the Paddle game.

Step 12: Stop play mode by clicking on the **Play** arrow again.

Step 13: Save.

Saving here isn't truly necessary, but it's a good habit to save your work at a good stopping point. Your next goal is to create the boundaries of the playfield. For this game, the boundaries are at the top, left, and right, with an open area at the bottom.

Step 14: Create a **Cube** with **Position** (**0**, **15**, **0**) and **Scale** (**30**, **1**, **1**).

Step 15: Rename it **BoundaryUpper**.

The Scene panel is still using the Top view, which isn't what you want any more.

Step 16: Select the **Back Isometric** view in the Scene panel using the Scene Gizmo.

To get the Isometric view, uncheck Perspective in the Scene Gizmo menu. Alternatively, you can click on the label below the Gizmo to switch between Isometric and Perspective views.

Step 17: Select **Playfield**, hover the mouse in the Scene panel, and press **f**. Then **zoom in** on the playfield with your **scroll wheel**.

It's a little strange that you're looking at the Playfield from the back, but that's what you need to do because there are negative Z coordinates for the Point Light and the Main Camera.

Step 18: Enter a new **Z Position** of **1.1** instead of 0. Note that the position is 1.1 instead of 1, just as you did in the previous project.

Step 19: Right-click on **BoundaryUpper**, click on **Duplicate**, and rename the duplicate to **BoundaryLeft**.

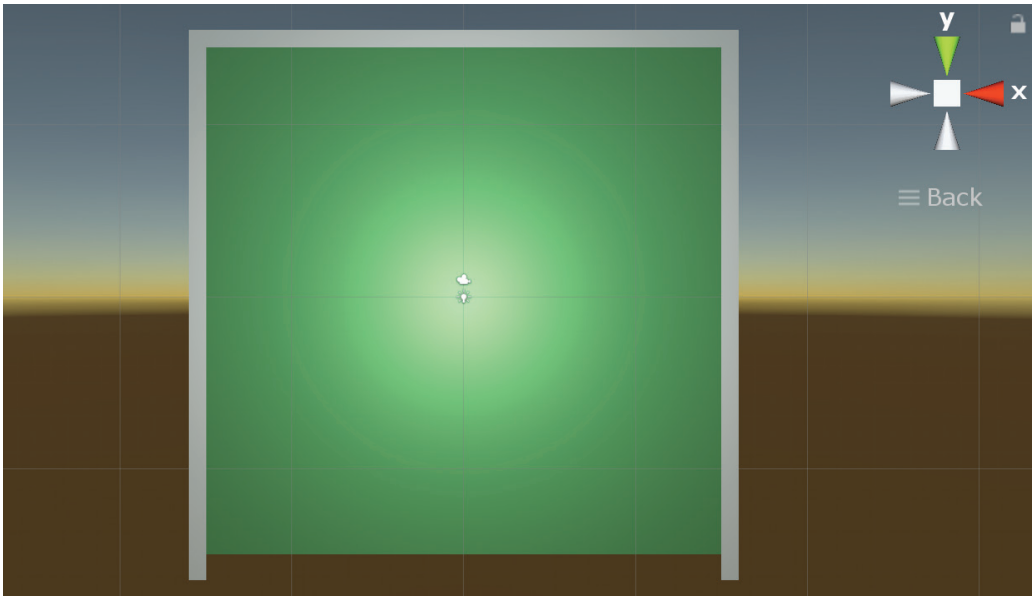
Step 20: Set **BoundaryLeft Position** to (**-15.5**, **-0.5**, **0**) and **Scale** to (**1**, **32**, **1**).

Step 21: Duplicate **BoundaryLeft**, rename the duplicate to **BoundaryRight** and change the **X Position** to **15.5**.

Compare your Scene panel to Figure 6.2.

Step 22: Save.

In the next section, you'll add the player character, which for this game is a paddle at the bottom of the playfield.



▲ **FIGURE 6.2** The playfield for the Brick game.

VERSION 0.02: THE PLAYER

Just as in the ClassicPaddleGame project, the player character is a paddle, but this time it moves left to right, and there's only one of them.

Step 1: If necessary, start up Unity and load the project.

Step 2: Create a **Cube** and name it **Paddle** at **Position** (**0**, **-15**, **0**), **Scale** (**4**, **1**, **1**).

The origin of the coordinate system is at the center of the playfield, so to move the paddle to the bottom requires a negative Y Position.

Step 3: Create a **Material** in the Project panel, rename it **Mat Paddle**, make it **red**.

Step 4: Drag **Mat Paddle** on top of **Paddle**.

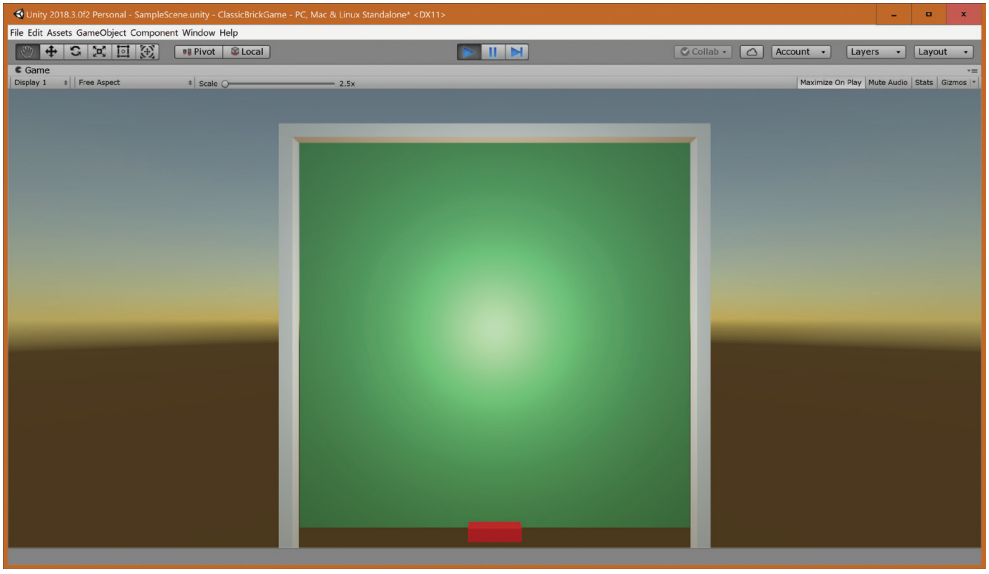
The paddle should now be red instead of grey. You're ready to create a script for the paddle so it's controlled by the player.

Step 5: Select **Paddle** and create a C# Script for it using **Add Component** with name **PlayerScript**. Then type in the following Update function:

```
// Update is called once per frame
void Update()
{
    if (Input.GetKey("left"))
    {
        transform.Translate(-20 * Time.deltaTime, 0, 0);
    }
    if (Input.GetKey("right"))
    {
        transform.Translate(20 * Time.deltaTime, 0, 0);
    }
}
```

This `Update` function is pretty much the same as in ClassicPaddleGame except it moves the paddle from left to right instead of up and down. It does this by changing the x coordinate in the Translate calls instead of the y coordinate.

Step 6: Save the file in Visual Studio and run the game. Test the left and right arrow keys. When you're done testing, click on the play arrow again to stop play mode.



▲ **FIGURE 6.3** The red paddle controlled by arrow keys.

The arrow keys should move the paddle left and right and your Game panel should look similar to Figure 6.3.

Controlling the paddle using arrow keys is not nearly as much fun as using the mouse, so you'll add that feature. The arrow controls can stay, just because they don't do any harm and you never know, maybe there's players out there who would prefer the arrow keys.

Step 7: Insert the following two lines at the end of the Update function:

```
float h;


h = 30.0f * Time.deltaTime * Input.GetAxis("Mouse X");
transform.Translate(h,0,0);
```

The h variable is an abbreviation for horizontal offset. It is calculated by taking the output of the `Input.GetAxis` call and multiplying it by a time factor and a speed factor of 30. The `f` after the `30.0` is necessary for floating point constants in C#. This code could have been squeezed into a single statement by inserting that long expression for h into the Translate call. You avoided that in order to make the code clearer.

The mouse pointer shouldn't really be on the screen when the game is getting played like this, so here is a one-line fix.

Step 8: Insert the following code into the Start function:

```
Cursor.visible = false;
```

The mouse cursor can be turned back on by the player with the "Esc" key. It's a bit strange to allow both arrow and mouse controls at the same time, but it really doesn't matter.

Step 9: Test, Save, and **exit Unity**.

VERSION 0.03: BASIC BALL MOVEMENT

The ball in this game is pretty much the same as in the Paddle game.

Step 1: Start up Unity and load your project, if necessary.

Step 2: Create a **Sphere** in the Hierarchy window, rename to **Ball** with Position (**0**, **-7**, **0**).

This is the initial ball position. It's a little lower on the screen to make room for the bricks.

Step 3: Make the Ball **yellow** using a new material called **Mat Ball**.

As before, you create the material "Mat Ball," make the material yellow, and drag it on top of the Ball.

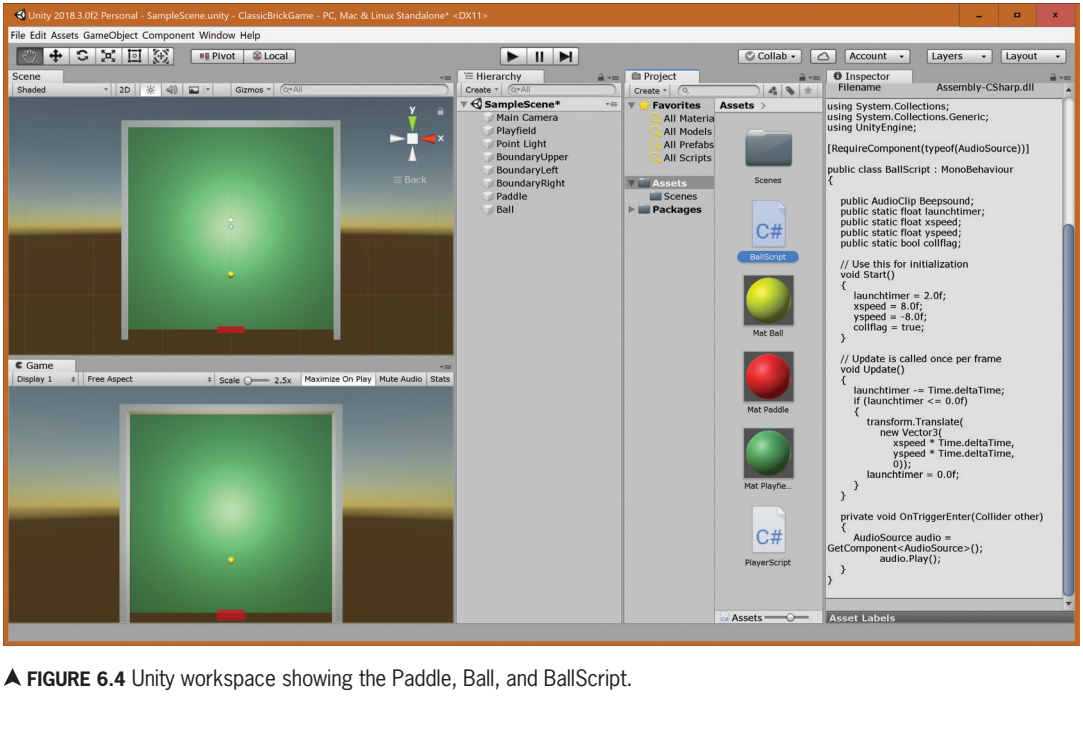
Step 4: Select **Ball**, do **Add Component** with name **BallScript**, and enter the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(AudioSource))]

public class BallScript : MonoBehaviour
{
    public AudioClip Beepsound;

    public static float launchtimer;
    public static float xspeed;
    public static float yspeed;
    public static bool collflag;

    // Use this for initialization
    void Start()
    {
```

```
        launchtimer = 2.0f;
        xspeed = 8.0f;
        yspeed = 8.0f;
        collflag = true;
    }

    // Update is called once per frame
    void Update()
    {
        launchtimer -= Time.deltaTime;
        if (launchtimer <= 0.0f)
        {
            transform.Translate(
                new Vector3(
                    xspeed * Time.deltaTime,
                    yspeed * Time.deltaTime,
                    0));
            launchtimer = 0.0f;
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        AudioSource audio = GetComponent<AudioSource>();
        audio.Play();
    }
}
```

Your screen should look like Figure 6.4. In that figure the Inspector panel was scrolled down to show the text for BallScript.

The `Time.deltaTime` variable is a built-in Unity variable that returns the amount of time, in seconds, since the last time the Update function was called. For more information on this and many other Unity features, do Help – Scripting Reference and do a search.



▲ **FIGURE 6.4** Unity workspace showing the Paddle, Ball, and BallScript.

Step 5: Save your work and start testing.

The ball should be stationary for two seconds and then move up towards the upper right and off the screen without bouncing. The audio isn't working yet.

This is a great example of incremental development. You eventually want the ball to bounce off the boundaries, but first you just want it to sit there and then move along the specified velocity vector with the components xspeed and yspeed.

This is about as much code as you should ever write all at once without testing. There are software developers out there who spend days, weeks, or even months writing thousands of lines of code without testing any of them. Then they start testing. Odds are very high that they will have countless bugs. Needless to say, that is a horrible situation. How can you find, fix, or even test such a mess of buggy code? It's much better to write a little, test a lot, fix, and repeat.

While you were at it, you added audio code. This is an example of what not to do, but people often do this anyway. It would have been cleaner and better to keep

it simple and to not yet add dead code (code that's not used right now). Even though you added some code for audio, the audio isn't working yet, which is what you would expect because you don't even have your audio asset yet, nor is it connected to the Ball object.

Next, you'll get the audio working by reusing the pluck.wav file from the Assets folder of the previous project.

Step 6: Assets – Import New Asset… and use the **pluck.wav** file from the previous project. Test the asset by previewing it.

Step 7: Assign the **pluck** sound to the **Ball** object by dragging it.

Step 8: Test.

You should hear the pluck sound at the beginning, but then never again.

Step 9: Uncheck Play on Awake.

Step 10: Save and **Exit**.

Because you unchecked Play on Awake, there's no audio at all in the game now. That's because there are no collisions triggering it. You'll need to remember to test the audio when you add collisions in the next section. You might have been better off adding the audio code after implementing collisions.

VERSION 0.04: COLLISIONS

To do collision with the playfield, you'll start by writing a short script for the right and left boundaries.

Step 1: Start Unity and **load** the project.

Step 2: Create a **C# Script** and call it **WallScript**. Insert the following code:

```
private void OnTriggerEnter(Collider other)
{
    BallScript.xspeed = -BallScript.xspeed;
    BallScript.collflag = true;
}
```

Step 3: Drag it on top of **BoundaryLeft** and **BoundaryRight**.

This isn't quite it yet. It's easy to forget to set the triggers and rigidbody setting.

Step 4: Select the **Ball**, and select **Component – Physics – Rigidbody**.

Step 5: Uncheck Use Gravity.

The physics engine supports gravity by default, but in this game there's no gravity.

Step 6: Select **BoundaryRight** and **check** the **Is Trigger** box in the Box Collider in the Inspector panel. Do the same for **BoundaryLeft**.

Now you should be able to run the game and have the ball bounce off the right wall. Also, you should hear the Pluck sound when that happens. Of course, because you haven't put in the collision code for the upper boundary, the ball will behave strangely when hitting it. The next steps add proper collision for the upper boundary.

Step 7a: Create a **C# Script** and name it **WallTopScript**.

Step 7b: Open it and **copy** the code from WallScript.cs into it.

One fast way to do this is to open WallScript.cs in another tab, Edit – Select All, Edit – Copy, select the WallTopScript tab, Edit – Select All, and Edit – Paste. It's even faster if you use the keyboard shortcuts.

Step 8a: Replace both instances of `xspeed` with `yspeed` in the `OnTriggerEnter` function. Your code should look like this:

```
private void OnTriggerEnter(Collider other)
{
    BallScript.yspeed = -BallScript.yspeed;
    BallScript.collflag = true;
}
```

You're not done yet!

Step 8b: If the class name in WallTopScript.cs is WallScript, make the class name WallTopScript!

This step is necessary if you actually copied the entire contents of the WallScript.cs file, rather than just the OnTriggerEnter function. The class name must match the name of the associated file.

Step 9: Save WallTopScript in Visual Studio and assign the script to **BoundaryUpper**.

Step 10: Select **BoundaryUpper** and **check Is Trigger** in the Box Collider component.

When the game is played now, the ball should bounce off the right, top, and left boundary, and then fall through the bottom. Also, it acts weirdly when it hits the paddle.

There are two more collision cases to deal with, ball vs. lower boundary and ball vs. paddle. First, you'll create the lower boundary as an invisible barrier.

Step 11: Select **GameObject – Create Empty**, rename it **BoundaryLower**. Move it to **Position** (**0**, **-17**, **0**) with **Scale** (**35**, **1**, **1**).

Step 12: Select **Component – Physics – Box Collider** and **check** the **Is Trigger** box in the Inspector.

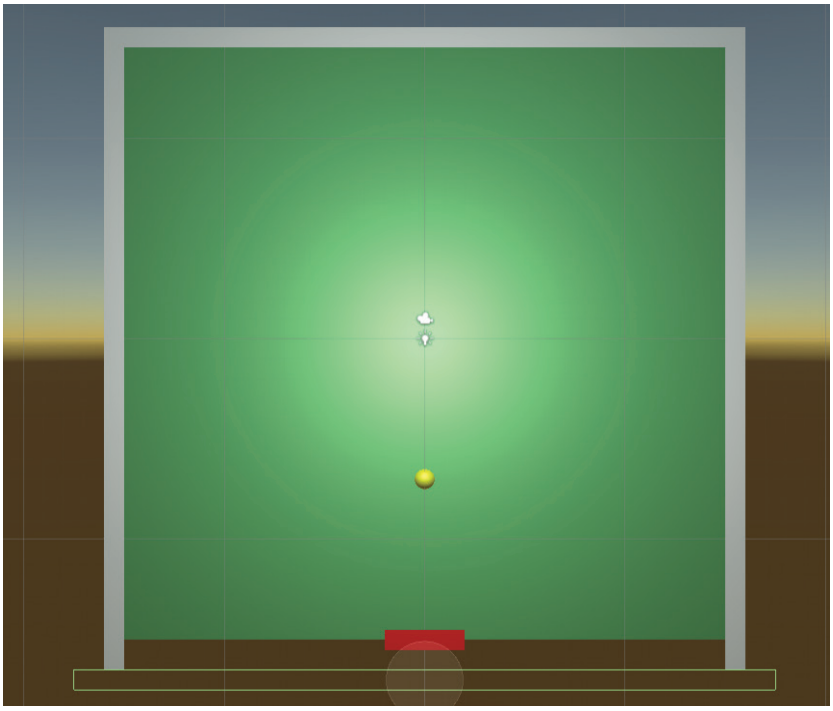
This invisible box is a way to detect when the ball has escaped from the playfield at the bottom.

The Scene panel should now look like Figure 6.5.

Notice that BoundaryLower is visible in the Scene panel but not in the Game panel, which is exactly how you want it. Next, you need to write a script that handles what to do when the ball hits that lower boundary.

Step 13: Create a **C# Script**, rename it to **BallRelaunch**. Then open the script and enter the following code:

```
private void OnTriggerEnter(Collider other)
{
    other.transform.position = new Vector3(0, -7, 0);
    BallScript.xspeed = 8.0f;
```



▲ **FIGURE 6.5** Scene Panel showing BoundaryLower.

```
        BallScript.yspeed = -8.0f;
        BallScript.launchtimer = 1.0f;
    }
```

Step 14: Assign **BallRelaunch** to **BoundaryLower**.

This script deserves some explanation. The variable `other` is the object that collides with our lower boundary. This code magically repositions that object, presumably the ball, to its starting position and resets the speed. It also resets the `BallScript.launchtimer` variable to one second so that the player has a little bit of time to get ready for more action. You can and should test this right now, or you can wait until after the next two steps.

To add collision with the paddle, first make the Paddle object a trigger:

Step 15: Select **Paddle** and **check** the **Is Trigger** checkbox in the Box Collider.

Step 16: Enter the following function at the bottom of **PlayerScript**:

```
private void OnTriggerEnter(Collider other)
{
    BallScript.yspeed = -BallScript.yspeed;
    BallScript.collflag = true;
}
```

This happens to be the exact same code you just entered into WallTopScript.cs. Go ahead and try it. You now have a bare bones brick game without the bricks. The ball bounces the way it's supposed to, and the player character works. You even have rudimentary sound.

There's one serious flaw in your current code, and it won't really become apparent until later. You don't have any way of controlling what the ball does when it hits the paddle. In real table tennis, you would have smashes, strange spin shots, and of course, you'd have some way of aiming where the ball goes.

There are countless ways to implement ball control, but the simple way in the original arcade *Breakout* is a good starting point: If the ball hits the left side of the paddle, it bounces to the left, and if it hits the right side, it bounces to the right. The following modified trigger code does that.

Step 17: In **PlayerScript** modify the **OnTriggerEnter** function as follows:

```
private void OnTriggerEnter(Collider other)
   {
      BallScript.yspeed = -BallScript.yspeed;

      if (other.transform.position.x >
          gameObject.transform.position.x)
      {
          BallScript.xspeed = Mathf.Abs(BallScript.xspeed);
      }
      else
```

```
        {
            BallScript.xspeed = -Mathf.Abs(BallScript.xspeed);
        }
        BallScript.collflag = true;
    }
```

The `Mathf.Abs` function is the absolute value function that returns the positive value of a number. The code checks to see if the ball is on the right side of the paddle. If it's on the right, then the x component of the ball velocity is set to be positive, otherwise it's set to be negative. In either case, the y component of the ball velocity is reversed.

The variable `other.transform.position.x` looks complicated. You read it from back to front like this: the x-coordinate of the position of the transform of the other object. That's a long way of saying the x coordinate of the colliding object, which happens to be the ball.

This is a great example of the kind of "fake" physics that is prevalent in classic games. Real physics requires compute power that wasn't yet feasible at the time. Much of the development time and effort was spent on technical issues such as this.

Step 18: Test it, **save** it, **exit** Unity.

It's time to add some bricks. After all, this is a brick game.

VERSION 0.05: BRICKS

You're ready for a more advanced programming technique. How can you add an array of bricks to your playfield? It's tempting to create them the same way as you've been creating all of your objects, but this would be very time consuming. Instead, you'll be creating a "BrickMaker" object and create your bricks using a double loop in the associated script.

First, you'll create the display of the bricks. Then, you'll add collision handling so that the bricks actually disappear when they get hit by the ball.

Step 1: Start up Unity and **load** the project.

Step 2: Click on **GameObject – Create Empty** with name **BrickMaker**.

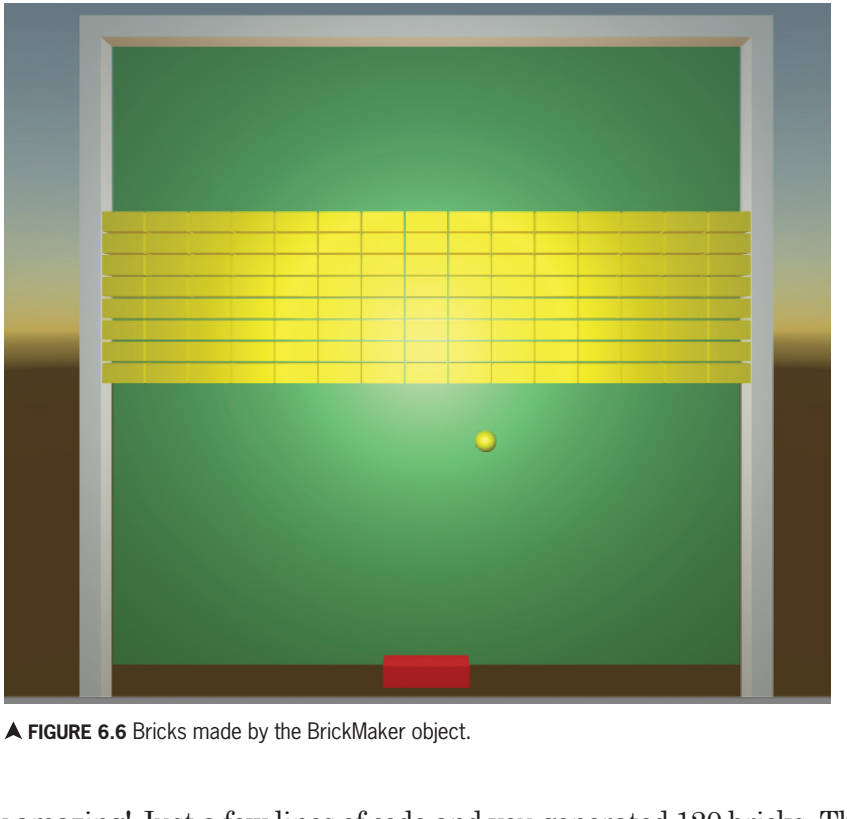
Step 3: Create **MakeBricksScript**, assign it to **BrickMaker**, and enter:

```
void Start()
{
    for (int y = 0; y < 8; y++)
        for (int x = 0; x < 15; x++)
        {
            GameObject cube =
                GameObject.CreatePrimitive(PrimitiveType.Cube);
            cube.transform.position =
                new Vector3(x * 2 - 14, y - 1, 0);
            cube.transform.localScale =
                new Vector3(1.9f, 0.9f, 1);

//          cube.AddComponent<BrickScript>();

            Material m_material =
                cube.GetComponent<Renderer>().material;
            m_material.color = Color.yellow;

            cube.GetComponent<Collider>().isTrigger = true;
        }
}
```

That's quite a bit of new code all at once! You'll run this code first, and then carefully read through it and try to understand it.

Step 4: Run the game.

The Game panel now looks like Figure 6.6.



▲ **FIGURE 6.6** Bricks made by the BrickMaker object.

Pretty amazing! Just a few lines of code and you generated 120 bricks. They don't do anything yet, but you can see them, and that's a good start. Let's go through the code together and try to understand what it does.

The two `for` statements set up the 2-dimensional array of bricks. The `y` variable traditionally keeps track of vertical position. Here you have 8 rows of bricks and the `y` variable ranges from row 0 to row 7 (which adds up to 8 rows). Programmers like to count starting at zero because that usually makes the geometrical formulas simpler and thus, more efficient. The `x` variable keeps track of the 15 horizontal brick locations, ranging from column 0 to column 14.

Inside of your double loop you create a brick using the `CreatePrimitive` statement. Then you compute the position and scale of each brick. The position depends on the `x` and `y` variables. The scale is set to make your bricks a width of slightly less than 2 and height slightly less than 1. The depth is 1 as always for all of your objects in this game.

The two slashes on the next line indicate a comment. This means that the line doesn't get used, but is just there for future reference. You'll be "uncommenting" this line later on by simply deleting the slashes. The following two lines set the brick color to yellow. The last line in the loop turns on the "Is Trigger" property for the bricks.

You're now going to change Point Light to a directional light in order to get flat lighting. This is a cosmetic style choice and doesn't affect the gameplay.

Step 5: Delete the **Point Light**, **create** a **Directional Light**, **rename** it to **Light**, and in the inspector change the **Intensity** to **0.63**. Verify that the **Rotation** is (**50**, **-30**, **0**).

You now have a much more cartoon-like look. You can have a more dramatic color change from row to row by setting different colors for different rows of bricks.

Step 6: Replace the m_material.color statement with the following code:

```
if (y < 2)
   m_material.color = Color.yellow;
else if (y < 4)
   m_material.color = Color.cyan;
else if (y < 6)
   m_material.color = Color.blue;
else if (y < 8)
   m_material.color = Color.red;
```

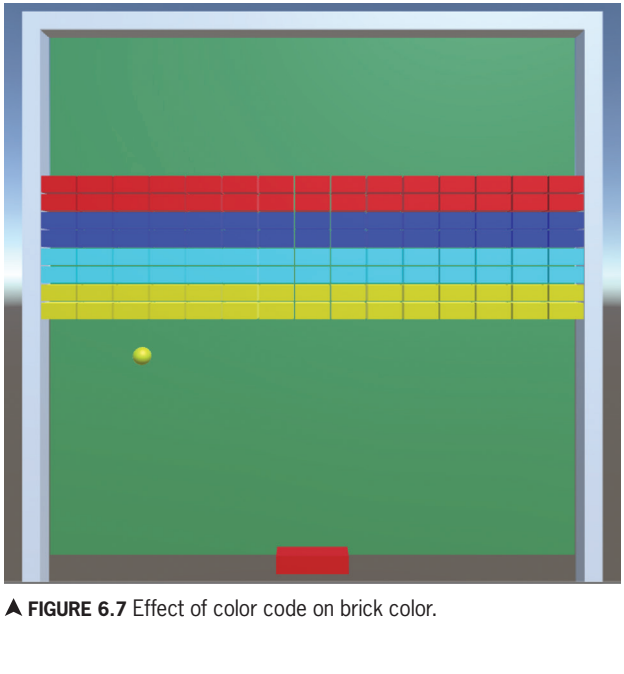
When you run the game now it should look like Figure 6.7.

You used some of the built-in colors of Unity. You could also explicitly set the RGB values of each color using the Color function.

Step 7: Save your work, **exit** Unity.

In this section, you saw the amazing power of programming. Rather than creating all those bricks manually, you wrote just a few lines of code that do the same thing. You were also able to set various properties of the bricks in code.

In the next section, you'll make the game playable.



▲ **FIGURE 6.7** Effect of color code on brick color.

VERSION 0.06: FIRST PLAYABLE

It's time to make those bricks disappear when the ball hits them.

Step 1: Start up Unity and **load** the project.

In the MakeBrickScript, there's a green line. It's commented out because it would cause an error otherwise. Each of those bricks will have a script, so first, let's write it.

Step 2: Create a **C# Script** and call it **BrickScript**. Enter the following code:

```
private void OnTriggerEnter(Collider other)
{
    if (BallScript.collflag == true)
    {
        BallScript.yspeed = -BallScript.yspeed;
        BallScript.collflag = false;
        Destroy(gameObject);
    }
}
```

Step 3: Delete the **slashes** at the beginning of the commented line in **MakeBricksScript**.

Step 4: Save the changes for both files in Visual Studio and try running the game.

Magically, the game is now playable.

The BallScript.collflag variable is a `bool` variable which is `true` when you want collisions to be active and `false` when you don't. The idea is that after the ball hits the first brick you don't want the ball colliding with other bricks but to instead have it immediately bounce back to the player or a wall. This is a somewhat weird piece of logic, but it works and it's simple. The original *Breakout* used similar logic, but the sequels went for something more realistic.

Once the script has determined that the collision should be done, it flips the y component of the ball velocity, turns off the collision flag, and finally destroys the brick that called it.

You can now better understand the reason behind the collflag statements in WallScript and WallTopScript. Those statements turn collisions on again when a ball hits a wall, thus allowing the ball to bounce back and forth between walls and bricks.

Step 5: Save and **quit**.

You now have a playable prototype. You even have some sound, just because it was easy to put in. If you're not hearing any sound, turn on your computer speakers, turn up the volume, and verify that you're getting the pluck sound every time there's a collision between the ball and something else.

In the next section, you'll add scoring to your game because a game without scoring isn't much of a game.

VERSION 0.07: SCORING

In this section, you'll create the score display, design the scoring rules, and finally implement them in your code. In general, it's good practice to make your games playable first. Only then does it make sense to add the scoring code.

Step 1: Start Unity and **load** the Project.

Step 2: Create a Score object by selecting **GameObject – Create Empty**, name it **Score**.

Step 3: Create a new **C# Script**, call it **Scoring**, and enter the following code:

```
public class Scoring : MonoBehaviour
{

    public static int score;
    public static int lives;

    // Use this for initialization
    void Start()
    {
        score = 0;
        lives = 3;
    }

    // Update is called once per frame
    void Update()
    {

    }

    private void OnGUI()
    {
        GUI.Box(new Rect(10, 10, 90, 30),
          "Score:   " + score);
        GUI.Box(new Rect(Screen.width - 100, 10, 90, 30),
          "Lives:   " + lives);
    }
}
```

Step 4: Assign the **Scoring** script to the **Score** object as usual by dragging.

To make the scores update, you now need to find a place in your code where the bricks get destroyed. You just wrote that code in the previous section.

Step 5: Edit **BrickScript** by inserting a single new line of code as follows:

```
private void OnTriggerEnter(Collider other)
{
   if (BallScript.collflag == true)
   {
      BallScript.yspeed = -BallScript.yspeed;
      BallScript.collflag = false;
      Destroy(gameObject);
      Scoring.score += 10;
   }
}
```

The "+=" command in C# adds the following number to the preceding variable. In this instance, 10 gets added to the score. You just added simple scoring but now it's time to update the lives counter. This is also very easy.

Step 6: Insert the following line of code at the beginning of the OnTriggerEnter function in the **BallRelaunch** script:

```
Scoring.lives--;
```

The two minus signs are the decrement operator in C#. This has the effect of decreasing the number of lives by 1.

Step 7: Save your work, **test** it, and **exit** Unity.

You should see the lives display in the upper-right corner of the game count down when you lose a ball off the bottom of the screen.

That's a good start, but there's a problem. You don't have a game over screen! Instead the lives counter goes negative. That's definitely a bug. The next section fixes this and along the way introduces the concept of multiple Scenes in Unity.

VERSION 0.08: TITLE SCREEN

In this section, you'll build a very simple title screen. It'll be a single image with the instruction to press a key to start the game. Pressing any key will go to the game. When the game is over, this title screen will appear again. That's about as simple as it gets, but because you're building everything from the ground up, it still takes some careful work to have this happen.

Step 1: Run GIMP and create a new image with dimensions 256 x 240 pixels.

Any dimension will do, but these dimensions were selected as a reminder of this common and very low resolution that was used by raster arcade games of the early '80s.

Step 2: Add the text "Brick Game, press any key" using the Text tool in GIMP.

It doesn't really matter how you do this as long as the text is legible.

Step 3: Export the image into the Assets folder of your Unity game project. The image should be in .png format and have the name **BrickTitleImage**.

Step 4: Exit GIMP.

The exported .png file is what will be used by Unity. You may wish to save to the GIMP native .xcf file format too, if you wish, but it's not necessary. It could be useful to have this .xcf file available as a starting point in a future GIMP session, but .xcf files aren't used by Unity, so you don't need to create it.

Step 5: Open Unity and load the **ClassicBrickGame** project.

Notice that BrickTitleImage automatically shows up in the Assets panel.

Step 6: Do **File – New Scene**, then right-click on Untitled in the Hierarchy and save the scene as **BrickTitleScene**. **Move** this new scene into the **Scenes** folder.

Step 7: Create a new **C# Script** with name **MainTitle**. Enter the code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
public class MainTitle : MonoBehaviour
{

    public Texture backGroundTexture;

    private void OnGUI()
    {
        GUI.DrawTexture(
            new Rect(
                0,
                0,
                Screen.width,
                Screen.height),
            backGroundTexture);

        if (Input.anyKeyDown)
        {
            Debug.Log("A key or mouse click has been detected");
            SceneManager.LoadScene("BrickScene");
        }
    }
}
```

Notice that there is a fourth `using` statement at the top of this file. This is necessary for the `LoadScene` statement. The Debug.Log statement sends a message to the Console. This is a useful method of keeping track of what's happening during testing.

Step 8: Assign the **MainTitle** script to the **Main Camera** game object.

Step 9: Select **Main Camera** and drag **BrickTitleImage** to the slot next to **Background Texture** in the inspector.

Running the game now will display the image, but when you press a key you will get an error because you haven't yet included both scenes in the build settings for the project. You also need to rename SampleScene with BrickScene.

Step 10: Save BrickTitleScene.

Step 11: Select **File – Build Settings**... and add the current scene to the build settings by clicking on the **Add Open Scenes** button.

Step 12: Go to **Assets** in the Project panel, double-click on the **Scenes** directory, right-click on **SampleScene**, and **rename** it to **BrickScene**.

Step 13: Add this scene to the Build Settings just like Step 11.

You'll now have two scenes in the current build settings for this project.

Step 15: Double-Click on **BrickTitleScene** in the Assets panel to select it. **Run** the game and press any key after the title screen appears.

You're now ready to go back to solving the problem that started all this. What should happen when the player runs out of lives? You just go back to the title screen.

Step 16: Insert this code into the BallRelaunch script:

```
if (Scoring.lives == 0)
   {
      SceneManager.LoadScene("BrickTitleScene");
   }
```

You'll also need to add

```
using UnityEngine.SceneManagement;
```

at the top of the file where you see all those "`using`" statements;

Notice the two equal signs next to one another. That's not a typo. You need both of those equal signs. Unlike in mathematics, in C# and most other modern programming languages, one equal sign is used for assignment, but two are needed when testing for equality.

Step 17: Test it by **losing on purpose** and then **playing** a **second game**.

Well, guess what, there's a bug! By the way, the automatic answer when someone says this is "Just one?" The nature of programming, and especially game programming, is that there're going to be bugs. A lot of bugs. The best defense against having buggy code is to test frequently and to fix all known bugs as soon as possible.

The bug is this: When you start a game, the ball gets launched up instead of down and bounces off of a brick. This isn't really a big deal. It could even be called a feature, but you should fix it, because it's easy to do.

Step 18: Make sure you've exited play mode.

Step 19: In the `Start` function of BallScript, change the initialization for yspeed to -8.0.

Step 20: Test, **save**, and **exit**.

This game is in pretty good shape now, so now you can release it.

VERSION 1.0: FIRST RELEASE AND POSTMORTEM

There's a relatively new adage in the game business: release early and release often. So, you're releasing this game even though it's still very basic. The procedure for releasing this game in Unity is pretty much the same as for the Classic Paddle Game, so the instructions won't be repeated here.

Releasing early and often wasn't really an option in the classic era. Arcade games had to be manufactured, sent to the distributors, and then to the arcades. Sending software updates to all those arcade machines was possible, but expensive. A lot of effort went into making the games bug-free and fun before the first release. Home games were typically manufactured as ROM cartridges. In that case the penalty for having a bug was extremely large. Hundreds of hours of testing were needed before the first release. Even so, disastrous releases did occur when products were rushed to market without sufficient testing.

It's time for a quick postmortem of the Brick game. Here's what went right. You made a simple brick game and it works. It sure looks a lot like the many similar brick games from the '70s and '80s, though the graphics are much better with the nice 3D effect of the bricks. The best part is that you learned quite a bit about how to make this kind of game in Unity.

What went wrong? Well, the game isn't very fun yet. There's no difficulty ramping and the game is just too simple, even by the standards of 1976. Nevertheless, the following exercises will help with this.

EXERCISES

1. Adjust the speed of the ball to make it faster, thus making the game more difficult. Put in a counter that increases the speed of the ball after the ball has had 8 collisions.
2. Adjust the numbers in the mouse control code and see what happens. What happens to playability if the mouse control is too sensitive or not sensitive enough?
3. Add a second paddle just above the first paddle and have the mouse control it simultaneously with the first. Try using different mouse sensitivities with the two paddles.
4. Create several different sound effects in Audacity and have different sounds for different types of collisions. If you did Exercise 1, increase the pitch of the collisions when the ball speed increases.
5. The title screen text is fuzzy. Experiment with the texture import settings in Unity to improve this. Hint: try the different filter settings.

Advanced Exercises for experienced Unity users:

6. Make it a two-player game by adding a second paddle of a different color and have the second paddle be controlled by a different set of keys.
7. Create a texture in GIMP and use it in the playfield.

CHAPTER

7 Space Invaders

IN THIS CHAPTER

Space Invaders is the first mega-successful video game, surpassing everything before it exponentially. The success is a result of great design that made the game much deeper than anything before. It was the first arcade video game that celebrated the skill of the players, a game where millions of players would play every day to get better and better, get higher scores in the process, and feel a sense of accomplishment much like in golf, bowling, or pinball.

HUGE MONEY, HUGE

In terms of money, *Space Invaders* broke new ground. By some estimates, *Space Invaders* grossed a coin drop of two billion dollars by 1982, making it the highest-grossing entertainment product of all time. It is also the top-selling arcade game of all time, having sold 300,000 units in Japan alone and being responsible for a shortage of 100-yen coins.

It's easy to forget the huge impact of this game. It showed that the public was willing to spend serious cash on video games.

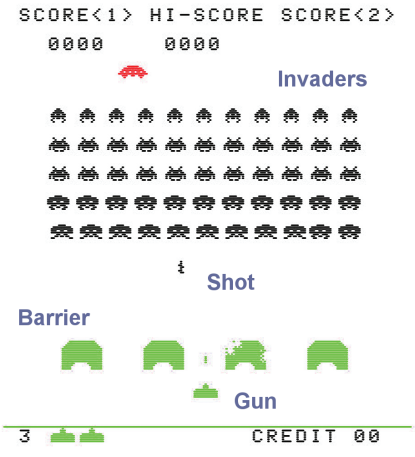
THE DESIGN OF *SPACE INVADERS*, TAITO (1978)

Space Invaders is the first vertical shooter. You control a laser gun with a two buttons and you shoot at a horde of alien attackers by pressing the fire button. To give the player a better chance to survive early on, there are four destructible barriers near the bottom of the screen, as depicted in Figure 7.1.

This game is still fun to play over forty years later. It introduced the basic gameplay formula of having three lives and getting an extra life after reaching a point goal. Countless games after it imitated the style of having a character move side to side and shoot up at an onslaught of enemies. Even today's FPS extravaganzas can be viewed as fancy sequels to *Space Invaders*.

This game was also the first major video game to get people to *really* care about their score.

FIGURE 7.1 *Space Invaders* Game Design Diagram.



SCORE EQUALS SKILL

Classic Game Design Rule 5: Score Rule: Score equals skill.

In *Space Invaders*, having a score higher than your friends was meaningful, just like in pinball. It also introduced the High Score display at the top of the screen. Furthermore, *Space Invaders* didn't forget about the expert players and made the score mean something even for the elite players. *Breakout* has no meaningful world record because, well, if you're an advanced player then it's no problem for you to get the maximum possible score of 896. On the other hand, *Space Invaders* requires expert skill to get a world-class score.

There are at least three major ways that Rule 5 is violated by designers: capping the score, allowing marathoning, and allowing score milking. Here's a look at all of these in turn.

Capping the score means there is an easily achievable and known maximum score for the game, for example, the maximum score of 896 in single-player *Breakout*.

Claiming to have the world record on this game is somewhat misleading because thousands of players have achieved it.

Marathoning is the practice of playing arcade games up to the limits of human endurance. Achieving a large score is more an indication of the ability to keep playing rather than a measure of skill. This happened with major games such as *Asteroids* and *Missile Command* and countless others.

Score milking occurs when players easily build up their score indefinitely without actually playing the game as intended. This was uncommon in arcade games, but it is frequently possible in home games, for example, *Super Mario Brothers*.

One of the great achievements of *Space Invaders* is that the scoring was, for the first time in an arcade game, truly meaningful. Unfortunately, the game doesn't quite live up to that achievement since there are a few experts who are able to marathon the game. Achieving a world record on this game is a combination of endurance and skill. In a world record run, the game is pretty much the same for hour upon hour of gameplay. You can see for yourself by searching the internet for videos of world records.

Now remember Rule 4? How are you going to test your scoring system? Well, for starters, it's important to get yourself an advanced player playing the game for a few weeks to see if he's still breaking his own scores and still has the desire to improve. If you don't have access to a top player or two, make sure that you're ready to respond when your game gets into their hands after you've released your game!

In *Space Invaders*, each wave of enemies would start a little bit lower on the screen compared to the previous wave. This simple device is a great way to ramp up difficulty and is a verification of the Difficulty Ramping Rule. The majority of classic arcade games, starting with *Space Invaders*, would use this device. Compare this to *Breakout* where, if you finished the first wave of bricks, you were handed another wave without any ramping.

GOING STRONG 41 YEARS LATER

Even though the arcade business faded away in the '80s and '90s, *Space Invaders* sequels continue to be sold. The latest version is *Space Invaders Infinity Gene*™ released for IOS in 2009, Xbox Live Arcade and PlayStation Network in 2010, and Android in 2011.

The *Space Invaders* characters are really the first video game characters to achieve iconic stature in popular culture. The aliens have been featured in street art, t-shirts, and even furniture.

Space Invaders changed video games from casual to hardcore, from diversion to hobby. We owe a debt of gratitude to game designer Tomohiro Nishikado and Taito. It's hard to imagine how video games would have evolved without the seminal influence of *Space Invaders*.

CHAPTER 8

Classic Game Project Three: Vertical Shooter

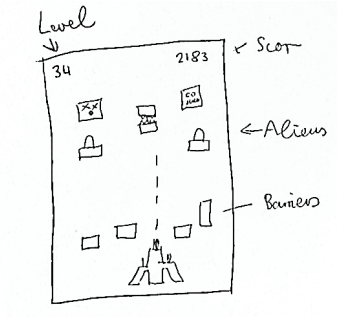
DESIGNING A SHOOTER

The third classic project is going to be quite a bit more ambitious if the size of this chapter is any indication. Your goal is to build a simple vertical shooter, similar to the great many arcade vertical shooters from the '70s and '80s. You're going to start by sketching the basic layout of the game.

The setting is an outer space battle where you control a spaceship near the bottom of the screen and shoot at alien spaceships and creatures that are attacking you. You are seeing the beginnings of some story elements here, but don't be too concerned about telling the story. Classic games are all about the gameplay.

Take a look at Figure 8.1. It's a very rough sketch of the layout. You'll want to display the score and the level, the enemies and the playfield, barriers to block the enemy shots, and, of course, the player character. The sketch doesn't include the scrolling star background.

▼ **FIGURE 8.1** Game sketch of vertical shooter.



In the next section, you'll start by building the playfield.

VERSION 0.01: THE PLAYFIELD

Because the game is set in outer space, you'll choose to display the playfield as black with a background of scrolling stars. There are two approaches to displaying

a star field, both valid: each star can be its own object, or you can simply display an image of the star field created in a paint program or captured with a camera.

FIGURE 8.2 Star field for the classic vertical shooter.

Step 1: Run Unity and create a **new 2D project** with name **ClassicVerticalShooter**.

This project is done in 2D, mainly to try out the very basics of developing a 2D game in Unity. You will draw a star field image in GIMP using the mouse. You may instead use the one provided on the DVD. Figure 8.2 shows what the star field on the DVD looks like.

Step 2: Create the following subfolders of the **Assets** folder: **Sprites**, **Prefabs**, **Scripts**, and **Sounds**.

The Scenes subfolder is already there as a default folder when you create a new project.

Step 3: Create a Star Field using GIMP. Make the size 1024 x 1024 pixels and export it to **starfield.png** in the newly created **Sprites** subfolder.

It's OK to leave Unity open while you do this. It's up to you how you want to draw the starfield. It should be mostly black with some outer space objects on it like stars of varying colors and sizes. The starfield on the DVD was created using a large brush of size 20 and a smaller brush of size 5. The colors are white, light yellow, light red, and light blue.

The size of 1024 by 1024 is no accident. Today's 3D hardware has an easier time displaying textures with dimensions that are powers of two. It probably doesn't matter much in this case, but it's a good habit to use powers of two for image sizes in games and 3D applications. The reason for this has to do with "mipmapping," a technique used by 3D display hardware to efficiently display textured objects that are far away from the camera. Yes, it's true that this project is using the 2D template, but Unity may still use 3D hardware when displaying graphics on 3D platforms.

In this game, the playfield will be scrolling, but first you'll just display it.

Step 4: Select starfield in **Assets/Sprites** panel, then **GameObject – 2D Object – Sprite**, name it **background**.

Step 5: For **background**, set **Position** (**0**, **0**, **1**), **Rotation** (**0**, **0**, **0**), **Scale** (**1**, **1**, **1**).

You are moving the background to a Z position of 1 so it will be behind the other sprites later on.

Step 6: Place **Main Camera** at **Position** (**0**, **0**, **-10**) and **Rotation** (**0**, **0**, **0**). Set the **Projection** to **Orthographic**. Change the **Size** to **3.5**.

The Size controls the zoom factor of the orthographic camera.

Step 7: Use **2 by 3** layout, then **select** and **focus** on the **background** using the **f** key.

Step 8: Rename **ScampleScene** to **mainscene. Save.**

Step 9: Double-click on the Scripts folder to select it and **create** a new **C# Script** called **starfield_scroller**. Then open the script and enter the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class starfield_scroller : MonoBehaviour
{
    public float scrollspeed;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
```

```
    {
        transform.Translate(0,scrollspeed * Time.deltaTime,0);
        if (transform.position.y < -10.0f)
        {
            transform.Translate(0, 20.48f, 0);
        }
    }
}
```

Step 10: Save the script and **assign** it to the **background** object.

Step 11: Select **background** and set **Scrollspeed** to -2 in the Inspector.

Step 12: Test

You'll see the starfield scroll down the screen but there's a gap above it before it reemerges. This is fixed in the next step:

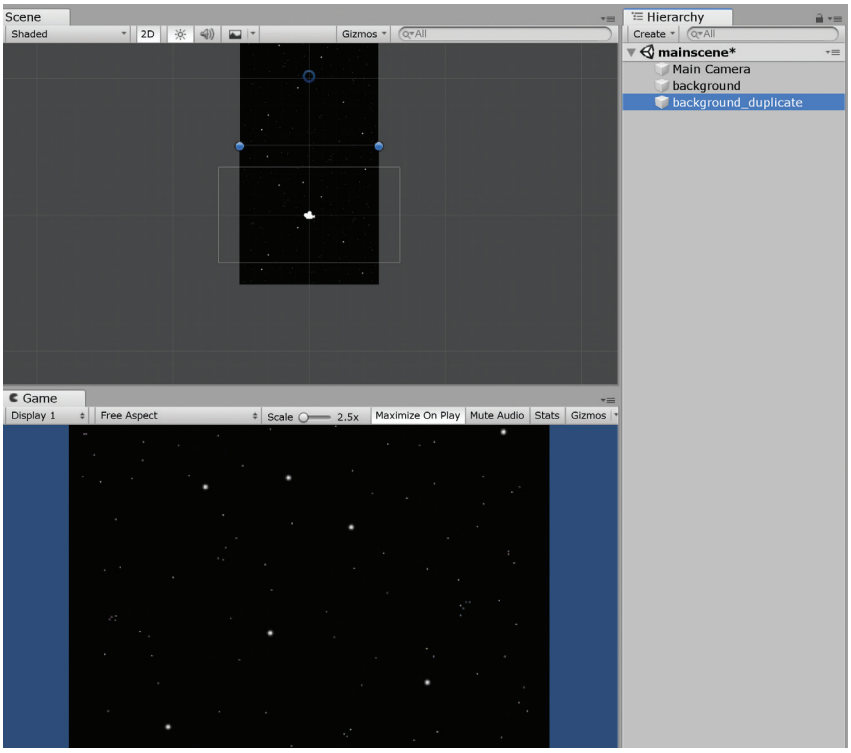
Step 13: Duplicate background, rename to **background_duplicate,** set **Position Y** to **10.24**.

This has the effect of placing a duplicate of the background immediately above. When you run the game now you should get a seamless vertical scroll. Compare your Scene and Game panels to Figure 8.3.

Those "magic numbers" of 10.24 and 20.48 deserve an explanation. 10.24 is the vertical size of the starfield divided by the Pixels per Unit setting in the starfield Import Settings. That number is the vertical size of the background sprite in world units. Of course, 20.48 is twice that. The `starfield_scroller` script moves the starfield up by two starfield heights once the sprite is completely scrolled below camera view. You can watch this in action by running the game and turning off Maximize on Play.

Step 13: Save and **exit** Unity.

In summary, in this section you created a background object and assigned a starfield texture to it. Then you used a clever technique to scroll the background in an endlessly repeating animation.



▲ **FIGURE 8.3** Duplicated background to achieve seamless scrolling.

VERSION 0.02: THE SPACESHIP

Now that you have a playfield, it's time to make your main player character, the spaceship. Here's the plan for this section. You're going to draw a spaceship in GIMP and then display it in Unity. The spaceship is going to be a nonanimated 2D sprite with alpha. Let's first explain what that is.

Nonanimated is simple enough. You just have a single image for your spaceship. There's really no need to animate the spaceship because it's a solid object without moving parts. The word *sprite* just means that the image can be moved on the screen. In the early days of video game development, the hardware commonly supported both sprites and stamps. Stamps, as opposed to sprites, were rectangular chunks of graphics that couldn't be moved relative to each other, though it was typically possible to move all the stamps as a unit.

Alpha is a term used to describe transparency. The spaceship will fit into a 32 x 32 grid. Some of the pixels in the grid will be used to display the spaceship, whereas other pixels will be transparent. The transparent pixels will have an alpha value of 0, the solid pixels 1. It's possible to have an alpha value in between. For example, an alpha of 0.5 would mean that the pixel colors are intended to be combined with the background graphics, giving a translucent effect.

Step 1: Run GIMP.

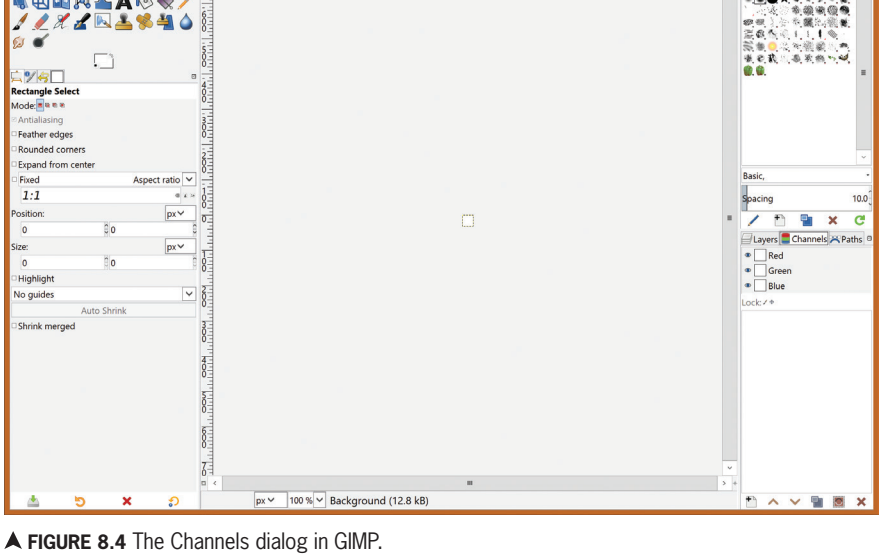
Step 2: Select **Windows** – **Dockable Dialogs** – **Channels**.

Step 3: File – **New** and choose an image size of **32 x 32** pixels.

Your screen should look something like Figure 8.4.

Notice that you have three channels: Red, Green, and Blue. Soon you'll have an Alpha channel as well. The image looks tiny, so zoom in on it so you can better see what's happening.

Step 4: Select **View** – **Zoom** – **8:1**.



▲ **FIGURE 8.4** The Channels dialog in GIMP.

Feel free to zoom in even more. This depends on your screen resolution.

Step 5: Select the **Pencil** tool (the tool that looks like a diagonal pencil).

As the mouse hovers over the tools on the left side of the screen you will see pop-ups that tell you more about each icon.

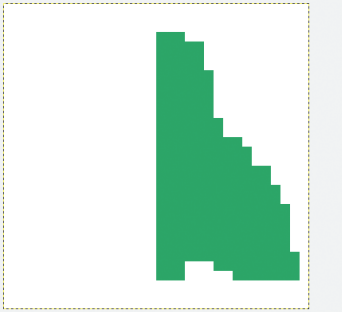
Step 6: In the Tool Options dialog, make the **Size** 1 pixel.

Step 7: Choose a **foreground** color of **dark green**.

This is done on the color selection area at the bottom of the toolbox. The foreground box is the upper-left rectangle. Click on it to get the color selection dialog. Then use the color selection dialog to choose a dark shade of green. Don't make it took dark, so it contrasts with the black starfield.

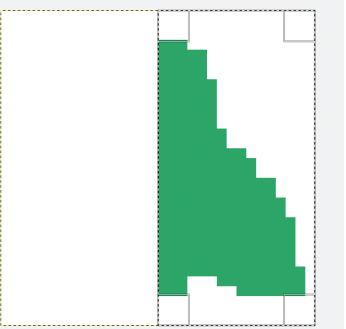
Step 8: Use the mouse and the left mouse button to **draw** a shape like the one shown in Figure 8.5. This is the right half of your spaceship.

FIGURE 8.5 Starting to draw the spaceship in GIMP.



It's not important that your drawing matches the book's version pixel for pixel. Feel free to draw something else that looks similar to a top view of a spaceship. This is just a starting point. The first thing you're going to improve is the symmetry. Your goal is to take the right half, flip it, and copy it on top of the left half of the image. Here's one way to do this.

FIGURE 8.6 Selecting the right half of the spaceship.



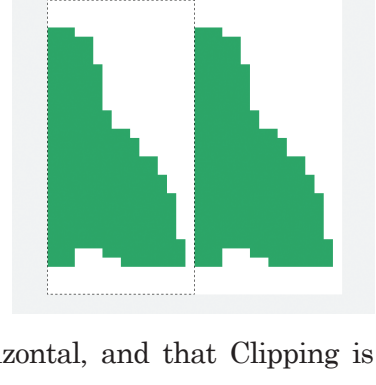
Step 9: Use the **Rectangle Select Tool**, the upper-left tool in the tool box, and select the right half of the image. The size of the selection should be **16 x 32**.

You can watch the bottom of the window to see the size change as you drag the mouse. This looks like Figure 8.6.

Step 10: Edit – Copy. Make sure you're still using the Rectangle Select tool and **select** the **left half** of the image. Then **Edit – Paste As – New Layer**.

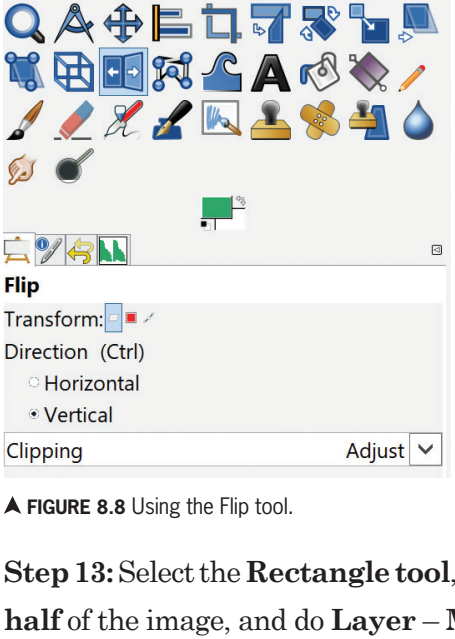
This is a tricky step. If you did it correctly your new image looks like Figure 8.7

▼ **FIGURE 8.7** Duplicating the right half of the spaceship.



You're now ready to flip the left half of the image.

Step 11: Select the **Flip tool** as shown in Figure 8.8. Check that the Transform is set to Layer, that the Direction is Horizontal, and that Clipping is set to Adjust.



▲ **FIGURE 8.8** Using the Flip tool.

Step 12: Click on the **left half** of the image.

This last click flips the left half of the image. This is quite an improvement! Symmetry is the secret to making great-looking pixel art. It often doesn't matter how well you can draw, as long as you incorporate symmetry.

You're not done yet. The left half of the image is still sitting in a separate layer. You need to merge the layers.

Step 13: Select the **Rectangle tool**, click on the **right half** of the image, and do **Layer – Merge Down**.

▼ **FIGURE 8.9** Basic spaceship.

Your image should now look like Figure 8.9.

Next, you'll use the "Cartoon" filter to add a nice black edge to your spaceship.

Step 14: Select **Filters – Artistic – Cartoon**... and use a **Mask radius** of **1.3** and **Percent black** of **0.5**.

Feel free to experiment with these values or even to try some of the other filters. Your result should look something like Figure 8.10.

You're now ready to add alpha to your 2D image. You might have noticed that the Channels dialog now displays an Alpha channel, but it doesn't have any content yet.

Adding alpha to this particular image is really easy because you didn't use white when drawing your spaceship.

Step 15: Select – By Color, set Threshold to **70**, and **click on the white background** area of the image.

You have just selected the white background, plus some light grey pixels near the spaceship.

Step 16: Do **Colors – Color to Alpha** and make sure that the **From** color is **white**, which is the default.

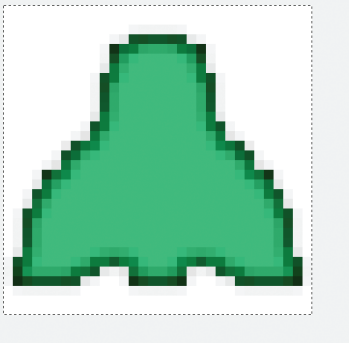
You now have an outline of the spaceship in the Alpha channel in the Channels dialog as shown in Figure 8.11.

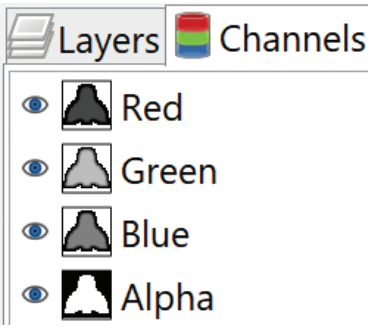
The spaceship itself now has a checkerboard background to indicate transparency. It should look like Figure 8.12.

Next, you'll save your work. There are two files that you want to save, .xcf and .png.

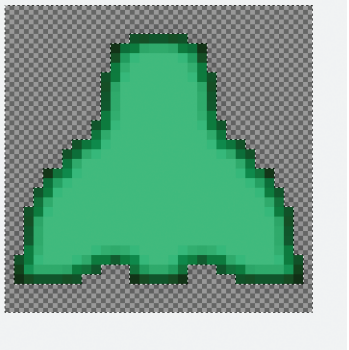
Step 17: Click on **File – Save As** and use the name **ship.xcf** in the directory "**ClassicVerticalShooter/Assets/Sprites**."

▼ **FIGURE 8.10** 2D spaceship, no alpha.





▲ **FIGURE 8.11** RGB plus Alpha channel for the spaceship.



▲ **FIGURE 8.12** Final 2D spaceship with alpha.

Step 18: Do **File** – **Export** to save **ship.png** in the same directory. IMPORTANT: Use **compression level 0**.

This is a very small file so there's no need to compress it. If you use compression with small sprites, you can easily get noticeable artifacts when displaying them in your game later on.

Step 19: Exit GIMP.

Step 20: Run Unity and double-click on the **Assets/Sprites** folder.

Step 21: Click on **ship** in the Assets panel and look at the Inspector panel. Set the **Filter Mode** to **Point (no filter)** and **Wrap Mode** to **Clamp**, then **Apply**.

Step 22: Drag the **ship** asset into the **Hierarchy** panel.

Step 23: Set **Position** to (**0**, **0**, **0**), **Rotation** to (**0**, **0**, **0**).

You can now see the ship in the starfield. To see the ship in the Scene panel, focus on it.

Step 24: Save your work, then **exit** Unity.

In the next section, you'll continue with your development of the sprites in this game.

VERSION 0.03: SPRITES

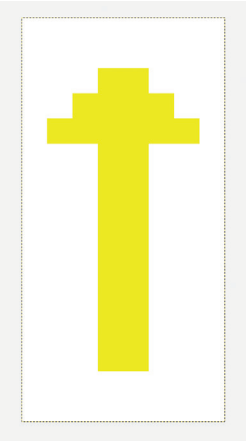
You just made a single 2D sprite for your spaceship, which was a good first step. Next, you'll create additional sprites, the shots, and you'll learn how to dynamically create and destroy them. This is necessary so that when a shot hits something the shot disappears, and when the player hits a "Fire" button a shot is created. You'll start by drawing a shot in GIMP. This is similar to drawing the spaceship.

Step 1: Open GIMP and create a **new image** with a **width** of **8 pixels** and **height** of **16 pixels**.

This is a tiny image, so zoom in on it by repeatedly pressing the "+" key.

Step 2: Select the **pencil tool** and set the **size** to **1**. Zoom in on the image, make the **foreground color yellow** and **draw an arrow** pointing up as displayed in Figure 8.13.

FIGURE 8.13 Arrow skeleton.



Step 3a: Layer – Transparency – Add Alpha Channel

This step is necessary for this image because there is only one layer. Earlier, when you made the ship, the alpha channel was added automatically when you created a second layer.

Step 3b: Select – By Color and click on the white background.

Step 3c: Add the alpha by doing **Colors – Color to Alpha…** and then draw a **white border** at the top and **fins** at the bottom as shown in Figure 8.14.

FIGURE 8.14 Arrow with alpha and fins.

You're getting a taste of how all graphics were created in the early days of video games. That's right, the artist, or possibly even the programmer, drew every piece of graphics one pixel at a time. That was an advance over even earlier days, when programmers typed in numbers to create the graphics. Things have come a long way since then.

Step 4: Save the image to **arrow.xcf** and **export** to **arrow.png**, and put both files into the **Assets/Sprites** folder of your Unity project, once again making sure to use compression level 0.

You can now go straight into Unity and use this new png file to make an arrow sprite.

Step 5: Open the ClassicVerticalShooter project in Unity. **Zoom** in on the Scene panel to approximately match the Game panel. Check that you have a png file called arrow, plus the xcf file for it in the **Assets/Sprites** folder.

Step 6: Set the **Filter Mode** for the **arrow** to **Point (no filter)** and **Apply**.

Step 7: Drag the **arrow** into the **Hierarchy** panel.

Step 8: Set the **Position** to (**0**, **-2.8**, **0.5**). Put the **ship** at the same position with **Z** of **0**.

The -2.8 was determined by trial and error. Make sure that the Main Camera is at (**0**, **0**, **-10**), or you might be placing the ship off-screen. Our goal is to have the ship and the arrow near the bottom edge of the Game panel.

Of course, the arrow shouldn't be just sitting out there in space. It should be shooting out of the front of the spaceship. To make that happen, you need to create a *prefab*. Prefabs are a great feature of Unity. They are templates that enable users to easily make linked copies (also called *instances*) of objects. Earlier you made a Prefabs folder to store them.

You should have Prefabs, Scenes, Scripts, Sounds, and Sprites folders in the Assets folder.

Step 9: Drag the **arrow** gameobject into the **Prefabs** folder and rename it **arrowprefab**.

Dragging objects from the Hierarchy back to the Assets folder or an Assets subfolder automatically creates a prefab.

Step 10: Drag the **arrowprefab** back into the **Scene**.

Step 11: Repeat the previous step a few times, placing arrows at different locations.

The Game panel should now have a few arrows in it as displayed in Figure 8.15.

▲ **FIGURE 8.15** Testing the arrow prefab.

Step 12: Run the game, then **stop** running it.

The ship and the arrowprefabs are stationary, whereas the stars are scrolling.

Step 13: Delete all **arrowprefabs** and the **arrow** in the **Hierarchy** panel. Careful: Don't delete the arrowprefab in the Prefabs folder.

You only need to keep the prefab itself in the Prefabs folder. You're now ready to use this prefab in your scripting.

Step 14: Save the **scene** and **project**.

Next, you'll create the code to control the ship.

Step 15: Select the **Scripts** folder and create a new **C# Script** with name **shipscript**. Enter the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class shipscript : MonoBehaviour
{
    public float shipSpeed;

    // Start is called before the first frame update
    void Start()
    {
        transform.position = new Vector3(0,-2.8f,0);
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKey("right"))
        {
            transform.Translate(shipSpeed * Time.deltaTime, 0, 0);
        }
        if (Input.GetKey("left"))
        {
            transform.Translate(-shipSpeed * Time.deltaTime, 0, 0);
        }
    }
}
```

Step 16: Save the code and drag it onto the **ship** object.

Step 17: Set **Ship Speed** to **5** in the Inspector panel and **test** out the code.

Notice that Unity displays the variable named `shipSpeed` in the code as `Ship Speed` in the Inspector panel. The extra space and the capitalization are added for cosmetic reasons. You're now controlling the ship with the arrow keys on the keyboard. You could also add mouse control as was done in the brick game, but that might make the game too easy, so that option will be left for later experimentation.

Next, you'll add a boundary check to make sure the ship doesn't disappear.

Step 18: Add the `screenBoundary` variable below the `shipSpeed` variable declaration as follows:

```
public float screenBoundary;
```

Step 19: At the end of the Update function, add the following lines:

```
if (transform.position.x < -screenBoundary)
    transform.position = new Vector3(
      -screenBoundary,
      transform.position.y,
      transform.position.z);
if (transform.position.x > screenBoundary)
    transform.position = new Vector3(
      screenBoundary,
      transform.position.y,
      transform.position.z);
```

You might be wondering why you need to create a whole new vector rather than just setting the x coordinate of the position directly. This is a result of the way C# implements vectors, so the short answer is that C# won't let you do that. If you try that you'll get an error message.

Step 20: Save the new version of **shipscript** and set the **screen Boundary** to **3** in the Inspector.

Step 21: Try out the game and test to see what happens at both edges of the screen and adjust the Screen Boundary accordingly so the ship can move close to the edge.

The ship should act like it's hitting an invisible wall. You should be able to move the ship closer to the edge, so try 4 for the Screen Boundary. You could make that number slightly larger, but for now 4 is good. You are taking tiny steps here, adding small improvements, and immediately testing them. Next, you'll add code to shoot arrows.

Step 22: Create a new **C# Script** in the **Scripts** folder and rename it to **shotscript**. Open the script and enter the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class shotscript : MonoBehaviour
{
    public float shotSpeed;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(0, shotSpeed * Time.deltaTime, 0);
    }
}
```

Step 26: Save the file in Visual Studio.

Step 27: Select arrowprefab in the Prefabs folder, click on Open Prefab, if necessary. Do **Add Component – Scripts – Shotscript**. Then set the **Shot Speed** to **4** in the Inspector.

You just created a script for the arrows, but you have no arrows in your scene right now to test. Just drag an arrow back into the scene, and test out the shot code as follows:

Step 28a: In the Hierarchy panel, click on the arrow next to **arrowprefab**

This step should bring back the mainscene hierarchy that was temporarily missing when you edited the arrowprefab.

Step 28b: Drag an **arrowprefab** into the **Hierarchy** panel and **run** the game.

The arrow should fly up and away at a fairly rapid speed. This is as good a time as any to deal with the arrow after it flies off the top of the screen. You want to destroy the arrow when that happens. This is a one-liner.

Step 29: Stop running the game, then **insert** the following code to **shotscript** Update function.

```
if (transform.position.y > 6.0f)  Destroy(gameObject);
```

Step 30: Save your change and **test** the code again.

How can you tell if it's working? Running the game looks exactly the same. There are several ways to do this.

Step 31: Change the 6.0 to a 1.0 in **shotscript**. **Test**, and then **undo** the change.

You now see how the arrow disappears when it hits a y coordinate of 1.0. This is a reasonable way to test your code, but there is a better way.

Step 32: Turn off Maximize on Play and **run** the game. **Stop** the game and **turn on Maximize on Play**.

When the game is running look at the Hierarchy panel. You'll see the "arrowprefab" object disappear soon after the arrow disappears from the top of the screen. You can also zoom out in the Scene panel and actually see the arrow disappear above the playfield area.

Why is it important to have this cleanup code even though there's no discernible difference when you play the game? You'll be creating many new shots while you're

playing the game, and every shot uses up computing and memory resources. It's a good idea to destroy shots that aren't needed any more so that you don't run out of resources, or slow down your game, or both.

Practically speaking, it would take many thousands of shots, maybe even millions, before you'd notice a difference. Nevertheless, it's a good habit to clean up after yourself. In the classic era, when memory was relatively expensive, a great deal of effort had to be put into managing memory. Even then, it was common to hear the phrase: memory is cheap. Thankfully, memory is orders of magnitude cheaper forty years later.

Speaking of cleaning up after yourself:

Step 33: Delete the temporary **arrowprefab** instance in the **Hierarchy**. **CAREFUL**: Don't delete the arrowprefab itself in the Prefabs directory!

You're now ready to launch the arrows under player control.

Step 34: In **shipscript** add the variable "shot" by adding the following line:

```
public GameObject shot;
```

Step 35: In the Update function, add the following code:

```
if (Input.GetKeyDown("space"))
{
    Instantiate(
      shot,
      new Vector3(
         transform.position.x,
         transform.position.y,
         0.5f),
      Quaternion.identity
   );
}
```

What's going on here? The Instantiate function creates a shot at the same position as the ship. The `Quaternion.identity` sets the rotation to be unchanged, which means that the arrow is pointing up.

Step 36: Save the file in Visual Studio, then select **ship** in the Hierarchy.

Step 37: Drag **arrowprefab** from the Assets panel into the **Shot** property of **Ship-script** in the Inspector panel. **Run** the game and **test**.

This is pretty easy to test. Pound on the space bar and move the ship. Your game panel should look similar to Figure 8.16.

Look at all those arrows in the Hierarchy panel. When the game is paused, you can click on each one and look at the properties. This technique can be very helpful when debugging.

Step 38: Save your Scene and Project and **exit** Unity.

In this section, you learned about prefabs and how to use them to create and destroy sprites dynamically using the `Instantiate` function. In the next section, you'll create sprites for aliens.

▲ **FIGURE 8.16** Ship shooting four Arrows.

VERSION 0.04: ALIENS

Now that you have a spaceship and shots, you need something to shoot at. True to your 2D design for this game, you'll make 2D animated sprites to display the aliens. The design for the aliens is quite simple. They'll be walking left and right in their formation using a four-frame walk animation.

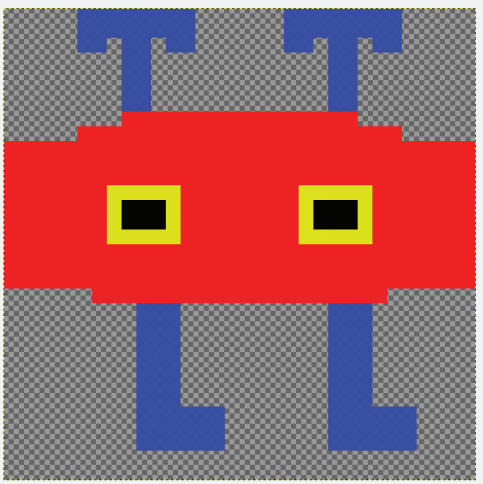
You'll be looking at 3D animation techniques later on, but for now you'll do animation the old-fashioned way: one frame at a time. This section introduces Unity's 2D animation features. First, you will create three frames of an animation in GIMP. Then you will use Unity to animate the frames.

Step 1: Open GIMP and open a **new image** with size **32 x 32** pixels. Select a **red pencil** of **size 3** and draw a red blob like the head shape in Figure 8.17.

If the background color isn't white, use the bucket fill tool to make it white.

Step 2: Draw yellow and **black eyes, blue antennas**, and **blue legs.**

You might wish to achieve symmetry using the flip tool as described earlier in this chapter. It's OK if your drawing doesn't match the one from the figure as long as



▲ **FIGURE 8.17** Alien drawing.

you draw something similar. Keep in mind that the white color will be converted to alpha just as you did earlier for the spaceship and arrow graphics, so don't use white for anything other than the background.

Step 3a: Layer – Transparency – Add Alpha Channel

Step 3b: Select – By Color and click on the white background.

Step 3c: Colors – **Color to Alpha**, save the file in the **Assets/Sprites** directory as **alien1.xcf** and then export to **alien1.png** using **compression level 0**.

Step 4: Fill the bottom alpha area with white, then redraw the legs as in Figure 8.18.

Step 5: Select – By Color, click on white, **Colors – Color to Alpha, save and export** to **alien2** as in Step 3c.

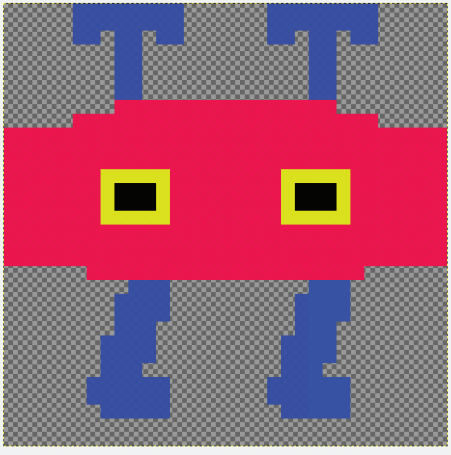
Step 6: Repeat Steps 4 and 5 using Figure 8.19 and name **alien3**.

Next, you'll bring this simple animation into Unity.

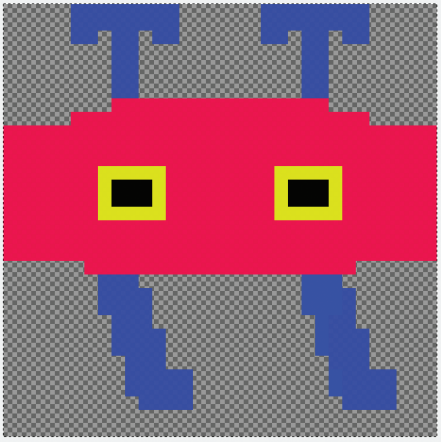
Step 7: Run Unity with the **ClassicVerticalShooter** project.

Step 8: Select the **alien1** sprite. Choose **Point (no filter) Filter Mode** and **Apply.**

Step 9: Repeat the previous step for **alien2** and **alien3**.



▲ **FIGURE 8.18** Alien frame 2.



▲ **FIGURE 8.19** Alien frame 3.

Step 10: Drag the **alien1** sprite into the **Hierarchy panel**.

If you wish, you can run the game like this and shoot at the alien. Of course, the alien doesn't get destroyed yet, nor does he move or shoot back at you.

Step 11: Select alien1 in the Hierarchy panel and then do **Window – Animation – Animation**

This is the first time you're using the Animation window. This very useful Unity feature makes it easy to do 2D animations without writing code. You may need to expand the right panel to expose the `Create` box.

Step 12: Click on the Create box in the Animation window. Name the animation alien.anim.

Step 13: Drag alien1 to time 0, alien2 to time 0:15, alien1 to 0:30, alien3 to 0:45 and alien1 to 1:00.

Step 14: Run the game. Be sure to have Maximize on Play selected when you do this.

You should see alien1 doing an animation in the Game panel. Notice that alien1 now has an Animator component.

Just as you did with the arrow object, you're going to make a prefab, so you can easily create multiple aliens using a script.

Step 15: Select the **Prefabs** folder in the **Assets** panel and drag the **alien1** object into it.

Step 16: Rename the alien1 prefab to **alienprefab**.

Recall that dragging an object into the Assets folder automatically turns it into a prefab.

Step 17: **Test** the prefab by making a few test instances in the Scene. Run the game to check out the test prefabs. **Stop** running the game.

Step 18: Delete any **alien objects** in the **Hierarchy** panel, including the original **alien1** and any objects with the name alienprefab.

You've encountered this before and it bears repeating. Be sure to only delete Hierarchy objects and keep the prefab itself in the Assets/Prefabs folder.

You're now ready to build your grid of aliens. This is done using a technique similar to your creation of the bricks in the Classic Brick Game.

Step 19: Save

This step is unnecessary, but it's a good idea to save your progress often. You can rename the project by adding version numbers into the project name. That way, if you need to, you can go back to a previous version rather than starting from the very beginning if things go awry.

Note for advanced users: It might be time to start using version control. Unity supports several possible version control systems. Feel free to explore this further on your own.

Now that you have an alien prefab you will write code to generate an array of aliens.

Step 20: GameObject – Create Empty with name **alienfactory**. Assign to it the new C# Script with name **alienfactoryscript** and move it to the Scripts folder. Then enter this code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class alienfactoryscript : MonoBehaviour
{
   public GameObject alien;

   public void MakeAliens()
   {
      for (int i = 0; i < 15; i++)
         for (int j = 0; j < 6; j++)
```

```
            {
                GameObject al = Instantiate(
                    alien,
                    new Vector3((i - 7) * 0.5f, (j - 2) * 0.8f, 0),
                    Quaternion.identity);
            }
        }

        // Use this for initialization
        void Start()
        {
            MakeAliens();
        }
    }
```

Step 21: Save your file in Visual Studio.

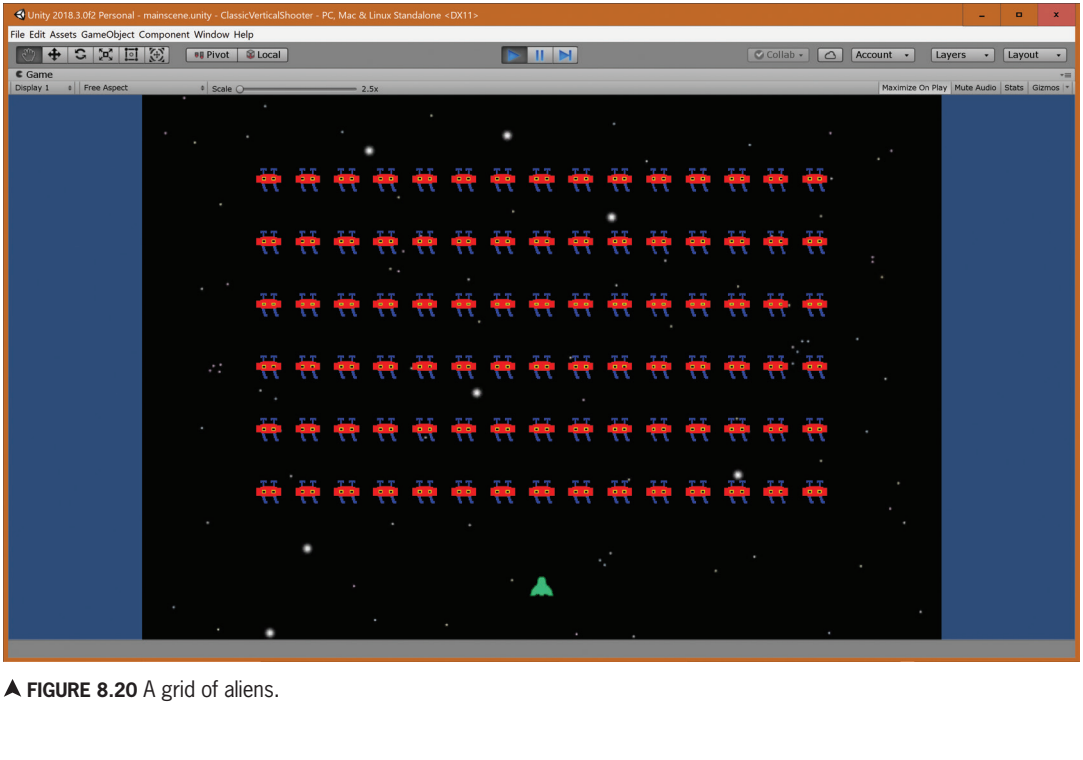
Step 22: Assign alienprefab to **Alien** in the Inspector for **alienfactory** and try out the game.

The game panel should now look like Figure 8.20. This is a good time to go through the code in alienfactoryscript. You didn't bother to keep the `Update` function because it won't be needed. This script simply creates the grid of aliens at the start of each level. There is nothing left to do during gameplay. The `MakeAliens` function consists of a double loop. It generates 15 columns of 6 aliens. The numbers inside the `Vector3` call control the positions of the aliens.

You're making some good progress. As you know, those shots from the spaceship aren't doing any damage to those pesky aliens. It's time to add collision detection between arrows and aliens.

In previous projects, it was OK to just detect collisions with any other type of object. But here you want to detect only objects of a specific type. For this you're going to use the "tag" feature of Unity.

Step 23: Create a C# script called **alienscript** and assign it to **alienprefab**.



▲ **FIGURE 8.20** A grid of aliens.

Step 24: Insert the following `OnTriggerEnter` function at the end of alienscript:

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "shot")
    {
        Destroy(gameObject);
        Destroy(other.gameObject);
    }
}
```

This function checks to see if your alien is colliding with a shot. If so, it destroys itself and the shot. In order to make this code work, you'll set up the tags for the shots. You're using Visual Studio, of course, to enter this code. Don't forget to save when you're finished editing. You can run the game now, but it still won't have collisions between arrows and aliens because the tag isn't set up yet.

Step 25: Select and open **arrowprefab** and notice that it is untagged.

You can see this by looking at the Tag property at the top of the Inspector panel. As you might guess, tags are a way to group objects together and, thus, making some scripting tasks easier. Before you can tag your arrowprefab with the "shot" tag, you need to create the "shot" tag.

Step 26: Click on **Untagged** and choose **Add Tag** at the bottom.

This activates the TagManager. That's where you create or otherwise manage the tags for your project.

Step 27: Click on the plus sign below the text "List is Empty". Give the new tag a name of "shot", and save.

Step 28: Click on arrowprefab in the Prefabs panel, and Open Prefab. This closes the tagmanager.

Step 29: Select the **shot tag** from the drop-down menu next to the Tag property.

If you're expecting your collision detection to work now, you're mistaken. You still have to carefully add some physics and collider components to your prefabs to make this work.

Step 31: Make sure that the **arrowprefab** is still selected and do **Component** – **Physics** – **Rigidbody**. **Uncheck Use Gravity** because the game is in outer space after all. Now do **Component** – **Physics** – **Box Collider**, **check Is Trigger**, and make the **Size** (**0.08**, **0.16**, **10**).

You put a box collider around your sprite and made it very narrow in the x direction and full size in the y direction. The 10 in the z direction is somewhat arbitrary. It's there to make sure that the arrow collides with anything that overlaps with it in the x and y directions.

Step 32: Select and **open** the **alienprefab** and add a **box collider** for it as well, but no rigid body. The **Size** of this box collider is (**0.32**, **0.32**, **10**) and also **check Is Trigger**.

Step 33: Test it, save, and exit.

Finally! You can shoot the aliens. It took much more of an effort this time, but you once again added just enough features to make the game playable. Well, you don't really have a playable game just yet, but you're getting a feel for what it might be like to play the game, even though those aliens are just sitting there. In the next section, you'll turn things around and introduce alien shots.

VERSION 0.05: ALIEN SHOTS

It wouldn't be fair to just shoot at the aliens without having them shoot back. This is somewhat familiar territory because you just did something similar in the previous section.

Step 1: Open GIMP and load **arrow.xcf** from the **Sprites** folder in your Unity project. Zoom in using the plus key on your Numpad, if necessary.

Step 2: Click on the **Flip Tool** in the tool grid and click on the **Vertical** radio button for the **Direction** in the **Tool Options** dialog.

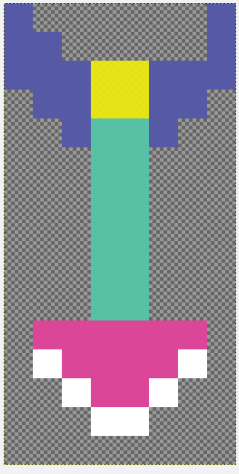
Step 3: Click on the arrow it to make it point down instead of up. Then **make some minor changes** to it using the **pencil tool**. Feel free to get creative. You may need to do an Anchor Layer command in the Layer menu to get access to drawing on top of transparent pixels.

When you're done, you should see a downward pointing arrow somewhat like Figure 8.21. You may choose your own colors, but make them bright, so the arrow can be easily seen by the player.

Step 4: Save as **ashot.xcf** and **export ashot.png** using **compression level 0**.

Step 5: Run Unity and load the **ClassicVerticalShooter**.

Once again, you'll bring in your new graphics into Unity, just as you did for the arrow.



▲ **FIGURE 8.21** Alien shot.

Step 6: Click on the **ashot** sprite and select a **Filter Mode** of **Point (no filter)**, and then **Apply**. Drag the **ashot** sprite into the **Scene panel** somewhere and use the **f key** to focus on it.

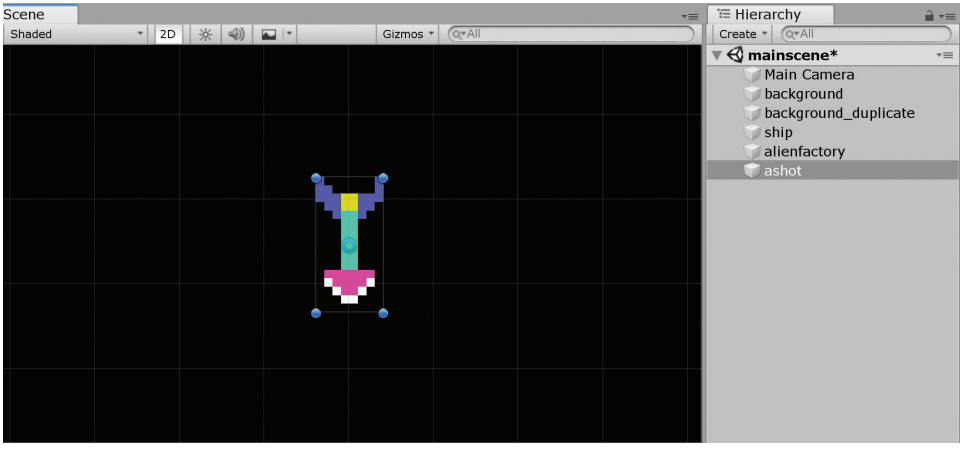
The alien shot looks clean and pristine, due to your choice of filter. Your Scene should look like Figure 8.22.

As before, the next step turns the alien shot into a prefab.

Step 7: Click on the **Prefabs** folder and drag **ashot** into it. Rename it to **ashotprefab**.

Step 8: Create a **new script**, call it **ashotscript**, assign it to **ashotprefab**, and enter the following code for it:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ashotscript : MonoBehaviour
{
    public float ashotSpeed;

    // Use this for initialization
```



▲ **FIGURE 8.22** The alien shot in the Unity scene.

```
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(0, ashotSpeed * Time.deltaTime, 0);
        if (transform.position.y < -16) Destroy(gameObject);
    }
}
```

This code simply moves the alien shot down the screen and destroys the shot if it falls off the bottom of the screen.

Step 9: Save your script code and set the **ashotspeed** to -5 for the **ashotprefab**.

Step 10: Test it.

Here's one way to test this. Drag a couple of ashotprefabs into the Scene panel and run the game. The shots should fly down the screen and disappear at the bottom. To verify that the "Destroy" is working, run the game without the "Maximize on Play" enabled and monitor the Hierarchy. Because all those aliens are clogging up the Hierarchy display, you can temporarily remove them by commenting out the `MakeAliens()` call in alienfactoryscript as follows:

```
void Start() {
// MakeAliens();
}
```

Just be sure to remove those slashes and bring back the aliens when you're done debugging.

This kind of testing, where you modify the source code in order to test, is called "white box testing." If you don't modify anything, and don't even look at the source code while testing, you're doing "black box testing."

Now it's time to make those aliens shoot at you.

Step 12: Insert the following code into the Update function in **alienscript**:

```
//  shoot sometimes

if (Mathf.FloorToInt(Random.value * 10000.0f) % 900 == 0)
{
   Instantiate(
      ashot,
      new Vector3(transform.position.x,transform.
                                                position.y,0.5f),
      Quaternion.identity
   );
}
```

This code creates a random float between 0 and 10000, converts it to an integer, then tests if it's a multiple of `900`. If it is, then the alien shoots straight down. That magic number of 900 can be adjusted to make the aliens shoot more or less frequently. This is a quick and dirty way of getting randomized shooting by the aliens. Of course, there are many other possible ways of doing this. For example, you could randomly select N number of aliens every M frames and have them shoot, with N and M depending on the level. Can you think of any other ways to implement randomized alien shooting? Maybe it shouldn't be random at all? You can leave these possibilities for future versions. It's better not to get overly distracted with too many brainstorms along the way but rather to get the quick and dirty code to actually work!

Step 13: Insert the variable **ashot** at the **beginning** of the script like this:

```
public GameObject ashot;
```

Step 16: Save the script file, select **alienprefab**, and **drag ashotprefab** to the **Ashot property** in the Inspector.

Step 17: Run the game.

You'll be getting shot at, but of course, you don't have the collision detect working yet so the alien shots go right through you without damage.

You'll do the collision handling pretty much the same way as for arrows hitting aliens.

Step 18: Add the following `OnTriggerEnter` function to **shipscript**:

```
private void OnTriggerEnter(Collider other)
{
   if (other.tag == "ashot")
   {
      Destroy(gameObject);
      Destroy(other.gameObject);
   }
}
```

Once again. you'll be using tags here. This time you'll have the tag "ashot" assigned to the ashotprefab.

Step 19: Save the **shipscript** file and select **ashotprefab**.

Step 20: Click on **Untagged** in the Inspector panel.

Step 21: Select **Add Tag** from the drop-down menu, use the plus icon to add the **ashot** tag, click on **ashotprefab**, and select **ashot** as the tag for this prefab.

Just as you had to do for the arrow prefab, it's necessary to add box collider and rigidbody components.

Step 22: Component – **Physics** – **Rigidbody** and **uncheck Use Gravity**.

Step 23: Component – **Physics** – **Box Collider** and set the **Size** to (**0.08**, **0.16**, **10**). **Check** the **Is Trigger** Box.

You also need to have a box collider for your ship.

Step 24: Select **ship** in the Hierarchy panel, do **Component** – **Physics** – **Box Collider**, and change the **Size** to (**0.32**, **0.32**, **10**).

You're finally ready to test the collision code.

Step 25: Run the game and watch what happens when an alien shot hits the ship.

Both the shot and the ship disappear. It's game over. This is a bit severe. Just one life!

You've "found the fun" just now. That's right, this game is fun just the way it is. Try to kill all the aliens. It's not exactly easy. If you think it's too easy, you can change shot speed or the rate the aliens are firing to make the game harder.

Step 26: Save your progress, take a **break**, you **deserve** it.

In the next section, you'll put some structure to the game and add multiple lives.

VERSION 0.06: SCORING AND LIVES

As the name implies, in this section you'll add scoring and lives. You'll use a technique called "finite state machines" or FSM for short. This is a common coding technique that goes way back to the early days of game development, yet is still used today. It's somewhat surprising how few of the old techniques have become obsolete many decades later.

Before you get into finite state machines, you'll put in a simple display of scoring and lives, just as in your previous projects.

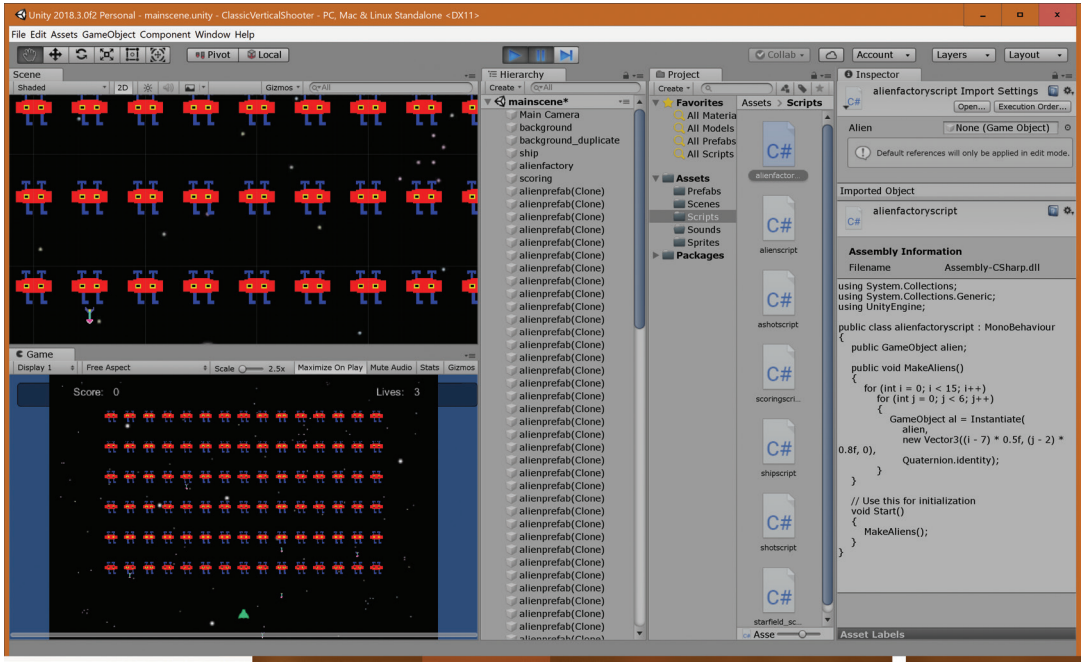
Step 1: Create an **empty GameObject** named **scoring**. **Assign** the new **C# Script** named **scoringscript** with the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class scoringscript : MonoBehaviour
{

    public static int score;
    public static int lives;

    void InitializeGame()
    {
        score = 0;
        lives = 3;
    }
```

```
    // Use this for initialization
    void Start()
    {
        InitializeGame();
    }

    private void OnGUI()
    {
        GUI.Box(new Rect(10, 10, 200, 30), "Score:    " + score);
        GUI.Box(new Rect(Screen.width - 200, 10, 200, 30), "Lives:    "
            + lives);
    }
}
```

Step 2: Run the game, **pause** it, and compare it to Figure 8.23.

You're displaying the score and the lives, though they still aren't functional. Also, take a look at the Scripts folder. You now have seven scripts: alienfactory, alienscript,



▲ **FIGURE 8.23** Initial scoring in Classic Vertical Shooter.

ashotscript, scroringscript, shipscript, shotscript, and starfield_scroller. If you're seeing something else in the Scripts folder, chances are your scripts are residing at the top level of Assets. This would be a good time to move any stray scripts into the Scripts folder.

Your Scene panel may look different depending on the zoom and pan setting. Figure 8.23 is zoomed in quite a bit, so you're seeing just a portion of the aliens. While the game is paused it is possible for you to zoom and pan in the Scene panel without affecting the game itself. This can be an effective debugging technique.

In the next step you'll get the scoring working.

Step 3: Add the following line in **alienscript** at the correct spot:

```
scoringscript.score += 10;
```

This code adds 10 to the score using the += operator. There are several other similar operators in C#, for example -= which subtracts the right side from the left side. You will find it instructive to search the Internet for the documentation for the += operator in C#. This will lead you to a more detailed explanation.

Can you guess where you should put the scoring update code in Step 3? Think about it before reading on. Usually the scoring code gets put where the action happens that you're trying to reward, in this case the destruction of the alien. Thus, the correct spot is in the `OnTriggerEnter` function, anywhere inside the curly brackets after the if statement, because that's where you're destroying the alien and the shot too. You could, of course, get much fancier with the scoring, but for now it's a good start. If you test the game right now you will see the score increase by 10 every time an alien is destroyed.

Next, you'll create finite state machine to handle the various states of the game.

Step 4: Create an **empty GameObject**, name **GameState**, script name **GameStateScript assigned** to GameState, and enter the following code:

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;

public class GameStateScript : MonoBehaviour
{

   public static int state;
   public const int PressStart = 1;
   public const int StartingPlay = 2;
   public const int GamePlay = 3;
   public const int Dying = 4;
   public const int GameOver = 5;
   public const int NextLevel = 6;

   // Use this for initialization
   void Start()
   {
      state = PressStart;
   }
}
```

The `const` declarations define states that your game can be in. They are set to be constants so that C# doesn't accidentally change them. Here is a more detailed description:

`PressStart` is the state where a press start message is displayed. During this state, the game waits for the player to hit a key to start the game. When that happens, the game enters the `StartingPlay` state.

`StartingPlay` is the state where the game is initialized and there are no user inputs being accepted yet. When the initializations are done, the game enters the `GamePlay` state.

`GamePlay` is the main state where the player is playing the game. If at any time the player gets hit by a shot, you go to the `Dying` state. If all aliens get killed, you go to `NextLevel`.

`Dying` is a state where the ship enters a death sequence, the aliens celebrate, etc. At the end of the death sequence, if there are no lives left, you go to `GameOver`, or else to `StartingPlay`.

`GameOver` is the state where a "GAME OVER" message is displayed. The message will time out and you go to `PressStart` next.

`NextLevel` is the state where a new wave of aliens is created, presumably different and more difficult.

It's now your job to start to implement these states according to this informal description.

Step 5: Run the game, then **stop running** the game.

The first step is, what else, to test the code to make sure it compiles and doesn't break anything. The GameState object doesn't do anything except initialize the "state" variable. This seems silly at first, but it's a valuable lesson. It takes about 30 seconds to test the game, so why not do it? After you're finished with your very short testing, implement the first state, `PressStart`.

Step 6: In **scoringscript**, **insert** the following lines into the `OnGUI` function:

```
if (GameStateScript.state == GameStateScript.PressStart)
{
   if (GUI.Button (new Rect (Screen.width/2 - 150,
                             Screen.height/2 - 50,
                             300, 50),"Click Me to Start"))
   {
      GameStateScript.state = GameStateScript.StartingPlay;
   }
}

   //  for debugging
   GUI.Box (new Rect (Screen.width/2 - 30,10,90,30),
       "State: "+GameStateScript.state);
```

This creates the "Click Me" button in the middle of the screen. It doesn't work quite how you want yet, but at least you can look at it. You added a debug output box to display the current game state. Unfortunately, the game state needs to be a static variable, and static variables can't easily be displayed in the Inspector, so you're using a GUI box to display the current state instead. When the game is released, you'll disable this debug display.

Displaying property values using the application is a time-honored tradition and goes way back to the old days of developing code using punch cards and line printers. It's still a useful method to use as an alternative to other debugging methods.

It's time to fix the next big problem, which is that the game is active during the PressStart state. This is pretty easy to fix. First, look at alienfactoryscript. The aliens are getting initialized in the start function, which is not where you want it.

Step 7: In **alienfactoryscript**, delete the call to MakeAliens in the Start function.

The Start function now looks like this:

```
void Start()
{


}
```

You could even remove the Start function completely, but it's easier to just leave it there for future use. Now, when you run the game, the aliens are gone. You'll get them back as follows.

Step 8: Open **shipscript.cs** and add the following variable declaration at the top:

```
public alienfactoryscript alienfactory;
```

Step 9: Change the name of the `Update` function to `ShipControl`, and insert the following new `Update` function below the end of `ShipControl`:

```
void Update()
{
   if (GameStateScript.state == GameStateScript.GamePlay)
       ShipControl();
```

```
    if (GameStateScript.state == GameStateScript.StartingPlay)
    {
        alienfactory.MakeAliens();
        GameStateScript.state = GameStateScript.GamePlay;
    }
}
```

Step 10: Save the file, select **ship** and assign **alienfactory** to the new Alienfactory variable in the **Shipscript (Script)** section in the Inspector panel.

Do this by dragging alienfactory from the Hierarchy into the Alienfactory property in the Inspector. If the Alienfactory property isn't there, be sure to save your editing in Visual Studio and run the game once to bring in the new version of shipscript.

It's instructive to follow the new logic in the Update function. You are only allowing the player to control the ship during the GamePlay state, and you initialize the aliens during the StartingPlay state, immediately followed by a transition to the GamePlay state. If you test the game right now, you can click on the "Click me to Start" button and play the game after that. Your next step is to make the lives counter work.

Step 11: Enter this new `OnTriggerEnter` function for **shipscript**:

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "ashot")
    {
        scoringscript.lives--;
        if (scoringscript.lives == 0)
        {
            Destroy(other.gameObject);
            GameStateScript.state = GameStateScript.GameOver;
        }
    }

}
```

Instead of destroying the ship every time it gets hit by an alien shot, you now decrease the lives counter. When the lives counter reaches zero, rather than destroying the ship, you change state to GameOver.

Step 12: Save the shipscript file and now **add** the following **code fragment** to the `OnGUI` function in **scoringscript.cs**:

```
if (GameStateScript.state == GameStateScript.GameOver)
{
   if (GUI.Button(new Rect(Screen.width / 2 - 200,
                           Screen.height / 2 - 50,
                           400, 50), "Game Over, Try again"))
   {
       InitializeGame();
       GameStateScript.state = GameStateScript.PressStart;
   }
}
```

The game structure almost works now, but there's a tricky bug. At the end of the game, you need to clean up after yourself and delete all the leftover aliens.

Step 13: Add the following code to the **Update** function in **alienscript.cs**:

```
if (GameStateScript.state == GameStateScript.GameOver)
{
    Destroy(gameObject);
}
```

This code is particularly interesting. It makes all aliens destroy themselves when the game state is GameOver. Now save all your script files and try out the game. It still has some problems, but it's basically playable.

Make the following minor adjustment. You're killing off the leftover aliens too soon. If you wait until you enter the PressStart state, it'll look a little better.

Step 14: In **alienscript replace** the `GameOver` with a `PressStart` as follows:

```
if (GameStateScript.state == GameStateScript.PressStart)
{
    Destroy(gameObject);
}
```

Now the aliens disappear after you click on the "Game Over" message.

Your next goal is to have the ship disappear and reappear depending on the game state. First, fix the bug where the lives counter reaches -1. It's bad enough that you can have zero lives left, but you don't even want to think about what it would mean to have negative lives.

Step 15: In **shipscript.cs**, add the following line at the beginning of the `OnTriggerEnter` function, immediately before the tag test:

```
if (GameStateScript.state == GameStateScript.GamePlay)
```

This assures that you have no ship-vs.-alien shot collisions except during gameplay.

You're now ready to add the death sequence for the ship. Currently, when the ship gets hit, it either has no reaction or, if you're on the last life, it just freezes and it's game over. What you really want is some kind of animation that shows the ship got hit, have the ship disappear for a while, and then you either go into the game over state or you try again with another ship. Here is where the "Dying" state gets used.

Step 16: Add a `deathtimer` variable declaration to **shipscript.cs** immediately above the `Start` function as follows:

```
float deathtimer;
```

Step 17: Add the following code fragment to the `Update` function:

```
if (GameStateScript.state == GameStateScript.Dying)
{
   transform.Rotate(0, 0, Time.deltaTime * 400.0f);
   deathtimer -= 0.1f;
   if (deathtimer < 5.0f)
   {
       GetComponent<Renderer>().enabled = false;
```

```
    }
    if (deathtimer < 0)
    {
        GameStateScript.state = GameStateScript.GamePlay;
        transform.position = new Vector3(
           0.0f,
           transform.position.y,
           0.0f);
        transform.rotation = Quaternion.identity;
        GetComponent<Renderer>().enabled = true;
    }
}
```

Step 18: Change the `OnTriggerEnter` funtion as follows:

```
private void OnTriggerEnter(Collider other)
{
    if (GameStateScript.state == GameStateScript.GamePlay)
    if (other.tag == "ashot")
    {
        scoringscript.lives--;
        deathtimer = 10.0f;
        GameStateScript.state = GameStateScript.Dying;
        if (scoringscript.lives == 0)
        {
            Destroy(other.gameObject);
            GameStateScript.state = GameStateScript.GameOver;
        }
    }
}
```

The only change in the `OnTriggerEnter` function was to initialize the `deathtimer` and to change the state to `Dying`. Notice that when the lives counter hits zero, you bypass the `Dying` state and go directly to GameOver. This isn't quite what you want but it's good enough for now.

When you test this code, the ship does a rotation animation when hit, disappears, and then reappears, sometimes with disastrous consequences because it might get resurrected right on top of an alien shot! The fix for this is to have the aliens stop shooting when the ship is in its death sequence.

Step 19: Add the following line in **alienscript**:

```
if (GameStateScript.state == GameStateScript.GamePlay)
```

as the first statement in the `Update` function.

Now clean up what's happening with the ship right before Game Over. When you detect a collision with an alien shot, you go into the `Dying` state, regardless of how many lives are left.

This has the effect of simplifying the `OnTriggerEnter` function.

Step 20: In **shipscript.cs**, **replace** the `OnTriggerEnter` function with the following code:

```
private void OnTriggerEnter(Collider other)
{
   if (GameStateScript.state == GameStateScript.GamePlay)
   if (other.tag == "ashot")
   {
       scoringscript.lives--;
       deathtimer = 10.0f;
       GameStateScript.state = GameStateScript.Dying;
       Destroy(other.gameObject);
   }
}
```

Step 21: Also in **shipscript.cs**, **edit** the `Dying` section at the bottom of the **Update** function to look like this:

```
if (GameStateScript.state == GameStateScript.Dying)
{

   transform.Rotate(0, 0, Time.deltaTime * 400.0);
```

```
        deathtimer -= 0.1;
        if (deathtimer < 5.0)
        {
            GetComponent<Renderer>().enabled = false;
        }

        if (deathtimer < 0)
        {
            GameStateScript.state = GameState.GamePlay;
            Transform.position = new Vector3(
            0,transform.position.y,0);
            Transform.rotation = Quaternion.identity;
            GetComponent<Renderer>().enabled = true;

            if (scoringscript.lives == 0)
            {
                GameStateScript.state = GameStateScript.GameOver;
            }
        }
    }
```

Step 22: Save your files, **test**, and **exit** Unity.

In this section, you developed a finite state machine to handle the basic structure of your game. In the next section, you'll implement a death animation for the aliens by creating a small finite state machine for each alien.

VERSION 0.07: ALIEN DEATH SEQUENCE

Your next goal is to have the aliens go through a death animation when they get hit by an arrow and then disappear. This is pretty similar to what you just did with the player character, so it will seem like familiar territory. The main difference is that you are now dealing with an entire array of aliens.

Step 1: Run Unity, take a look at the code below and then **edit alienscript.cs** to match. There are some underlines on the left side of this code listing to indicate new lines.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class alienscript : MonoBehaviour
{
      public GameObject ashot;
      public int state;
      public float timer;

   // Start is called before the first frame update
   void Start()
   {

   }

   // Update is called once per frame
   void Update()
   {
      if (GameStateScript.state == GameStateScript.GamePlay)
      if (Mathf.FloorToInt(Random.value * 10000.0f) % 900 == 0)
      {
          Instantiate(
             ashot,
             new Vector3(
                transform.position.x,
                transform.position.y,
                0.5f),
             Quaternion.identity
          );
      }
```

```
__      // if it's dying go through the death sequence
__      if (state == 1)
__      {
__          transform.Rotate(0, 0, Time.deltaTime * 400.0f);
__          transform.Translate(
__                0.3f * Time.deltaTime,
__                3.0f * Time.deltaTime,
__                0, Space.World);
__            transform.localScale = transform.localScale * 0.99f;
__            timer -= 0.1f;
__            if (timer < 0.0f)
__            {
__                Destroy(gameObject);
__            }
__       }
        if (GameStateScript.state == GameStateScript.PressStart)
        {
            Destroy(gameObject);
        }

}

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "shot")
        {
            scoringscript.score += 10;
__          state = 1;
__          timer = 5.0f;
            Destroy(other.gameObject);
        }
    }
}
```

Don't forget to delete the "Destroy(gameObject)" line at the end of OnTrigger-Enter!

Step 2: We also need to initialize the alien state to 0 in **alienfactoryscript**. Replace the `MakeAliens` function with the following code:

```
public void MakeAliens()
{
   for (int i = 0; i < 15; i++)
        for (int j = 0; j < 6; j++)
        {
           GameObject al = Instantiate(
              alien,
              new Vector3((i - 7) * 0.5f, (j - 2) * 0.8f, 0),
              Quaternion.identity);
           al.GetComponent<alienscript>().state = 0;
        }

}
```

That looks like a lot of code, but most of it got entered earlier in this chapter. This is a good time to review it, try to understand it, and to make sure your old code didn't get changed somehow. The new code has to do with the state and timer variables. The initialization in alienfactoryscript is just one line. The alienscript changes are more substantial, but also straightforward.

The strangest thing is the line with the 0.99 in it. That line makes your object smaller by 1 percent. The effect is that the aliens appear to shrink as they spin off the top of the screen. Notice that you have two state variables affecting the aliens, the game state and the alien state. The alien state is very simple. If it's 0, it's alive and kicking, if it's 1, it's dying. Those constants in the code, numbers such as 0.99, 400.0, 3.0, and 0.3, are commonly called *fudge factors*. Yes, really. It's fun to change the numbers and watch the effect on the death sequence in the aliens. In production

code it's a good idea to replace the fudge factors by more meaningful variables and to document the effects. Sometimes though, it makes the code easier to deal with and to understand if the fudge factors are "hardwired" into the code, such as in the current version of alienscript.

Step 3: Test, **save**, and **exit**.

The game is starting to look pretty good, but you still don't have sound!

VERSION 0.08: SOUND

Designing the sound for a game can be a full-time job for several people in a major title. For you, it's a small section in a large chapter. The classic approach to sound effects in games is to just throw some simple effects in there without too much planning, experiment a little bit, and don't worry about being realistic.

The vacuum of outer space is completely silent. This hasn't stopped countless sci-fi movies from adding sound effects to their space battles. You're going to keep things extremely simple and just do two sound effects and no music in this game. You need a sound effect for when the arrow gets launched, and another for when aliens get hit. Most space shooters use some kind of laser "bleep" for shots and an explosion sound when an alien gets hit.

Step 1: Start **Audacity**.

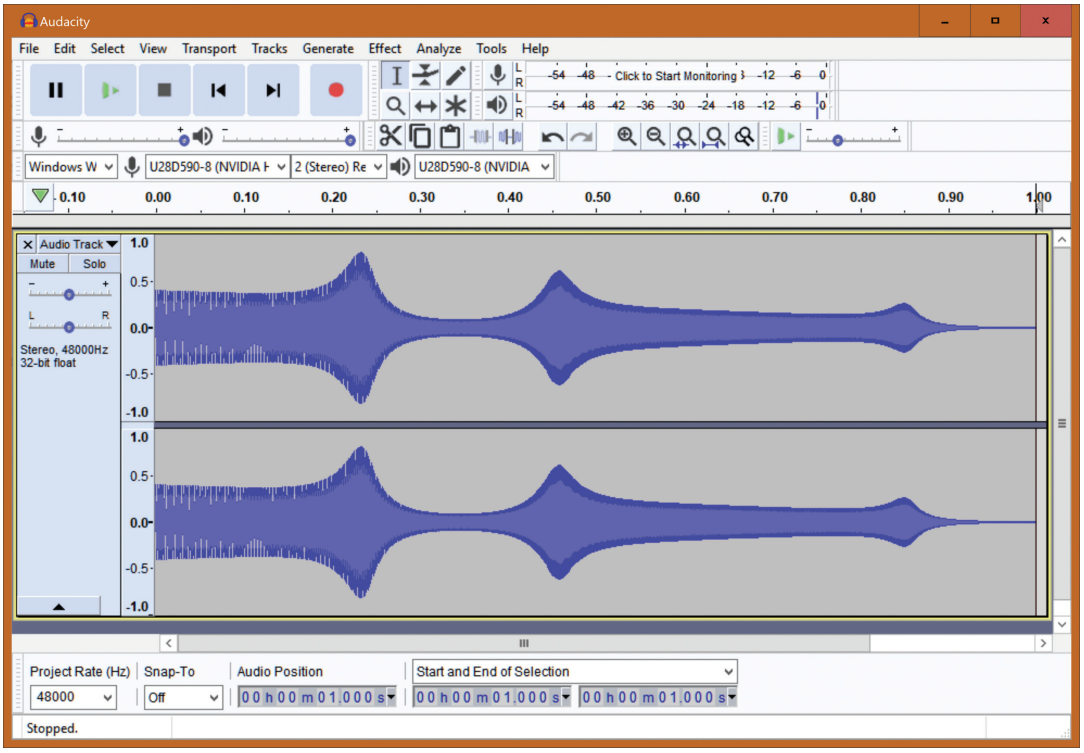
Step 2: Tracks – Add New – Stereo Track.

Step 3: Generate – Chirp with the settings **Frequency Start 440**, **End 1320**, **Amplitude Start 0.8**, **End 0.1**, **Interpolation Linear** and **Duration 1 second**.

Step 4: Apply the **Wahwah** effect with settings **LFO Frequency (Hz): 1.5**, **LFO Start Phase (deg.): 0**, **Depth (%) 70**, **Resonance 2.5, Wah Frequency Offset (%) 30** and **Output Gain (db) -6.0**.

Your wave forms should look like Figure 8.24.

That sounds pretty weird. You want your sound effect to be shorter.



▲ **FIGURE 8.24** Wahwah sound effect.

Step 5: Select everything after 0.30 and **delete** it. **Listen** to what's left, then **Save** the project to **cshot.aup** and **export** to **cshot.wav** to the Assets/Sounds folder of your Unity project. **Exit** Audacity.

Next, you'll make a simple explosion sound.

Step 6: Start Audacity, **add** a **new stereo track**, **Generate** – **Noise** with **Noise type: Pink**, **Amplitude 0.7** and **Duration** of **1 second**. Do **Effect** – **FadeOut** and the **Effect** – **WahWah** with the same settings as the cshot sound effect.

This sound effect is even stranger, just what you want. Of course, feel free to make your own different and bizarre sound effects. The only thing that really matters is that they are about 0.3 seconds and 1 second in duration, and even that can be changed quite a bit.

Step 7: Save this "explosion" sound effect to **cexplo.aup** and **export** to cexplo.wav, both in the Assets/Sounds folder of your Unity project.

Was there any method to this madness? Not really. Just as in the old days when technology was much more primitive, game designers simply fiddled with the numbers until they liked what they heard. Today, many independent game developers still make sound effects this way, mainly because it's cheap and fun.

Step 8: Open Unity and load the **ClassicVerticalShooter** project.

Step 9: Verify that the sounds are in the Assets/Sounds folder. Preview them.

Step 10: In **shipscript.cs**, in the section where you test for the `space` key, right after the `Instantiate`, insert

```
gameObject.GetComponent<AudioSource>().Play();
```

Step 11: Add an **AudioSource** component to the **ship**, select **cshot** as the AudioClip, and **uncheck Play on Awake**.

Step 12: Test and **Save**.

If everything worked, your cshot sound will play every time you fire a shot with the space bar. One fun slider to play with is the Pitch in the Audio Source section.

Step 13: Set the **Pitch** in the Audio Source section to **1.4**.

That was pretty much the same procedure you used in your previous projects. For the explosion sound the procedure is very similar.

Step 14: Do Step 10 but insert the code in alienscript immediately after the score is increased by 10.

Step 15: Add an Audiosource component to alienprefab, select cexplo as the AudioClip and uncheck Play on Awake.

Step 16: Save, test, and **exit** Unity.

Additional sound effects are certainly possible for this game. Feel free to add your own sounds for aliens shooting, game over, and maybe even a speech sound for

starting the game. Check out the exercises at the end of this chapter for more possibilities for sound.

In the next section you'll put in level handling.

VERSION 0.09: LEVELS

So far one of the truly defining features of classic gaming has been ignored: difficulty ramping implemented as levels. A large majority of video games increase the difficulty of the game depending on the progress of the players. This is only logical. The players would get bored if the games didn't continue to challenge them as they got farther along.

In your vertical shooter, you'll increase the difficulty with each wave of aliens. You'll keep it simple and increase the rate of shots getting fired. In preparation for this, you need to think about how to test it. It would be a lot simpler if there were fewer aliens.

Step 1: Run Unity and in **alienfactoryscript** change the **15** and the **6** to **2** and **2** and **save** the file.

You should now see just four aliens instead of 90. It's now a lot easier to shoot all the aliens.

The next thing to think about is how to detect when you have no more aliens on the screen. That's going to be your trigger for starting the next level. To do this you create a variable to count how many aliens exist at any given moment.

Step 2: Declare the **aliencounter** variable at the beginning of **scoringscript** as follows:

```
public static int score;
public static int lives;
public static int aliencounter;
```

You're declaring it in scoringscript rather than in alienscript because it's a single global variable, and that's a good place for it. The initialization needs to happen when you create the aliens, which is in alienfactoryscript.

Step 3: In the `MakeAliens` function, at the beginning, just before the double loop, **insert** the line

```
scoringscript.aliencounter = 0;
```

Step 4: Inside the double loop, before the `Instantiate` statement, immediately after the opening bracket, **insert** the following line:

```
scoringscript.aliencounter++;
```

Step 5: Save the file in Visual Studio.

To see that it's working, change your debug display.

Step 6: In **scoringscript**, change the debug display code as follows:

```
GUI.Box(new Rect(Screen.width / 2 - 60, 10, 120, 30),
   "Aliencounter: " + scoringscript.aliencounter);
```

You had to make the rectangle larger to fit the longer label.

Step 7: Save all your changes in Visual Studio and **run** the game.

You should see an aliencounter of 4 displayed at the top center of the game screen. Next, make the counter decrease when aliens get destroyed. This is easy to do in alienscript.

Step 8: In **alienscript.cs**, insert the following code immediately after the `Destroy(gameObject)` statement at the end of the death sequence code:

```
scoringscript.aliencounter--;
```

Step 9: Save your work and **try it out**.

The `aliencounter` variable should now decrement whenever an alien disappears. What's next? How about another small change with big consequences!

Step 10: At the beginning of scoringscript, insert a new `level` variable as follows:

```
public static int score;
public static int lives;
public static int level;
public static int aliencounter;
```

Step 11: Add the following

```
if (scoringscript.aliencounter == 0)
{
   GameStateScript.state = GameStateScript.StartingPlay;
   scoringscript.level++;
}
```

immediately after `scoringscript.aliencounter--` in **alienscript.cs**.

Step 12: Back in **scoringscript.cs**, initialize `level` in the `InitializeGame` function:

```
void InitializeGame()
{
   score = 0;
   lives = 3;
   level = 0;
}
```

Step 13: You also want to display the level, so change the debug display to this:

```
// for debugging
   GUI.Box (new Rect (Screen.width/2 - 60,10,120,30),
      "Level: "+scoringscript.level);
```

Step 14: **Save** all your changed script files and try out the game.

Your level should be displayed, and it should increment every time you clobber those four aliens. Also, merely by transitioning to the `StartingPlay` state in Step 11, you automatically get a new batch of aliens via the `Update` function in shipscript.

Now do some cleanup. You started with level 0, but that was a mistake. People want to start at level 1.

Step 15: Change the initialization of `level` to **1** in **scoringscript.cs**.

Step 16: The comment "`// for debugging`" is incorrect at the bottom of the **scoring** script, so replace it with "`// level display`". Save your changes and test again.

In the old days, many decades ago, programmers were encouraged, or even required by their employers, to put lots of comments into their code. Years of experience have taught us that comments are often incorrect, especially when the code gets reworked and changed a lot. The modern bias is to write code so well and so clearly that comments become mostly unnecessary.

You can never test enough, and this is a great example. You have a pretty serious bug. If you wish, you can try to duplicate it by dying right as you shoot the last alien. Guess what, your ship never comes back even though you get advanced to the next level. It takes some patience to do this, or you can increase the shot rate of the aliens to make testing easier.

What's going on here? Well, the player is still in the death sequence when we're changing state, which doesn't work. To fix it, do the following:

Step 17: Take the following code section from alienscript.cs:

```
if (scoringscript.aliencounter == 0)
{
    GameStateScript.state = GameStateScript.StartingPlay;
    scoringscript.level++;
}
```

and move it to the `Update` function in shipscript.cs, immediately after the call to `ShipControl`. Use cutting and pasting to do this edit fairly quickly.

The beginning of that `Update` function should now look like this:

```
if (GameStateScript.state == GameStateScript.GamePlay)
{
    ShipControl();
    if (scoringscript.aliencounter == 0)
    {
        GameStateScript.state = GameStateScript.StartingPlay;
        scoringscript.level++;
    }
}
```

The effect of this change to your code is that you're only advancing to the next level during `GamePlay`, not during the `Dying` state.

Step 18: Save your changes in both of the affected files and test again, making sure that you can die and advance to the next level after the death sequence completes.

This was a nasty and subtle bug that can only be revealed by thorough testing. You're lucky that it was found now, rather than after release.

Finally, you'll make use of the `level` variable and increase the difficulty of the game depending on the level. A simple start is to change the firing rate of the aliens depending on the level.

Step 19: Add the following statement at the beginning of the alienscript class in **alienscript.cs**:

```
int[] levelarr = { 50, 30, 20, 10 };
```

This creates an array of tuning numbers for the first four levels of the game.

Step 20: Replace the beginning of the `Update` function with:

```
//  shoot sometimes

int levindex;
levindex = scoringscript.level - 1;
if (levindex > 3) levindex = 3;
if (levindex < 0) levindex = 0;

if (GameStateScript.state == GameStateScript.GamePlay)
    if (Mathf.FloorToInt(Random.value * 10000.0f) %
        (
        levelarr[levindex]
        * scoringscript.aliencounter) == 0)
    {
        Instantiate(
            ashot,
            new Vector3(
                transform.position.x,
```

```
            transform.position.y, 0.5f),
        Quaternion.identity
        );
    }
```

This code looks at the array `levelarr` and, depending on which level we're at, shoots alien shots at that level's shooting rate. The shooting rate also depends on the aliencounter in order to make the game get more aggressive when there are fewer aliens on screen.

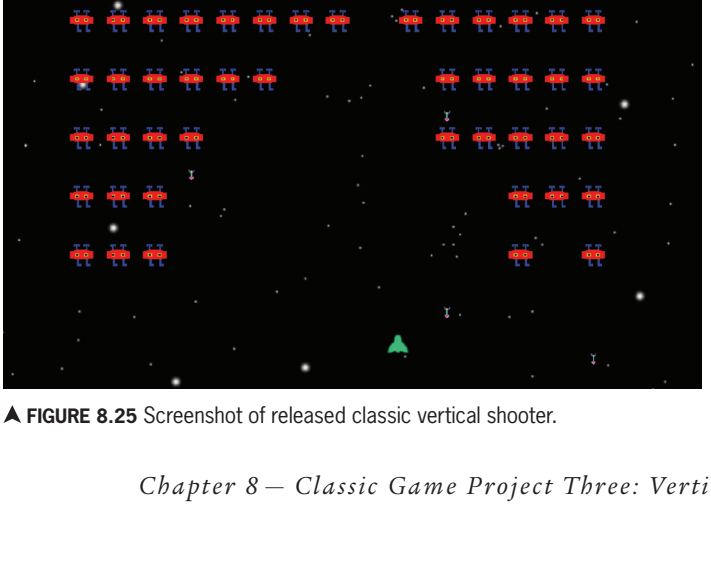
Additional difficulty ramping possibilities are explored in the exercises at the end of the chapter.

Step 21: Save your edits, **test** the game.

Wow, this game is difficult at level 4, and you haven't even brought back the original 90 aliens.

Step 22: In **alienfactoryscript**, bring back the **15** and **6** in the loop statements. **Test** and **tune**. **Save** and **exit**.

You have a pretty good game here with a world of potential for expansion and enhancements. The time has come to release it. Figure 8.25 shows a screenshot.



▲ **FIGURE 8.25** Screenshot of released classic vertical shooter.

VERSION 1.0: RELEASE AND POSTMORTEM

You've reached an important milestone. You now know enough basic game development techniques to make some interesting 2D and 3D games. You know how to make graphics, animations, sounds, collisions, and basic game logic.

The game is ready for release the way it is, but, of course, there's always room for improvement. The biggest and most obvious problem is that you're not matching the initial design sketch very well. There are several missing elements such as the barriers. This is perfectly OK. Also, you didn't put in the ending yet. What matters is that the game is fun and makes sense. It's up to you to take it from here and add the missing elements, or to invent your own enhancements. Check out the exercises for additional development practice and ideas for where to take the game from here.

The best part is that it's really fun to play, and you have a mechanism in place to tune the difficulty ramping. It's still a very small game by today's standards, but it wouldn't be that difficult to make it larger. The colors are vivid and the sound effects are weird and fit the game well.

The development of the game was a valuable learning experience. You used GIMP, Audacity, and Unity, and saw how the different parts fit together and interact. You also did much more programming this time around, but it was relatively easy compared to production coding. The built-in functions of Unity do much of the heavy lifting. On the negative side, it all seemed just a little more difficult than it should have been. This is the nature of technical work. As great and useful as the tools are, there are still hoops to jump through and hurdles to overcome to make it all happen the way you want. It seemed tedious to have to type all that code. There really ought to be an easier way, but the sad reality is that real game development can be tedious at times, especially when you're doing something new.

The single most important lesson from this chapter is simply this: Take small steps, test each step along the way whenever possible, and then fix any problems and bugs right away. All in all, it felt good to create this game. If you did all the steps in

this chapter and finished with a playable game, congratulations! Now keep going, because this is just the beginning.

EXERCISES

1. Make the starfield scroll horizontally instead of vertically. Then, make the starfield scroll in different directions depending on the level of the game.
2. Create a second starfield in GIMP and use alpha to make the background transparent. Display the second starfield on top of the original starfield and scroll it at a different vertical rate.
3. Draw a more detailed starship in GIMP using a 64 x 64 texture. Give it a different color scheme. Integrate the new starship into your game by using a variable named "`shiptype`" and setting it to 1 for the original ship, 2 for the new ship. At the beginning of the game, allow the user to choose which ship to use by pressing a key on the keyboard. Integrate a text display that explains which keys to use for which ship type.
4. Use GIMP to draw an animated arrow with 4 frames using the same technique you used to animate the aliens. Replace the arrow in the game with the animated arrow and animate it in Unity.
5. Rearrange the aliens into a grid of 12 by 4 aliens. Change the layout of the aliens so they cover the entire top half of the screen.
6. Change the "shoot sometimes" code to something less random. Add a shot timer to each alien and have each alien shoot after the timer expires. Then reset the timer based on a random range of values.
7. Create a sound effect for the loss of a life by the ship using Audacity and make it work in the game.
8. Use a recording device to record your own voice saying, "Game Over." Use Audacity to edit the sound and integrate the sound into the game using Unity.
9. Make the aliens move left and right, similar to the movement in *Space Invaders*.

10. Add barriers at the bottom of the screen and have them block shots both by you and by the aliens. Have the barriers show destruction every time they get hit by somebody. When they get destroyed, have them disappear from the screen entirely. Optional: Animate the barriers so that they move left to right.
11. Make the scoring fairer by increasing the score value of the aliens depending on the level. Change the graphics of the aliens depending on the level.
12. Create two more alien types by drawing them in GIMP and putting them into the game. Arrange the different alien types row by row, so that the top two rows have different aliens than the next two rows, etc. Make the new aliens more valuable by increasing the score awarded for hitting them.
13. Create a flying saucer at the top of the screen, have it move left to right. Make it difficult to hit and make it worth 1000 points when the player shoots it.
14. Show the high score on top of the screen at all times. Optional: Save the high score in a file every time it changes and load it from that file when the game starts.

15*. Use Blender to make a 3D model of the arrow. Make a spinning animation of the arrow and save the animation in a series of eight .png files. Redo Exercise 4 with these graphics instead of the hand-drawn GIMP graphics. This technique is called pre-rendering.

16*. Use the pre-rendering technique from Exercise 15 to make animations for the ship, the aliens, and the flying saucer in Exercise 13. Use them in the game.

CHAPTER

9 Scramble

IN THIS CHAPTER

Scramble is one of the first scrolling arcade shooters, developed by Konami and distributed by Stern in the United States in 1981. It introduced millions of players to forced scrolling backgrounds, checkpoints, and the concept of level design.

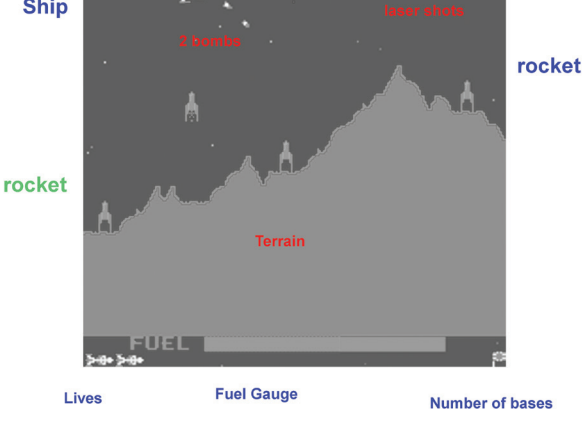
SCROLLING SHOOTER

Scramble was, in its day, one of the major arcade games. The game is still a lot of fun today, decades later, and well worth a closer look. In *Scramble*, the player moves a spaceship along in a forced scroll, shoots aliens while trying to stay alive, and explores new levels on the way to the goal of destroying a well-fortified base.

Figure 9.1 shows the basic screen layout of *Scramble*.

Movement is always to the right, and the screen scrolls at a constant speed. The player character can move anywhere within the confines of the camera view. Controlling the movement of the spaceship is critical. If the ship touches the ground or any other solid object, the player loses a life.

FIGURE 9.1 *Scramble* screen layout.



There are two weapons, a laser to the right and a bomb to the right and down. Each weapon is controlled by its own button. An eight-way arcade joystick controls movement.

EXPERTS RULE

Scramble uses the idea of a *checkpoint*. Checkpoints are invisible spots in the terrain. When a player dies, he continues the game from the closest checkpoint, provided the player has previously crossed that checkpoint.

This feature can be generalized to the following rule:

Rule 6: Experts Rule: Keep experts interested.

It's boring to repeat the same levels over and over, especially for experts. This realization has led to several advances in game design such as checkpoint systems, level-select, and secret warps. But it all boils down to keeping players interested in the game, regardless of their skill level or familiarity with the game.

Scramble ramps difficulty in a subtle way for the benefit of experts. After the base is destroyed, the six levels repeat, but the rate of fuel consumption is increased. After the third base, the game stops ramping difficulty. The designers decided that the game was difficult enough at that point. They were correct, in a way, except that many top experts had no difficulty playing the game all day long.

In the early '80s, Atari coin-op used the phrase "lunatic fringe" for the players who could play arcade games for hours on a single quarter. The feeling was that there weren't very many players like that so they didn't really matter. Later on, arcade game designers realized that the top experts do matter because they would tie up machines for too long. This led to the invention of *level-select*, first used in *Tempest*. Level-select allows a player to select a starting level at the beginning of the game, and at the end of a game the player could start another game at the beginning of the most recently completed level group. Level groups were designed to be long enough

to stop beginners from getting through them, but short enough to allow experts to zip through.

Some years later, level-select was replaced with add-a-coin, a feature that contributed to the demise of the entire coin-op industry! The add-a-coin feature simply allowed people to add a coin at game over and keep playing at essentially the same point of the game. This encouraged players to put a lot of coins into a new game to see how far they could go. Eventually, they would run out of time and money and go home. The next time, in order to get to the same spot in the game, they would have to put in a lot of quarters again, so usually there wouldn't be a next time.

Level-select, also used in Atari's *Millipede*, led to a better experience for the players. They would put in a few quarters to reach a point where they were challenged but not frustrated. The starting level would stabilize and players would then play many games at that stabilized starting level.

Why did add-a-coin lead to the demise of coin-op? It's simple. Games that incorporated add-a-coin would make good money in the first week or two at a location, but then the earnings would drop dramatically. Needless to say, this was not good business. Of course, the rise of home consoles is generally seen as the real culprit, but add-a-coin didn't help.

SCRAMBLE SEQUELS

Konami's official sequel to *Scramble* is *Super Cobra* (1981), a very similar game when compared to *Scramble*. The player character in *Super Cobra* is a helicopter and there's more of the same design elements. There are eleven sections per level instead of six in *Scramble*, and there's a larger variety of enemies, including tanks that move. In general, the game is more difficult than *Scramble*, and there's more territory to explore, but the controls are the same, and the quest for fuel still dominates the gameplay.

Later on, *Gradius*, *Parodius*, and *Xevious*®, while not officially sequels of *Scramble*, share significant design elements with *Scramble*. The arcade shooter genre was

eventually replaced by first-person shooters as the favorite for hard-core gamers, but there's a little bit of *Scramble* in every modern FPS.

The forced scrolling mechanic lives on as a popular control mechanism in platformers. While it's true that platformers mostly allow the player to control scrolling, it's a nice change of pace to include a few forced scrolling levels, for example, the underwater levels in the *Super Mario Bros.*® series.

Years later, the concept of a *rail shooter* emerged, which is basically any shooting game or level in a shooting game where your main path is on a rail, though your specific movement might be controllable within the confines of the main path. There are too many games in this genre to mention here, but they all can trace their origins to the early forced scrollers.

In the next chapter, you'll be designing and developing a side scrolling game inspired by the scrollers of the '80s but implemented using Unity's 3D engine.

CHAPTER 10

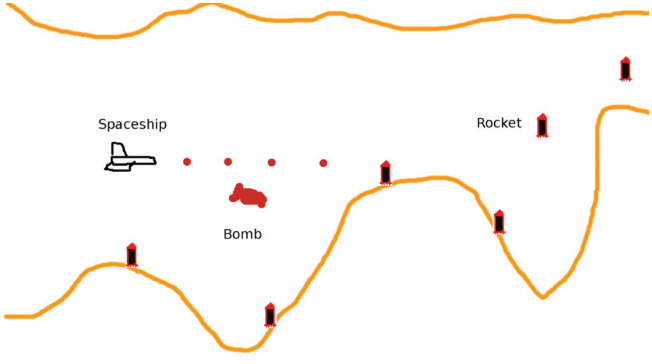
Classic Game Project Four: Scrolling Shooter

DESIGNING A SCROLLING SHOOTER

In the fourth classic project, you'll make a scrolling shooter in the spirit of *Scramble*. In the early '80s, scrolling shooters typically would scroll in a horizontal direction with the playfield scrolling to the left, which makes the player character appear to be moving to the right. Soon thereafter vertical scrolling shooters would become commonplace with the playfield scrolling down, making the player character appear to be moving up. Regardless of the scroll direction, this really opened up great possibilities and challenges. The big design issues with these types of games revolve around the backgrounds, the enemies, the weapons, and the player controls.

Once again, you'll start by making a simple sketch of the game screen. Take a look at Figure 10.1. It shows a spaceship flying along an alien planet getting attacked by rockets launched from the ground and flying enemies above ground. The spaceship has bombs and horizontal shots as weapons to fight back against its enemies.

FIGURE 10.1 Game sketch of scrolling shooter.



Sometimes it's easier to just use GIMP instead of paper to make the sketch. To put all

those identical rockets into your sketch, you can make a separate image and save it as a custom brush. This is enough of a concept to get you started, even though the sketch doesn't show everything, such as scoring.

You could choose to use Unity's 2D mode as in Chapter 8 to make this game, but here is a great opportunity to move to 3D, especially because both Blender and Unity are designed to make 3D art and games. In the early '80s, real-time 3D was just getting invented and very costly, so for most arcade game developers that wouldn't have been a practical option.

It's important to distinguish between a 3D tool chain and 3D gameplay. Our tool chain fully embraces 3D technology, including 3D models, a perspective view, and 3D lighting. Contrast that with the gameplay, which is firmly rooted in 2D. Over the years, this way of developing classic games and their sequels has become very popular with game developers.

The basic idea for using 3D technology on a 2D game is simple: Make a 3D game but give the player 2D controls. Usually this is done by limiting the location of the player character to a 2D plane and putting the camera at a fixed distance from that plane. The camera looks in a direction that's perpendicular to that plane. That's the setup you used in your Classic Paddle Game and Brick Game. This time around you're going to move the camera, which will result in a scrolling effect.

The advantages of using 3D tech vs. 2D tech are numerous. First and foremost, 3D technology is more easily ported among the various platforms. It is resolution independent and can be adjusted to handle the graphics capabilities of high-end gaming PCs, low-end phones, and anything in between. Another huge advantage is this: Most developers use 3D tech for their 3D games already, so for them it's less of a learning curve to adapt that technology for 2D gameplay.

There are, however, some real disadvantages to 3D tech as well. The graphic look may appear less clean, memory usage might be larger, and the graphics processing power needed to adequately display your scenes may not be available on some of the

target game systems. Still, the advantages usually outweigh the disadvantages, especially when targeting consoles or PCs. The choice of 2D vs. 3D is ultimately up to the designer.

It's time to get started. As always, you'll build the playfield first.

VERSION 0.01: THE PLAYFIELD

The plan for this section is to create the playfield in Blender, but first you'll create the Unity project and set up the folder structure. This really should be done with all of your Unity projects so that you have a place to save your assets.

Step 1: Start up Unity and create a project with the name **ClassicScrollingShooter**. Use the **3D Template**.

Step 2: Create the following folders in the Assets panel: **Materials**, **Models**, **Scripts**, **Prefabs, Sounds**.

The Models folder will be used to store our various Blender files. The other folders contain the usual assets. You should now have six folders in the Assets panel, including the Scenes folder that was there already.

Step 3: Rename the SampleScene to **mainscene**. To do this, go to the Scenes folder in Assets, click on SampleScene to select it, click on the name of SampleScene, then type the new name. **Save** and **Exit**.

Next, you'll use Blender to make the terrain for your game. The terrain will consist of a 3D mesh, built using some very powerful features built into Blender.

The following steps will be used to create a section of terrain in Blender. The plan is to create a 2D grid, shape it, and then extrude it into the third dimension.

Step 4: Start **Blender**.

Step 5: Click on the Splash Screen to remove it. Hit the "**Delete**" key and then the "**Enter**" key to remove the default cube. Note to Mac Users: you may need to use the x key instead of the Delete key.

Step 6: Add – Mesh – Grid.

If, perchance, the Grid doesn't appear in the center of the screen, then you might have accidentally moved the cursor away from the center. This is easy to do. To fix this, do Object – Snap – Cursor to Center, then repeat this step.

Step 7: Press the **t** key **twice** to turn the Tools panel off and back on.

That shows you where the Tools panel is, on the left side. At the bottom of the Tools panel you'll see text entry boxes for X Subdivisions and Y Subdivisions for the Grid object.

Step 8: Enter **100** for **X Subdivisions**, **3** for **Y Subdivisions**.

Step 9: Right-click on the new grid, then type **5** and **7** into the numeric keypad. If you don't have a numeric keypad, enable "Emulate Numpad" in User Preferences – Input, which allows you to use numbers on your keyboards instead.

You should now see the Top Ortho view of the Grid object. The text "Top Ortho" is displayed in the top-left corner of the 3D View.

On the numeric keypad, the "5" key switches between the *orthographic* and *perspective* views, the "7" key selects the *top view*, the "1" selects the *front view*, and the "3" key selects the *right view*. These are the bread-and-butter keys in Blender to get to a known view. You can also zoom in and out with the plus and minus keys.

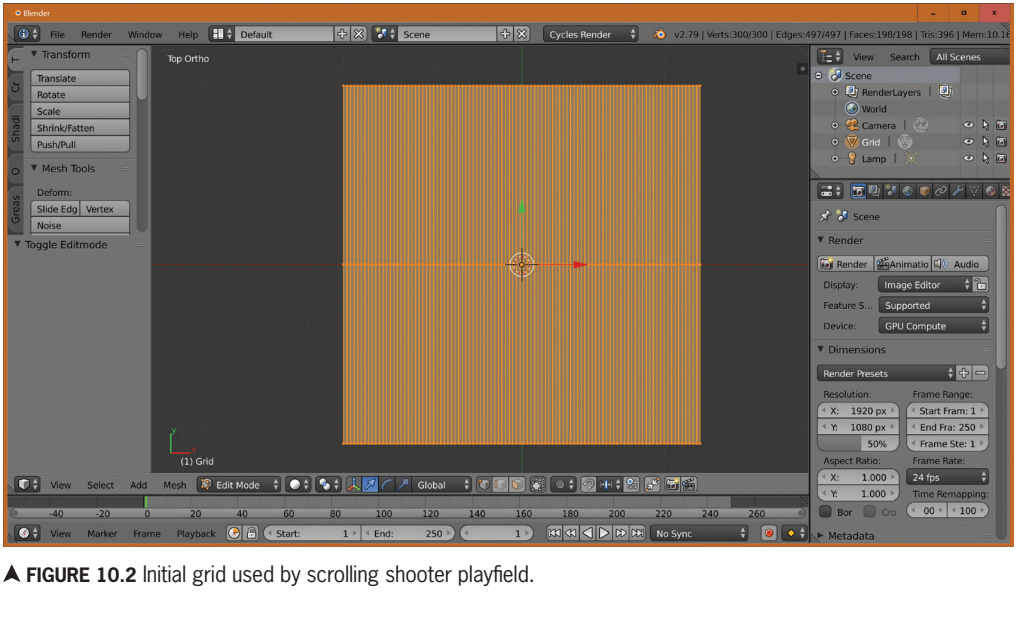
Step 10: Use the **<Tab>** key to enter **Edit mode**.

The <Tab> key toggles between the two major modes of Blender, Edit mode and Object mode. In Edit mode, you have the ability to edit the currently selected object at a low level. In Object mode, you work with multiple objects, create new objects, delete objects, etc.

Step 11: Use the Scroll Wheel on your mouse to **zoom in** on the grid.

Your Blender Screen should look like Figure 10.2. You might have Blender Render selected instead of Cycles Render. That's OK.

Next, you'll delete the unnecessary lower half of the Grid object.



▲ **FIGURE 10.2** Initial grid used by scrolling shooter playfield.

Step 12: Press **a** to deselect everything.

The "a" key flips between selecting and deselecting every part of the Grid object. The orange color highlights the edges of the selected items. The "a" key is very useful and worth remembering.

Step 13: Press **b** to enter Border select mode, also sometimes called box mode. Draw a box around all of the vertices of the bottom edge of the square.

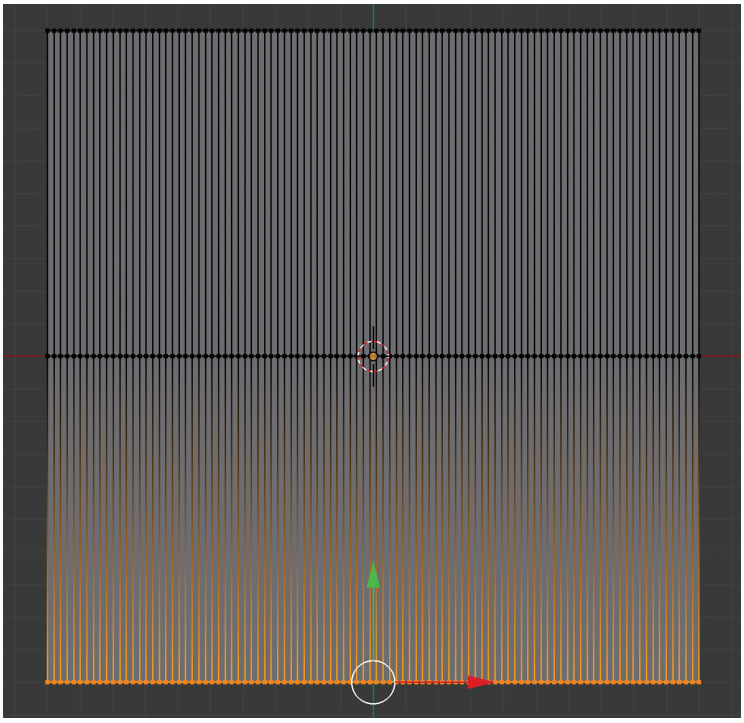
The grid should now look like Figure 10.3.

Box mode lets you select everything inside a box. Your goal is to delete all those vertices, so do this:

Step 14: Press **x** and select **Vertices** to delete all the vertices of the bottom edge.

Not only does this delete the selected vertices, it also deletes all connected edges and faces.

It's time to save your work. After that you're going to experiment with this piece of geometry and put the result into Unity to see what it looks like there. Then you'll get back to this point and start over.



▲ **FIGURE 10.3** Using Box mode to select the bottom edge of the grid.

Step 15: Save the file in **ClassicScrollingShooter/Assets/Models** using the name **BasicGrid.blend**.

Next, we'll enable Proportional Editing to make the top edge look like terrain. Look at Figure 10.4 to find the icon and Figure 10.5 to see the goal, and then do this:

▲ **FIGURE 10.4** Proportional editing icon, circular shape in the middle.

Step 16: Click on the **Proportional editing** icon below the 3D view and **select Enable**.

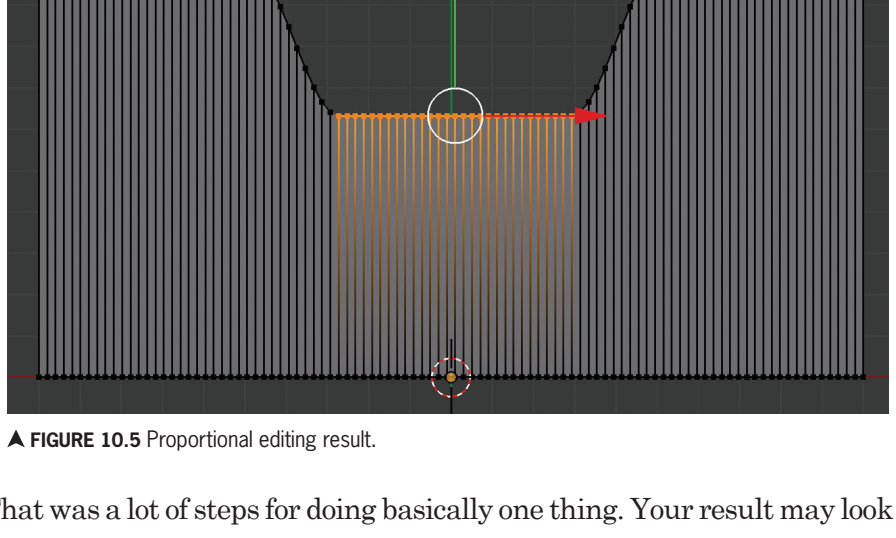
Proportional editing is a feature in Blender which, when enabled, causes nearby vertices, edges, and faces to be affected when you edit something. You'll see this effect in the next few steps.

Step 17: Press **b** to enter Border select mode. Using the mouse and the **left mouse button**, **select** a few vertices from the middle of the **top edge** by dragging.

Step 18: Press **g** to grab the vertices, press **y** to restrict the movement to the y axis.

Step 19: Scroll the mouse **wheel** to adjust the size of the circle. The circle indicates the area of influence for proportional editing and needs to be smaller.

Step 20: Move the **mouse down** a short distance, and then **left-click** to finalize the new vertex positions.



▲ **FIGURE 10.5** Proportional editing result.

That was a lot of steps for doing basically one thing. Your result may look different than the figure, but you're just testing so you don't need to match the figure exactly.

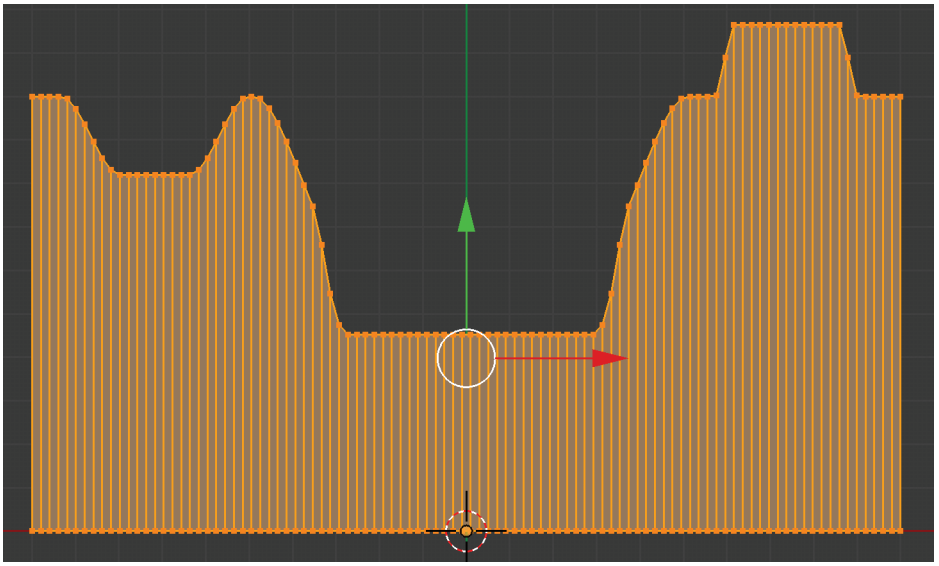
Step 21: Press **a** to deselect the vertices.

Step 22: Repeat Steps 17–21 a few times, using different selections of vertices.

You'll end up with something like Figure 10.6. The next steps turn the distorted grid into a piece of terrain.

Step 23: Press **a** to select all vertices.

If you forgot to do Step 21, you'll need to type "a" a second time. Your goal is to color every vertex and line orange.



▲ **FIGURE 10.6** Distorted grid, the result of multiple proportional edits.

Step 24: Press **1** on your numeric keypad to get a front view.

If you get a blank screen, you'll need to zoom out until you see a horizontal orange line. Then zoom in and pan (with Shift – Middle Mouse Button drag) to center the line.

Step 25: Press **e** to start extruding.

Step 26: Type **0.2** and **<Enter>** to set the amount of extrusion.

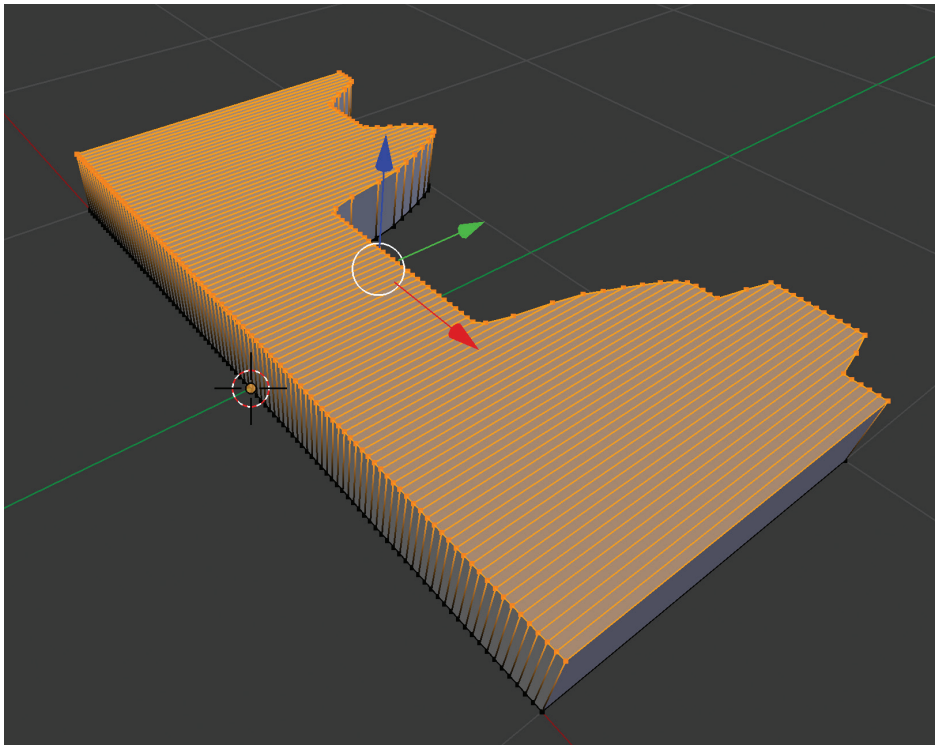
The user interface for this last step may seem a little unusual. You're typing in numbers and they magically show up below the 3D panel at the lower left. When extruding, or doing similar operations, you have a choice of setting the parameter of the operation with the mouse or by typing in numbers.

Step 27: Press **5** on the numeric keypad to get to Front Perspective view.

For this next step, you'll take a closer look at the effect of the extrusion.

Step 28: Press and hold the **middle mouse button**, and while holding that button **move the mouse** to rotate the view to match Figure 10.7. Alternatively, you can type the "6" and "8" keys on the numeric keypad to rotate the view in discrete steps.

The "2" and "4" keys can be used to rotate the view back if you went too far. Another useful technique is to hold the Shift key while dragging the middle mouse button to pan the view.



▲ **FIGURE 10.7** Extrusion.

Step 29: Press **5** and **7** on the numeric keypad to get back to the Top Ortho view.

Step 30: File – Save As… in the **Assets/Models** folder using the name **GridTest**.

Next, you're going to look at your piece of terrain in Unity. You can leave Blender open because we're going back to it in a few steps. Or, you can close it now and load the GridTest file at that time.

Step 31: In Unity, find **GridTest** in the Models folder and drag it into the **Hierarchy** (not the Scene panel) panel.

This is a common situation. We want the object to be placed exactly at (0, 0, 0). This is easier to do by just dragging it into the Hierarchy panel, rather than the Scene

panel. Notice that the X Rotation of GridTest is −90, so make it 0 instead in the following step.

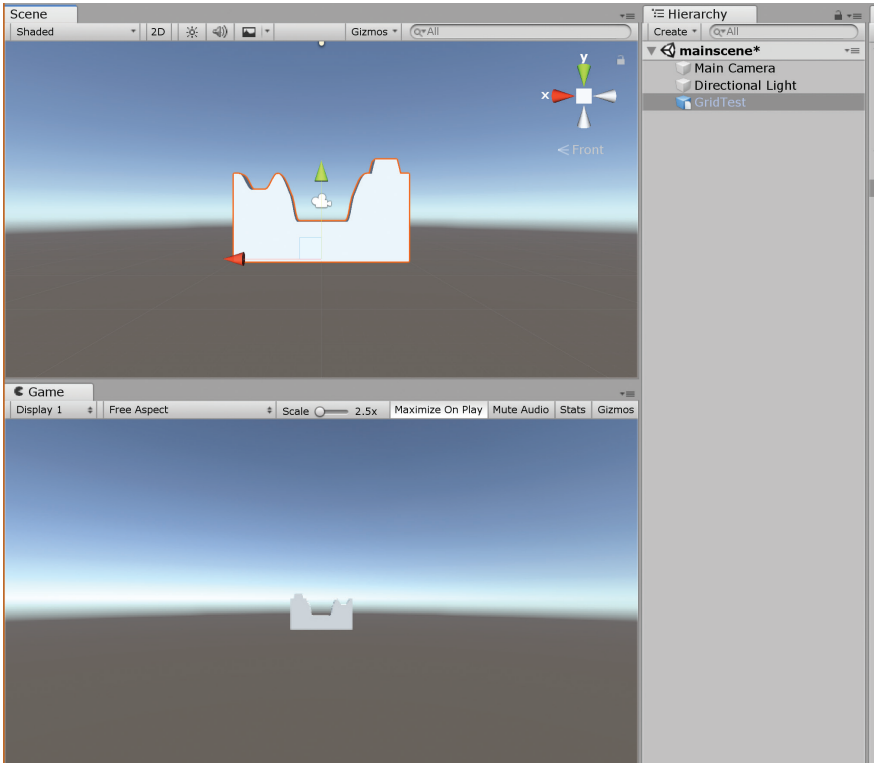
Step 31: Select GridTest and in the Inspector, enter **0** for **Rotation X**.

Step 32: Use a **Front Perspective** view by right-clicking the Scene Gizmo and selecting Front and Perspective.

Remember that clicking on the Gizmo text toggles Perspective and Ortho view in the Scene panel in Unity. This is the equivalent of the "5" key in Blender.

Step 33: Press **f** to focus on **GridTest**.

Your Scene and Game panels should now look like Figure 10.8. The f key only works if the mouse curser is hovering in the Scene panel. Alternatively, double-clicking on the object in the Hierarchy panel has the same effect.



▲ **FIGURE 10.8** GridTest in Unity.

Step 34: Save the scene and project in Unity. Then **exit** Unity.

You're not done with the playfield yet, but this test is a start. You'll complete the construction of your playfield later on in this chapter. You'll use Blender to make several different terrain pieces and assemble them to form a much larger playfield.

Rather than spending time creating the playfield right now in its entirety, you'd like to have some basic gameplay first. So, you'll move on to the next gameplay element. In the next section, you'll be modeling the scrolling spaceship.

VERSION 0.02: SPACESHIP PART 1: MODELING

In this section, you'll be using Blender to make the mesh for the scrolling ship. In the next section, Version 0.03, you'll use Blender's texture painting mode to paint the ship, and after that you'll bring it into Unity. If you wish, you may skip ahead and copy the .blend file from the DVD instead.

The ship starts out as a cube and you'll do some 3D editing to turn it into a spaceship. This technique is called *box modeling*. In box modeling you start with a primitive shape, such as a cube or a cylinder, and proceed to modify your model step by step, gradually converging toward the desired result. If you're new to 3D modeling, this technique is the one to learn first. After a little bit of practice with box modeling, you'll be ready to explore other, more advanced 3D modeling methods on your own.

Step 1a: In Blender, select **File – New**, accept the "Reload Start-Up File" prompt.

Step 1b: Your 3D View panel should show the default starting cube of Blender. If you don't see the starting cube, it's due to a modified start-up file. Do **File – Load Factory Settings** to restore it.

This has the effect of resetting your User Preferences, so review File – User Preferences and manually change any User Preferences you wish to use for this project. For example, if you don't have a numeric keypad on your keyboard, you'll need to check the Emulate Numpad option in the Input section.

Step 1c: File – Save As… with the name **ScrollingShip** in the Assets/Models folder.

Yes, this cube doesn't look anything like a spaceship. It's a good habit to save your work with the intended filename as soon as possible. That way you can just do a quick save later on without having to think of the name.

Step 2: Right-click on the starting cube, toggle into **Edit mode** (with **<Tab>**), and click on **Subdivide** in the Tool panel (on the left). You may need to scroll the Tool panel to see the Subdivide button. It's in the Mesh Tools Add section.

The cube appears to be cut into eight smaller cubes.

Step 3: Type **7** and **5** in the numeric keypad to get into the Top Ortho view.

Step 4: Type **a** to deselect everything.

Step 5: Type **z** to toggle into wireframe mode.

Step 6: Type **b**. Then border-select the **bottom three vertices** and **delete** them. You can use the **x key** to delete. Depending on your keyboard, the Delete key may also work.

Step 7: Type **z** to turn off wireframe mode. Then type **5** on the numpad. This gets a solid perspective mode. Then **hold** the **middle mouse button** and **move** the **mouse**. Then **let go** of the **middle mouse button** when you get a good view of the mesh. Scroll the mouse scroll wheel to zoom in.

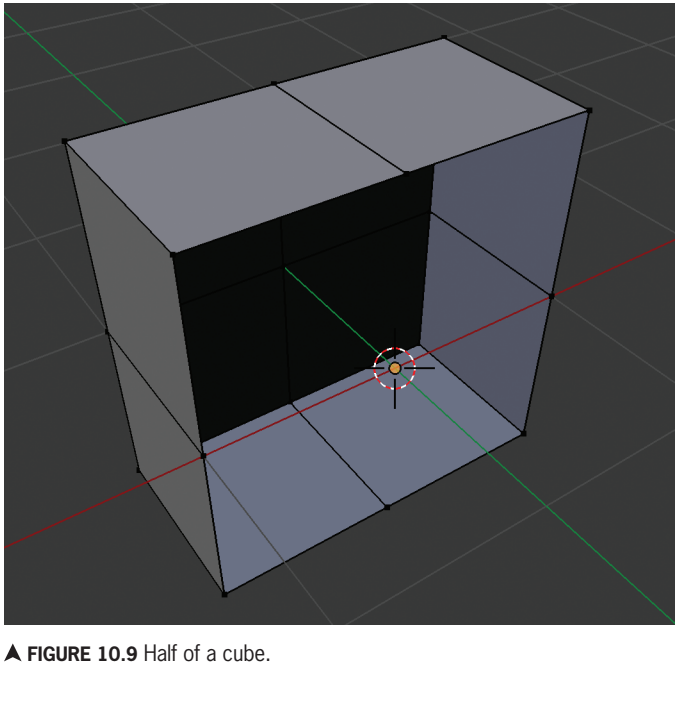
If you don't have a middle mouse button, you can type 2 2 2 4 4 ++ on the numeric keypad to rotate the view and zoom in using your keyboard instead. For Blender it is highly recommended that you use a three-button mouse. These mice are cheap and can be plugged into any PC or Mac, desktop or laptop.

Aim to have your view of the half-cube similar to Figure 10.9.

Step 8: Type **z** to toggle back into wireframe viewport shading.

Step 9: Type **a** to select everything, click on **Subdivide** again, and set the **Number of Cuts** to **2** in the Tool panel.

You just created the basic framework for the spaceship, even though it doesn't look like it just yet.



▲ **FIGURE 10.9** Half of a cube.

Step 10: Click on the **Object Modifiers icon** (the seventh icon, which looks like a wrench) in the Properties panel on the right and click on **Add Modifier – Mirror**.

This didn't appear to do anything yet, but watch what happens next.

Step 11: In the **Axis** section in the Properties panel, **check Y** and **uncheck X**.

You are now using the Mirror modifier along the Y axis. The half of the cube you deleted is now a mirror of the other half.

Step 12: Type **a** twice to make sure everything is selected.

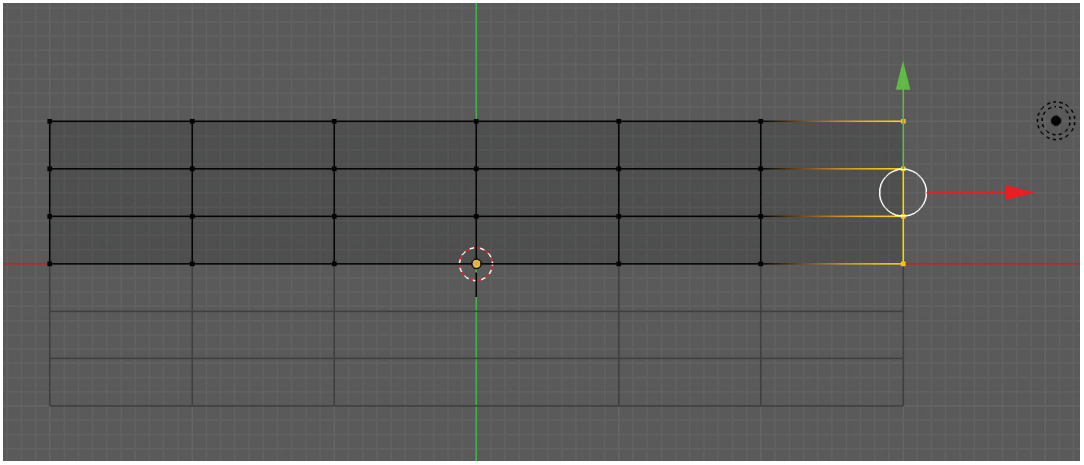
Step 13: Type **s x 3.0 <Enter>** to scale the mesh by a factor of 3 along the x axis.

Step 14: Type **7** and **5** on the numeric keypad to get to Top Ortho view.

Depending on your screen dimensions, you may need to zoom out using the mouse wheel, or the plus and minus keys on your numeric keypad, so you can see the entire mesh. If needed, pan the view to center the mesh. You pan by holding Shift – Middle Mouse Button and then moving the mouse.

Step 15: Type **a** to deselect everything, and then type **b** and border-select the right three vertices.

Your screen mesh should look like Figure 10.10. The cursor should still be at (0,0,0). The cursor is that red and white circle in the middle of the mesh. If you accidentally moved it, which is very easy to do, this is a good time to reset it. Do Mesh – Snap – Cursor to Center.



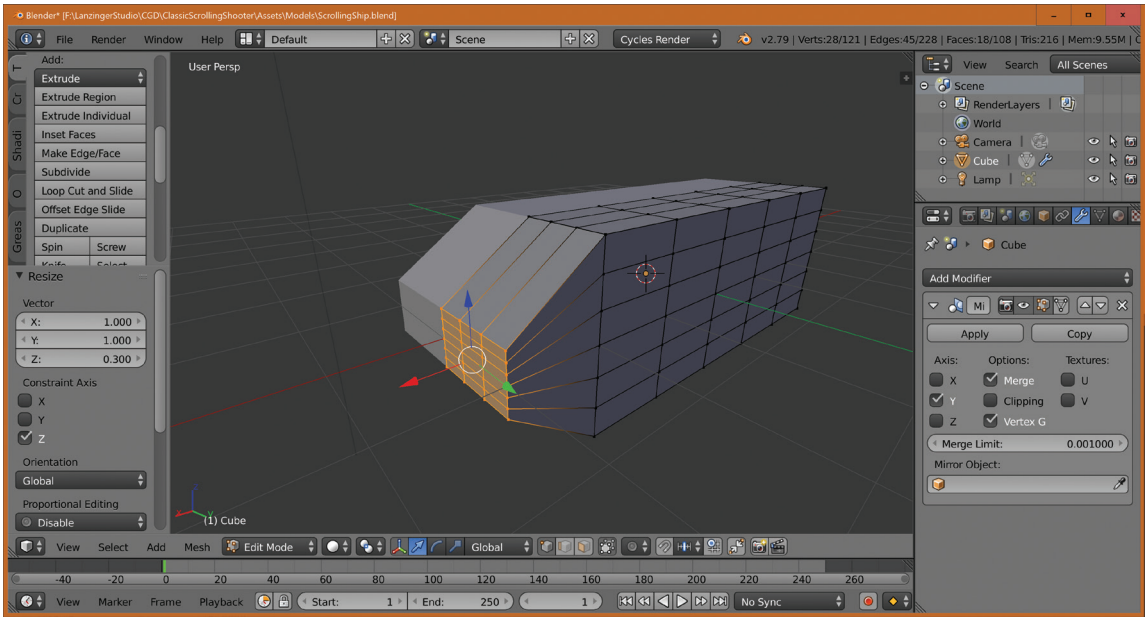
▲ **FIGURE 10.10** Stretched and mirrored cube, Top Ortho view.

Step 16: Type **1** on the numeric keypad to get to the Front Ortho view. Pan and zoom to center if necessary.

Step 17: Type **s z 0.3 <Enter>** to scale the front of our scrolling ship, restricted to the z axis.

Step 18: Type **z** to get Solid Viewport shading, **5** to use perspective view. **Spin and Zoom** the view of the scene until you have the front of the ship facing you, as shown in Figure 10.11.

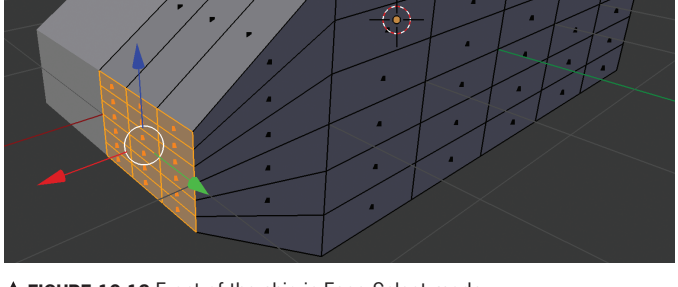
To do the spin, type 8, and then 6 several times, or use the mouse with the middle mouse button as before. You may need to zoom out to see the entire mesh. Again, if necessary, you can pan the view by holding Shift Middle Mouse Button and moving the mouse.



▲ **FIGURE 10.11** Front of the ship.

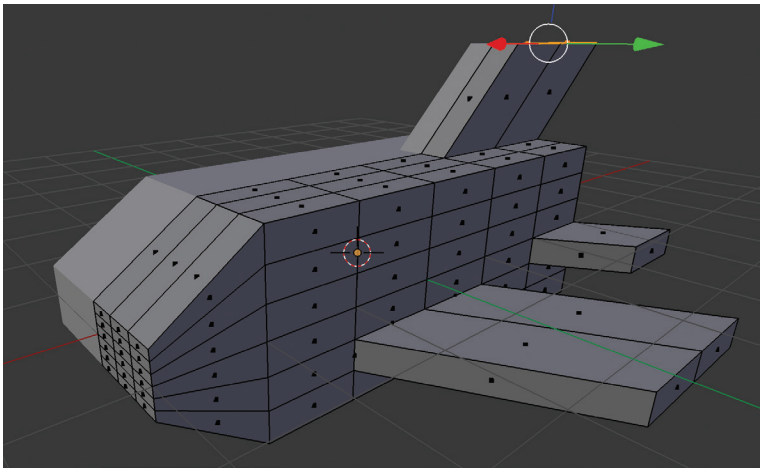
Step 19: Use **Face Select mode** by clicking on the Face Select icon. 

You can find the Face Select icon below the 3D panel. Hovering the mouse over an icon gets a pop-up text description. Notice that the faces now have dots in the center of them. Those are just face indicators that only appear during Edit mode and don't get rendered in the game. Your ship should now look like Figure 10.12.



▲ **FIGURE 10.12** Front of the ship in Face Select mode.

The next steps will create the wings of your spaceship. The plan is to select two faces and pull them away from the fuselage. This is called *extruding*. Take a look at Figure 10.13 to see your goal.



▲ **FIGURE 10.13** The result of extruding.

Step 20: Type **a** to deselect the faces at the nose of the fuselage.

Step 21: Right-click on the side face, second position from the bottom, third position from the front, then **<Shift>-right-click** on the adjacent face farther away from the front. These are the faces where the main wing attaches to the fuselage.

Use Figure 10.13 to help locate these particular faces.

Step 22: Type **e 2**, creating a wing. **Left-click** to stop the extrusion. Instead of typing the 2 you could move the mouse and adjust the amount of the extrusion to your liking.

Amazingly, the wing on the other side of the ship is also there because you still have the Mirror Modifier active. To get a better view, use the 8 and the 2 keys on your numeric keypad.

Step 23: Type **g x −1** to pull the wings away from the front somewhat. **Left-click** to stop the move.

The letter g stands for "grabbing." This is also a very popular modeling command in Blender.

Step 24: Right-click on a single side face at the back of the fuselage, third position from the bottom, and make a wing out of it just like you did in the previous two steps. Use an extrusion distance of 1 and a slant distance of 0.5.

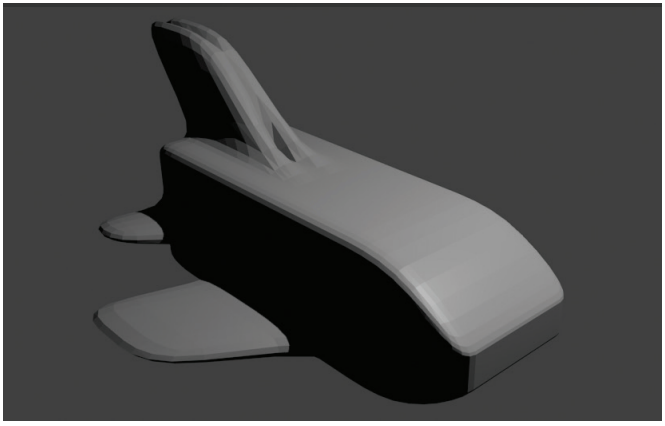
Step 25: Right-click and Shift-right-click on two faces on the top of the fuselage, extrude, and slant back as in the previous steps. Your goal is to create something similar to Figure 10.13.

Step 26: Add Modifier – Subdivision Surface in the Object Modifiers panel. Set the View Subdivisions to 2.

You now have two modifiers active, the Mirror and the Subsurf modifiers. The Object Modifiers panel shows all the currently active modifiers.

Step 27: Select **Blender Render**, then **Render – Render Image**.

Blender supports three renderers. To select Blender Render, look for the engine selector near the top of your window in the middle. Then click on it to choose the renderer. If you are using the Factory default settings, Blender Render will already be selected.



▲ **FIGURE 10.14** Blender Render of the Spaceship.

Your render should look similar to Figure 10.14. Those are beautifully curved wings, and it didn't take much effort at all. When you're done looking at the render, do **Render – Show/Hide Render View** to exit the render view.

Step 27: File – Save. The name of your model should still be **ScrollingShip**.

The mesh for the spaceship is now complete, so it's time to give it a good texture. In the next section, we'll use Blender's texture paint mode to do that.

VERSION 0.03: SPACESHIP PART 2: TEXTURING

Most 3D creation tools such as Blender have a great feature that allows the user to paint directly onto the 3D model using the mouse. The beginning of this section is optional, so if you wish, you can skip to Step 24 and use the unpainted version of ScrollingShip instead. You can also copy the textured version of ScrollingShip from this book's DVD, but that would be cheating. Even if you're not an artist, it can be very educational for you to go through these steps and learn a little bit about the world of 3D modeling and texturing.

Step 1: Load the saved work from the previous section, if necessary.

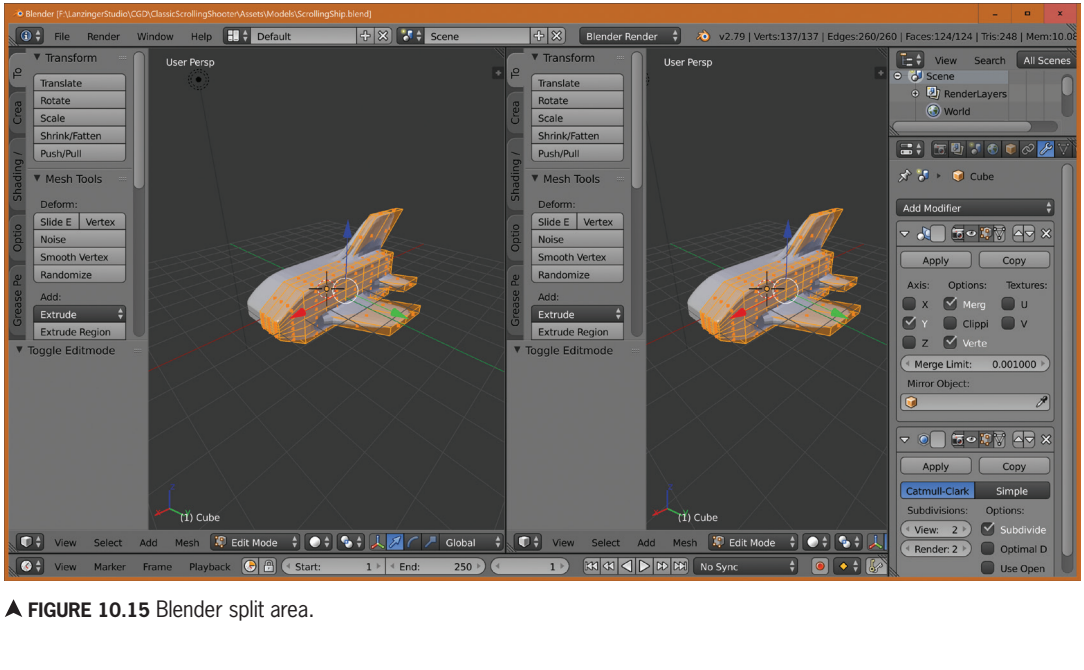
Step 2: Do the **Split Area** command as described below.

This is a simple step, but it requires some explanation for the uninitiated. Carefully move the mouse to the top edge of the 3D view until the mouse icon turns into a vertical double arrow. Then, right-click to bring up the "Area Options" menu. Click on the "Split Area" menu item, move the mouse horizontally to select a balanced split, and left-click to complete the split. Your screen should look like Figure 10.15.

Alternatively, you could have split the area by dragging the lower-left corner of the area like you did previously. Notice that the right view of the ship is clipped in Figure 10.14. This depends on the monitor resolution, so your screen may look a bit different.

Step 3: In the right panel, select the **UV/Image Editor** type.

You select the Editor type by clicking on the shaded cube icon, the one with arrows immediately to the right. There are two of these icons on your screen, one for each of the split areas. They are located in the bottom-left corner of each area. 



▲ **FIGURE 10.15** Blender split area.

You should now see one scrollingship on the left and the render result in the UV/Image Editor. In case you're curious, the UV has nothing to do with ultraviolet rays. It's a naming convention for texture coordinates.

Step 4: Hover the mouse over the spaceship on the left and type **a** to deselect the currently selected faces, if necessary. Then type **a** again to select all faces.

Your goal is to have all the faces appear orange. Some of the faces are partially obscured because of the subdivision modifier. You should verify that you are still in Edit mode and Face Select mode. You can tell that you're in Face Select mode because the faces have dots in the center.

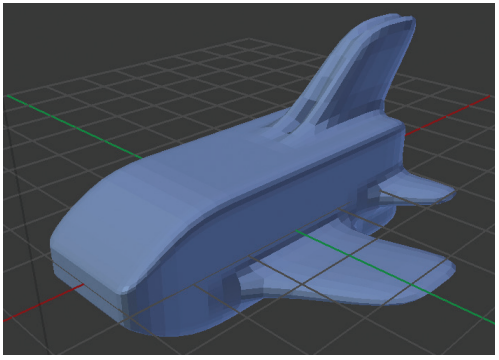
Step 5: Change **Edit** mode to **Texture Paint** in the 3D View.

You'll now see an error message at the top left with the message "Missing Data." There are two things you need to do to fix this. There are missing UVs and a missing Texture.

Step 6: Click on **Add Paint Slot – Diffuse Color**

This sets up a paint slot for texture painting. The default 1024x1024 size is OK. The default black color is not what you want. A light blue color is a better choice for now.

Step 7: Click on **Color**, and use the color dialog to make a light blue color. Uncheck Alpha, as you don't need an alpha channel here, and then click on OK. Your ship now looks like Figure 10.16.



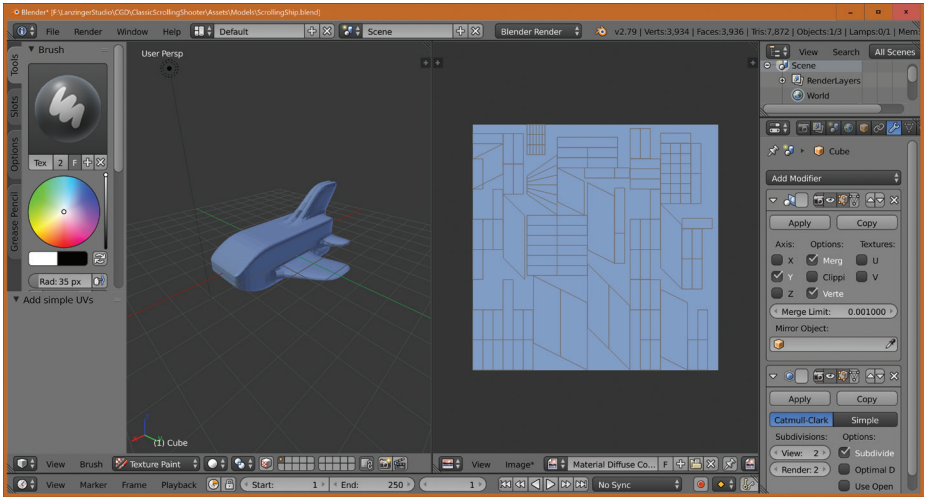
▲ **FIGURE 10.16** Light blue spaceship.

The ship actually appears a bit darker than the base light blue color, but it's good enough for now. You still have missing UVs.

Step 8: Click on **Add simple UVs** in the Tool panel.

Step 9: in the UV Image editor panel, link to **Material Diffuse Color** 

You do this by clicking on the image browse icon at the bottom of the UV Image editor and then selecting the Material Diffuse Color image. Your screen now looks like Figure 10.17.

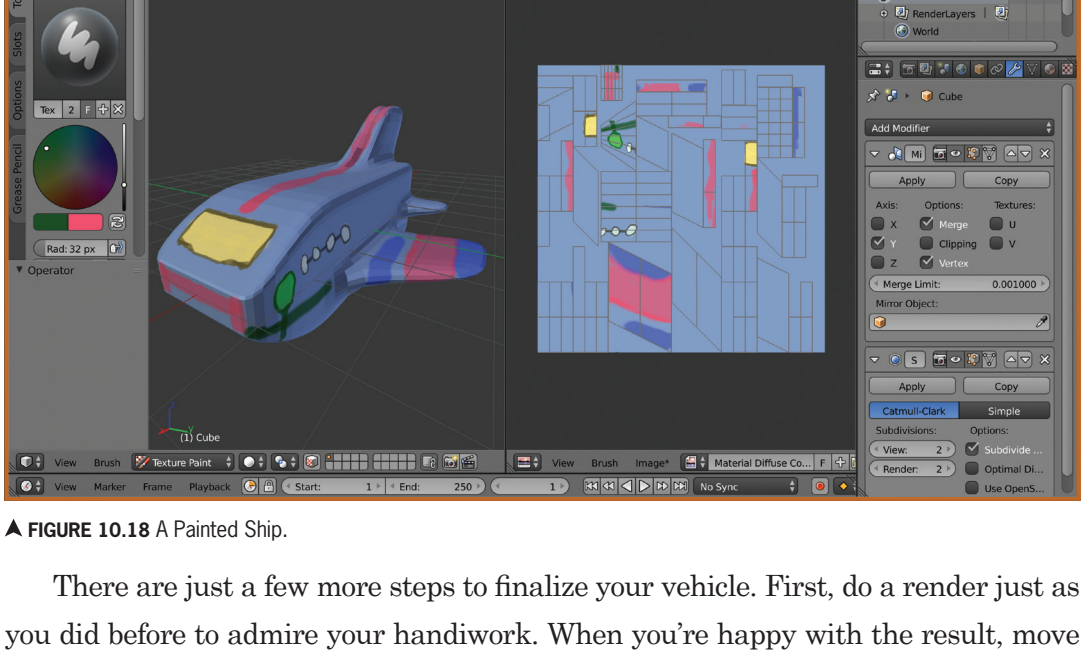


▲ **FIGURE 10.17** UV unwrap in action.

You may need to zoom out a bit by using the scroll wheel to see the entire image. You are now ready to texture paint by selecting colors in the color wheel and drawing on the model with your mouse. Give it a try! You can undo your drawing by hitting Control-z (Command-z on a Mac).

Step 10: Decorate your ship by selecting different colors and painting with them. You can rotate the ship while you're doing this using the middle mouse button or the numpad keys.

Notice that the Mirror Modifier is still in effect, even for texture painting. You can paint in the image editor as well as directly on the 3D model, but you need to change the View mode to Paint mode in the image editor to enable that. You can compare your creation with Figure 10.18.



▲ **FIGURE 10.18** A Painted Ship.

There are just a few more steps to finalize your vehicle. First, do a render just as you did before to admire your handiwork. When you're happy with the result, move on to the next steps.

Step 11: In the UV/Image Editor, click on **Image – Save As Image**. Use the name shiptexture.png and store it in the Models directory.

Step 12: File – Save As with the name ScrollingShipTextured.

You're keeping the old untextured ship, just in case you want to redo the texture painting later on with a fresh start, or if you wish to skip the texturing altogether.

You just finished the textured spaceship. Next, you'll look at it in Unity. Fortunately, this part is going to be very easy.

Step 13: Go to Unity. Load ClassicScrollingShooter.

Step 14: Select the **ScrollingShipTextured** asset in the Models folder.

There might be one or more ScrollingShip assets with white icons there as well. You can ignore them, as they are just the backup files used by Blender.

Step 15: Change the **Normals** from Import to **Calculate** and then click on **Apply**.

This last step has only a minimal effect because of the smooth nature of this model.

Step 16: Drag **ScrollingShipTextured** into the **Hierarchy** panel.

You're not seeing the texture yet.

Step 17: Drag the shiptexture from the Models panel on top of the ScrollingShipTextured object.

The ship is much too large in relation to the playfield. There's a simple remedy:

Step 18: Change the **Scale** from 1 to **0.02** for X, Y, and Z.

Step 19: Change the **Position** to **(0, 1, 0)**.

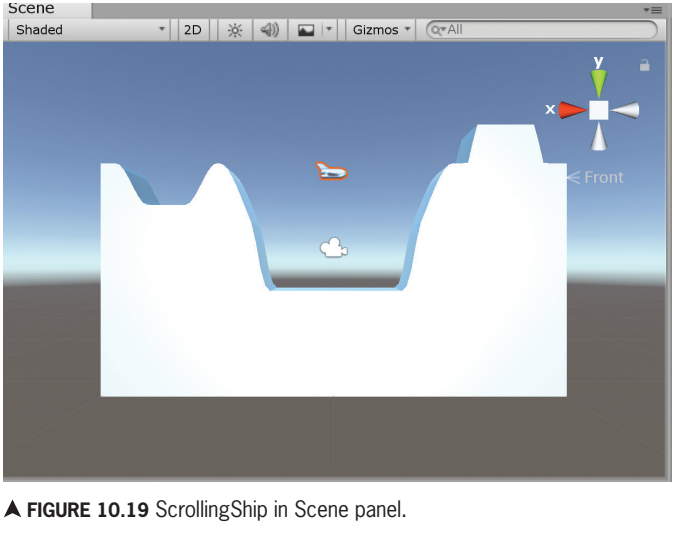
Step 20: Change the Position of GridTest to (0,0,0), if necessary.

Step 21: Change the View in the Scene panel to Front Perspective.

Step 22: Focus on Gridtest and zoom in.

Your Scene panel should look like Figure 10.19.

If your ship overlaps the GridTest mesh, move it up to make it visible.



▲ **FIGURE 10.19** ScrollingShip in Scene panel.

Step 22: Save.

You just learned a useful technique for texturing 3D models. The scrolling spaceship looks much more interesting as a result. Next, you'll make the spaceship fly.

VERSION 0.04: SPACESHIP CONTROL

Controlling the ship can be implemented in a number of ways. You're going to opt for a very simple solution: constant speed in the left-right direction and user control in the up-down direction. First, though, you need to set up the camera and lighting so you can see what you're doing.

Step 1: Select **Main Camera** and set **Position** to **(0, 1, 1.3)**, **Rotation** to **(0, −180, 0)**.

Step 2: In the Scene panel, continue to use **Front Perspective View**.

Step 3: Select **Directional Light and** set the **Rotation** to **(30, 0, 45)** and **Position to (0, 0, 0)**.

The position has no effect on the game itself with a directional light, but it does determine the location of the associated gizmo. The next step will allow you to control the size of the gizmos.

Step 4: Click on **Gizmos** in the **Scene** panel and adjust the size by sliding the **size slider** at the top right of the Gizmo panel.

This step is cosmetic, but it allows you to control the appearance of gizmos in the Scene panel. It's useful to know about this to avoid clutter and giant gizmos.

Speaking of cosmetic changes, this next one is truly remarkable and really improves the appearance of your game. You're going to add a better *skybox*. Skyboxes are a common and easy technique for making 3D games look realistic. Rather than creating and rendering individual objects that are far away, such as clouds, mountains, or thousands of trees, a few large textures are displayed in the background. It's called a box because the texture is pasted on the inside of a very large box so that no matter where the camera is pointing, there's always a visible background texture.

Your project currently uses the default skybox. That blue sky in the background with a slightly curved horizon is the skybox. To get a better one you're going to download one from the asset store.

Step 5: Account – Go to Account

You should see your basic account information in a new browser window. This would be a good time to review your account and make any changes you wish to make. When you're done, you may wish to close the browser window.

Step 6: Use the **Tall layout**.

This prepares you to get a better view of the Asset Store. You were probably using the 2 by 3 layout. To switch to the Tall layout, click on the Layout selector in the upper right corner of your window and select it.

Step 7: Window – Asset Store

This should open a view of the Unity Asset Store where the Scene panel used to be. You'll need to be connected to the internet for this to work.

Step 8: Search for TGU Skybox in the Asset Store panel and select the TGU Skybox Pack.

You may use another Skybox if you wish.

Step 9: Click on the Skybox, click on Import, and then Import again in the popup Import Unity Package window. You may see "Download" displayed instead of the first Import.

This skybox pack contains four skyboxes. You'll only use one of them, Nostalgia 1. When you're done installing this skybox pack it appears in the Asset panel as a folder with the name TGU Skybox Pack.

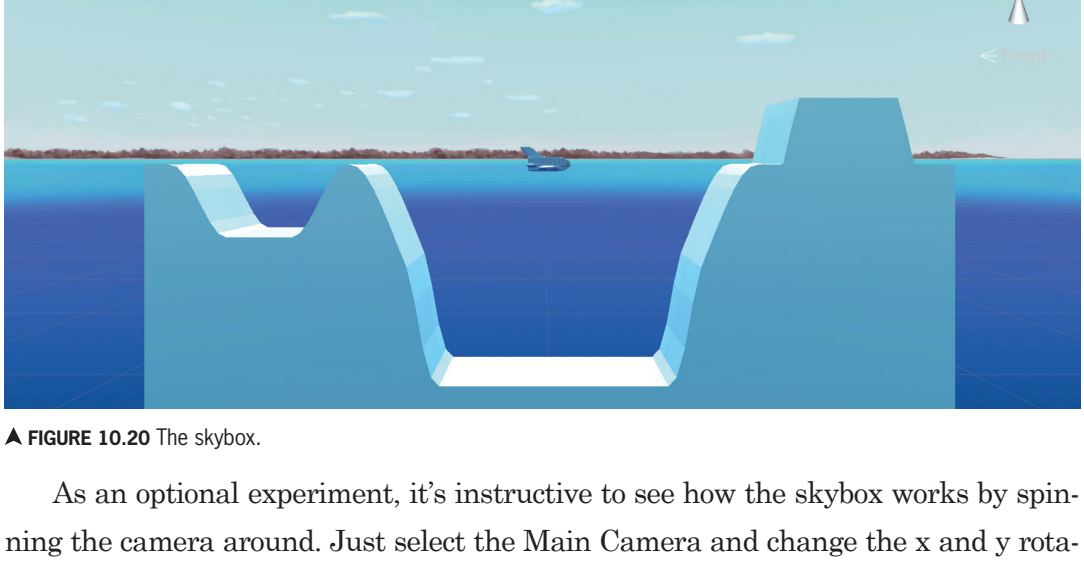
Step 10: Select the 2 by 3 layout, and use the Front Perspective View in the Scene panel.

Step 11: Window – Rendering – Lighting Settings

This opens a Lighting popup window which will allow you to change the skybox.

Step 12: Select the Nostalgia 1 skybox as the Skybox Material.

Your Scene panel now looks similar to Figure 10.20.



▲ **FIGURE 10.20** The skybox.

As an optional experiment, it's instructive to see how the skybox works by spinning the camera around. Just select the Main Camera and change the x and y rotation coordinates. Return them to (0, −180, 0) when you're done looking around.

Now that your scene is looking presentable, it's time to make the scrolling ship move.

Step 13: Assign the following code to **ScrollingShipTextured**, with the name **scrollingship**, and put the **scrollingship** script into the **Scripts** folder.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class scrollingship : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(-0.3f * Time.deltaTime, 0, 0);
    }
}
```

If you run the game now, the ship scrolls off the screen, never to be seen again. It's time to have the camera follow the moving ship.

Step 14: Create the script **camera** in the Scripts folder and assign it to **Main Camera**. Use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class camera : MonoBehaviour
```

```
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        GameObject player = GameObject.Find("ScrollingShipTextured");

        transform.position = new Vector3(
            player.transform.position.x,
            transform.position.y,
            transform.position.z
            );
    }
}
```

When you run the game now, the camera follows the moving ship. The code takes the current x-coordinate from the ship and uses that as the x-coordinate of the camera.

Next, you'll add some simple up and down controls to the ship.

Step 15: Insert the following code into the Update function of scrollingship.cs:

```
if (Input.GetKey("w"))
{
    transform.Translate(0, 0, 0.8f * Time.deltaTime);
}
if (Input.GetKey("s"))
{
    transform.Translate(0, 0, -0.8f * Time.deltaTime);
}
```

This code moves the ship up or down depending on key presses by the player.

Step 16: Test and Save

VERSION 0.05: LEVEL 1

The playfield is due for an expansion. You're going to go back to Blender and make several grid pieces similar to GridTest. Then, you'll assemble copies of them into a long strip, and join them all together into a single mesh consisting of several thousand faces.

As a historical note, this method of making a playfield would have been very foreign to game developers in the '80s. Instead, playfields were created using stamps. Each stamp was typically an 8 x 8 or 16 x 16 square. The stamps were laboriously drawn pixel by pixel, often with a limited color palette. The stamps would then be assembled using a stamp map. In a way, we're doing a similar thing here, just using 3D faces instead of pixels.

Step 1: In Blender, do **File** – **Open Recent** – **BasicGrid.blend**.

Step 2: Zoom out using the **Mouse scroll wheel** or the **numpad minus** key.

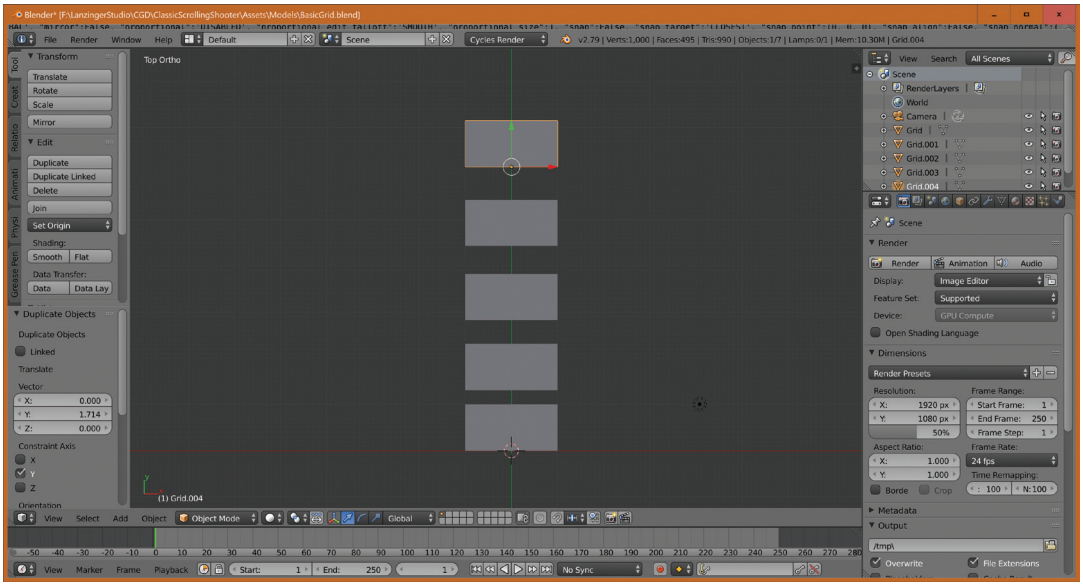
Step 3: Type the **<Tab>** key to enter Object mode.

Step 4: Type **a** to deselect everything.

Step 5: Right-click on the Grid object to select it.

Step 6: Type **<Shift> d** and then **y**, move the mouse up, and left-click to place the new copy of the Grid object.

Step 7: Repeat Step 6 four more times until you have a total of five Grid objects, stacked vertically, as shown in Figure 10.21. You will need to zoom out to see what you're doing. The pieces don't need to be spaced evenly. To match the view in Figure 10.21 you'll need to pan the camera by doing Numpad 7 followed by <Shift> Middle Mouse Button and move the mouse to center the five pieces. You'll probably want to zoom in after that.



▲ **FIGURE 10.21** Setting up Grid pieces.

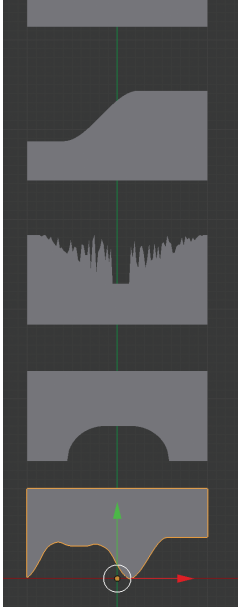
In the next step, you'll be using proportional editing as described in the very beginning of this chapter. Don't do the extrusion step yet because we'll be doing that later. You may wish to try the different falloff types, such as Smooth, Random, and Root. The falloff types are set in a menu immediately to the right of the proportional editing mode icon.

Step 8: Use Proportional Editing to create a collection of Grid pieces similar to Figure 10.22.

You'll need to go into Object mode, select the piece that you're editing with a right-click, and then go back to Edit mode for each of the pieces.

Step 9: Save the file with the name **GridPieces.blend**.

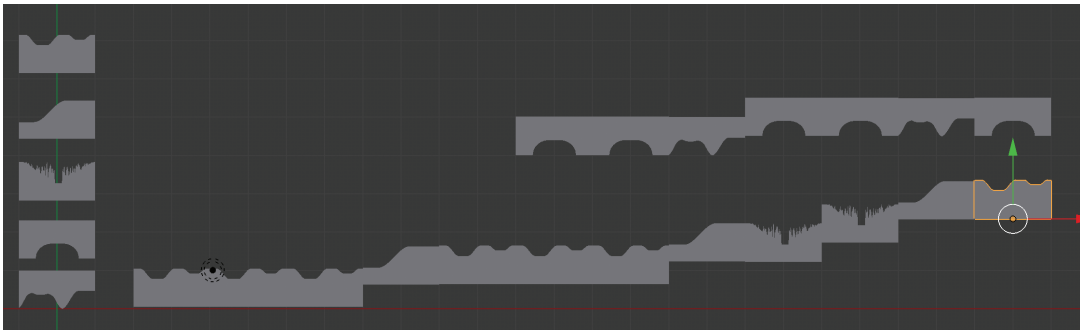
Now that you have a collection of playfield pieces, you'll assemble them.



▲ **FIGURE 10.22** Completed Grid pieces.

Step 10: Use **<Shift>-D** to make copies of the pieces and assemble them like in Figure 10.23. Turn on Snap to help line up the pieces. There are 12 pieces at the bottom and 7 at the top.

The Snap icon looks like a magnet and is near the right side of the icon strip at the bottom. To the right of the Snap icon is a setting for the type of element to snap to. Make sure that setting is set to "Increment." To the right of that is something called "Absolute Grid Alignment." That needs to be turned on. When you grab and move a grid piece with these settings the piece snaps to the grid and it becomes easier to align the pieces with one another. For finer control you can zoom in and the grid becomes more detailed.



▲ **FIGURE 10.23** Playfield layout.

Step 11: Save As with name **Level_1_layout.blend**

This isn't the completed level yet, but it's a good idea to save at this point. If you want to change the level later on this would be a good point to make changes.

Step 12: Delete the original pieces in the vertical stack.

Step 13b: In Object mode, **select** all pieces of the playfield, and do **Object - Join**.

Step 13: Select the playfield, go into **Edit** mode, and **select** all vertices using **a**.

Step 14: Type **1** to go into Front Ortho view.

Step 15: Type **e 0.3 <Enter>** to extrude by 0.3 units.

Step 16: Type **7** to go into Top Ortho view.

Step 17: File – Save As… using the name **level_1.blend**.

You just completed making the mesh for the Level 1 playfield.

Step 18: In Unity, select **Level_1**.

Step 19: In the Inspector, set **Normals** to **Calculate**.

Step 20: In Animations, **disable Import Animation** and **Apply**.

Step 21: Drag **Level_1** into the **Hierarchy panel**.

Step 22: Set **Position** to **(−10.4, 4, 0)** and **Rotation** to **(0, 0, 0)**.

Step 23: Create a Purple Material with Smoothness of 0.9 and assign it to Level_1.

Step 24: In the Hierarchy, **delete GridTest**.

As always, you'll do some testing of the new playfield.

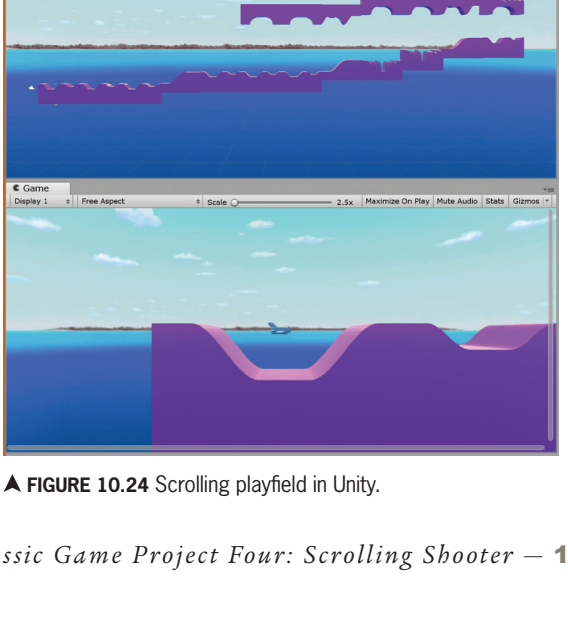
Step 25: Select Level_1 in the Hierarchy panel, and focus on it in the Scene panel using the **f** key. Use the mouse scroll wheel to zoom in on the playfield.

Step 26: Select ScrollingShipTextured in the Hierarchy panel.

Step 27: Save.

Your Scene and Game panels should now look like Figure 10.24. If the playfield is missing in the Game view, adjust the position x and y coordinates so that it does.

This playfield is relatively small, but it's large enough for development purposes. If you play the game now, you'll see that you have a problem with the camera not moving up and down with the ship. This will be fixed in the next section.



▲ **FIGURE 10.24** Scrolling playfield in Unity.

VERSION 0.06: ROCKETS

You need something for our scrolling ship to shoot at, and later on, you'll be making shots to be launched by the ship. In this section, you'll create the rockets that rise up as defensive weapons for the playfield. First though, you'll improve the camera so you can have a better view of what's happening.

Step 1: In camera.cs, replace the Update function with the following code:

```
void Update () {
    GameObject player = GameObject.Find("ScrollingShipTextured");

    float xpos = player.transform.position.x;
    float ypos = player.transform.position.y;

    float new_ypos = transform.position.y;
    if (new_ypos < ypos - 0.5f) new_ypos = ypos - 0.5f;
    if (new_ypos > ypos + 0.5f) new_ypos = ypos + 0.5f;

    transform.position =
        new Vector3(
        xpos - 0.7f,
        new_ypos,
        transform.position.z
        );

}
```

Step 2: Test and **Save**

This code puts the camera to the left of center and follows the ship up and down if the ship y position is more than 0.5 units away from the camera y position. Give it a try and move the ship up and down using the w and s keys.

Next, you'll build a rocket mesh in Blender.

Step 3: In Blender, **File – New**, and delete the default cube, then save as **rocket.blend**.

Step 4: Add – Mesh – Cylinder with **24** Vertices and Depth of **8**.

The settings for the cylinder are in the Tool panel on the left. You may need to scroll the Tool panel to find the cylinder settings.

Step 5: Move the mouse back into the 3D view panel, then **type g z 4 <Enter>**.

This moves the cylinder up so that the base is at an elevation of 0. The 4 was chosen because it is half the depth of the cylinder.

Step 6: Press **<Tab>** to enter Edit mode and zoom out so you can see the entire cylinder.

Step 7: Click on **Loop Cut and Slide** in the Mesh Tools panel in the Add section, move the mouse over the cylinder, **scroll** the mouse wheel until you see **four rings**, then **left-click**, **slide** the rings down a little, then **left-click** again to finish this operation.

The cylinder should now look like Figure 10.25.

This was a relatively quick way to chop up the cylinder into a mesh that you can now turn into a rocket.

Step 8: Type **1** and **5** on the numeric keyboard to get to Front Ortho view.

Step 9: Use **Wireframe** Viewport Shading.

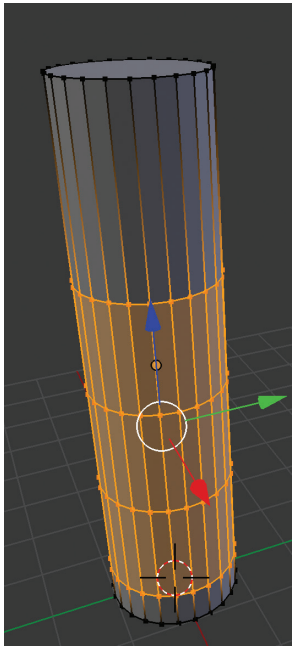
Step 10a: Deselect everything using the **a** key.

Step 10b: Use **b** to select the top vertices of the cylinder, then type **s 0.3 <Enter>**.

Step 11: Repeat Steps 10a and 10b for the next two rings with a scale factor of 0.5.

You just created the basic shape for the rocket. Next, you'll add the fins.

Step 12: Use **Solid Viewport shading**.



▲ **FIGURE 10.25** Result of a loop cut and slide.

Step 13: Use **Face Select** mode.

Step 14: Right-click on the face immediately to the left of center at the bottom.

Step 15: Type e 1 <Enter>.

Step 16: Type **numpad 6** six times.

Step 17: Repeat Steps 14–16 three times.

Step 18: Type **numpad 7**.

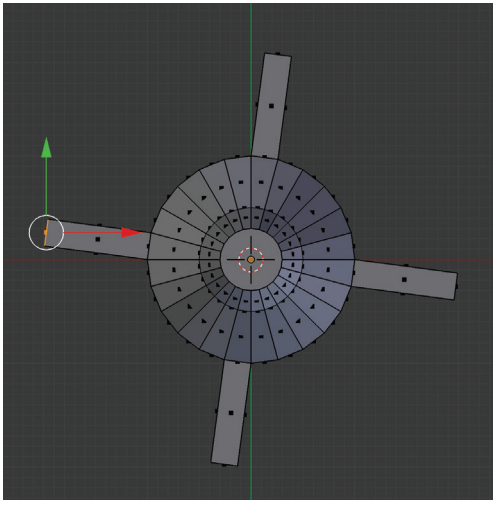
You're now looking at the rocket from the top with the four fins clearly visible as shown in Figure 10.26.

You'll do just one last tweak to the model in the following steps.

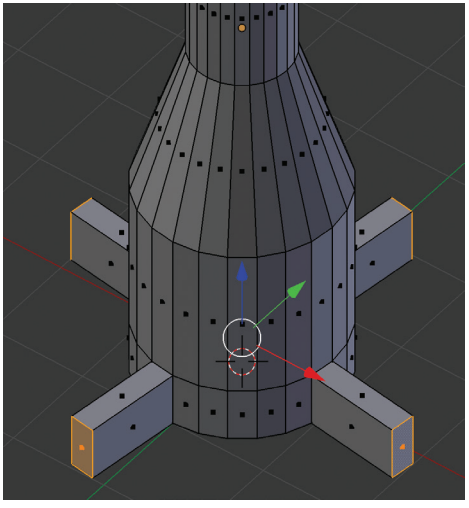
Step 19: Type **numpad 1**.

Step 20: Type **numpad 6** repeatedly to spin the rocket. While doing so, select all four outer faces of the fins using **right-click** for the first one and **<Shift> – right-click** for the other three.

Step 21: Hold the **middle mouse button** and **move** the **mouse** to look at the rocket as shown in Figure 10.27. Alternately, use the numpad 2-4-6-8 buttons to adjust your view.



▲ **FIGURE 10.26** Top view of rocket in Face Select mode.



▲ **FIGURE 10.27** Highlights on four outer faces of rocket fins.

Step 22: Type **g z −1 <Enter>**.

Step 23: Type **numpad 1**.

Step 24: File-Save.

The rocket mesh is now complete. This is what's called a low-poly model. Modern games use thousands of polygons to make very detailed meshes for game objects. In these classic game projects, you're going to be content with relatively simple meshes.

Step 25: In Unity, select the rocket model in the models folder.

Step 26: Turn off Animations

Step 27: Set Normals to Calculate.

Step 28: Drag the rocket model into the Hierarchy panel and select the rocket.

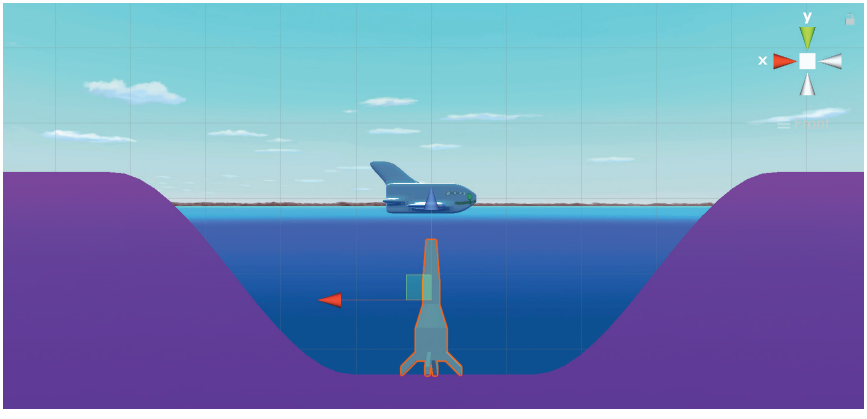
Step 29: Set **Position** to **(0, 1, 0)**, **Scale (0.02, 0.02, 0.02)**.

Step 30: In the Scene panel, select the Front Ortho view using the View Gizmo.

Step 31: Focus on the rocket using the **f** key.

Step 32: Zoom out using the mouse wheel, select the four-arrow Move icon at the top left next to the hand icon, and drag the rocket up and down with the yellow arrow to line it up with the playfield.

The Scene panel should now look similar to Figure 10.28.



▲ **FIGURE 10.28** Rocket on the playfield.

The Rocket in the Game panel doesn't look quite right because the playfield isn't centered correctly. You'll fix that in the next step.

Step 33: Select Level_1 and set the **Z Position** to **−0.15**.

This was a subtle change, but it adds to the look of the scene. The rocket and spaceship are now lined up with the center of the playfield geometry. The 0.15 was calculated by taking the depth of the playfield (0.3) and dividing it by 2. The rocket needs its own material, so you'll use the usual method to create one.

Step 34: In the **Materials folder**, create a new Material with name **RocketMat**, set the **Shader** to **Standard**, Albedo Color to a **bright red**, Smoothness 0.9, and **drag** the material onto the **rocket** object in the Hierarchy.

The rocket is just sitting there, and there's only one rocket. In the next section, you'll create many duplicates of the rocket and make them fly.

VERSION 0.07: FLYING ROCKETS

The plan for this section is to make a prefab out of the rocket, place many rockets on the playfield, and add code that makes them fly. After all of that, you'll start with collision detection between rockets and the scrolling spaceship.

Step 1: Check that you have a Prefabs folder.

Step 2: Drag the rocket object from the Hierarchy to the Prefabs folder.

This is the quick and easy way of creating a prefab. You can now remove the original rocket.

Step 3: Delete the rocket object in the Hierarchy panel. Then drag the rocket from the Prefabs folder back into the Hierarchy panel.

This step doesn't appear to change anything, but it does have an important side effect. The rocket in the Hierarchy is now an instance of the rocket prefab in the Prefabs folder. This allows you to make wholesale changes to all rockets by just changing the prefab.

In the next few steps, you'll make copies of the rocket and put them on the playfield.

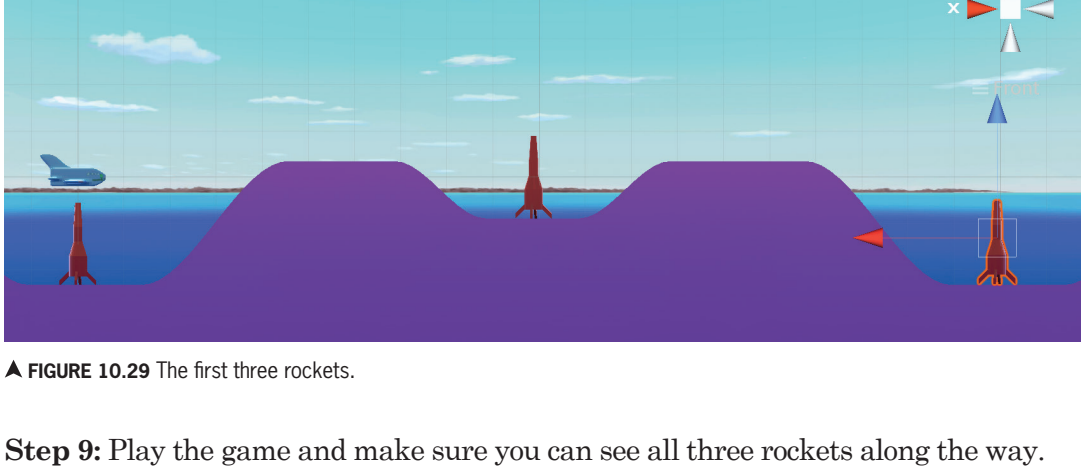
Step 4: Select the **rocket** in the Hierarchy panel.

Step 5: Edit – Duplicate.

Step 6: Select the Move icon (the one next to the hand icon) and grab the red arrow handle on the rocket to move it to the right a little.

Step 7: Zoom out, using the mouse scroll wheel, if necessary, and move the duplicate rocket to the next valley.

Step 8: Repeat Step 7 so that you have three rockets set up, similar to Figure 10.29.



▲ **FIGURE 10.29** The first three rockets.

Step 9: Play the game and make sure you can see all three rockets along the way.

Step 10: Stop playing the game.

Those rockets need to start flying, so you'll write a short script to make that happen. As an optional exercise, try to write a script that makes all the rockets fly straight up. Then compare it to the version in this next step.

Step 11: Create the script **rocket.cs** with the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class rocket : MonoBehaviour
```

```
{
    public float rocketspeed;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(0, 0, rocketspeed * Time.deltaTime);
    }
}
```

Step 12: Select the rocket prefab and use the Add Component box to add the rocket.cs component to it.

Step 13: Set rocketspeed in the rocket prefab to 0.5.

If you play the game right now, you'll see the rockets taking off. You have the basic motion working, but it would be wrong to have the rockets launch all at once at the beginning of the game. They need to wait until the ship is approaching before they launch. The following new code for rocket.cs fixes that.

Step 14: Replace the Update function in rocket.cs with the following code:

```
void Update()
{
    GameObject player = GameObject.Find("ScrollingShipTextured");
    if (player.transform.position.x - transform.position.x < 0.5f)
    {
        transform.Translate(0, 0, rocketspeed * Time.deltaTime);
    }
}
```

The if-statement ensures that ScrollingShipTextured is within 0.5 units to the left of the rocket for the rocket to move.

Step 15: Pick the second rocket from the left and change its rocketspeed to 1.0. Test this.

This shows the power of prefabs. You can override the rocketspeed property of an individual rocket, which is still inheriting the other properties of the prefab rocket.

Next, you'll add a particle system to simulate the exhaust of the rocket. Unity makes this easy.

Step 16: Click on GameObject – Effects – Particle System.

Step 17: Change the **Position** to (**0**, **1**, **0**). Set the **Size** to (**1**, **1**, **1**) if necessary.

You now have a particle system with the default settings located at (0, 1, 0). You now need to change the settings to simulate a rocket exhaust pointing down.

Step 18: Set **Duration 1.0**, **Start Lifetime 0.7**, **Start Speed −0.1**, **Start Size 0.05**, **uncheck Shape**, set the **Start Color** to a **bright orange**.

Feel free to experiment with the Particle settings. Particle systems are cosmetic special effects and usually don't affect gameplay directly.

The next steps line up our particle system with the leftmost rocket.

Step 19: Select the leftmost rocket, and Control-select (on a Mac it's Command-select) the Particle System so that both the rocket and the Particle System are selected. Focus on both using the **f** key in the Scene panel.

Step 20: Select the Particle System and move it to just below the bottom of the rocket in the Scene panel.

Step 21: In the Hierarchy panel, **drag the Particle System on top of the rocket**.

Test the game to see that the first rocket now flies with an orange exhaust trailing after it. You'll see that there is something happening, but the particle system is much too large. Something happened to the scale. This can be fixed in the following step.

Step 22: Change the **Scaling Mode** of the **Particle System** to **Hierarchy**.

When you test this again, you'll see that it's now working correctly.

Unfortunately, the Prefab for the rocket doesn't have the Particle System. This is easily remedied in the next step:

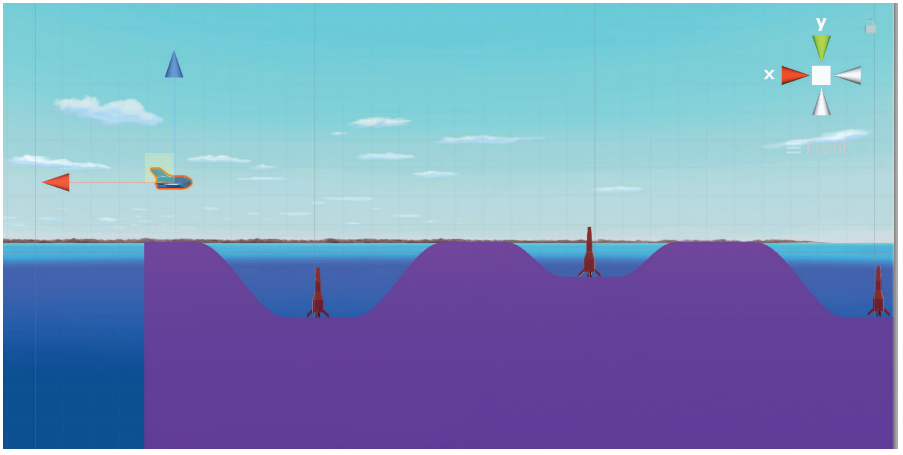
Step 23: Select rocket, click on **Override – Apply All** in the Inspector panel.

This has the effect of applying the change you made to the one rocket instance to all the instances of the rocket Prefab. You should now see a Particle System attached to the other rockets.

This would be a good time to look at short Unity video about prefabs. Go to *https://unity3d.com/learn/tutorials* and click on *Interface & Essentials* followed by *Prefabs – Concept & Usage*.

Next, you'll move the starting position of the ship. Right now, the ship starts on top of the first rocket, which is probably a poor design choice.

Step 24: Move the `ScrollingShipTextured` object to the left and up as shown in Figure 10.30.



▲ **FIGURE 10.30** Scrolling Ship starts at the left.

This is still not ideal, but it's good enough for now. The new starting position of the ship is about at (1,1,0). Your coordinates may be different depending on your design of the playfield. The final version should have the ship start with a few seconds of scrolling and no enemies to give the player a chance to get oriented before the shooting starts.

Step 25: Test and **Save**.

Next, you'll add collision detection between the ship and the rockets. What should happen when a rocket collides with the ship? For now, you'll simply destroy the ship and the rocket. You'll be using tags, just as in your previous projects.

Step 26: Insert the following code into the rocket class in rocket.cs:

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "scrollingship")
    {
        Destroy(gameObject);
        Destroy(other.gameObject);
    }
}
```

Don't forget to save your editing in Visual Studio. To enable this code to work you need to put in colliders and tags.

Step 27: Create a new tag with the name "scrollingship" and tag the ScrollingShip-Textured object with scrollingship.

Step 28: Add a box collider component to ScrollingShipTextured, and check Is Trigger.

Step 29: Add a rigidbody component to the rocket prefab, and uncheck Use Gravity.

Step 30: Add a box collider component to the rocket prefab, and check Is Trigger.

Step 31: If necessary, move the ship away from the first rocket so they don't collide right away.

Step 32: Test the game.

Step 33: Save.

You now have something resembling gameplay. The player needs to avoid the rockets or else it's game over. The rockets are flying and have a particle system that displays their exhaust. Before you go on to add more features, it's time to fix a bug

that you might have noticed when testing the rocket vs. ship collision. There is an error message with the following text:

```
NullReferenceException: Object reference not set to an instance
of an object
```

This message appears immediately after the ship collides with a rocket. What's going on here and how would you fix it? The first step is to open the console window with Window – General – Console. There you'll see many instances of this error message, plus other hints:

```
camera.Update() (at Assets/Scripts/camera.cs:18)
rocket.Update() (at Assets/Scripts/rocket.cs:18)
```

In these messages you see the error message is pointing at line number 18 in the `camera.cs` file as well as line number 18 in `rocket.cs`. When you look at Visual Studio's display of these files, you'll see line numbers at the left. Line numbers 17 and 18 in `rocket.cs` are this:

```
GameObject player = GameObject.Find("ScrollingShipTextured");
if (player.transform.position.x - transform.position.x < 0.5f)
```

When you think about what's happening, the variable `player` is pointing at the ship, but when the ship is destroyed the variable is set to null because the Find function won't be able to find the ship. This wouldn't be a problem if we had only one rocket, but with multiple rockets out there, the rockets that are still out there need this code to decide when to launch.

The real culprit here is a poor programming practice that has crept into your code. The Find function may or may not find what it's looking for. If it doesn't, the code needs to do something reasonable.

Step 34: Replace lines 17 and 18 in rocket.cs with this code:

```
GameObject player = GameObject.Find("ScrollingShipTextured");
if (player)
if (player.transform.position.x - transform.position.x < 0.5f)
```

All you did was insert the `if(player)` line to check that `player` isn't null before using it. You also need to fix the camera file in a similar manner. You'll need to group the code after the null check using curly brackets.

Step 35: Replace the **Update** function in **`camera.cs`** with this code:

```
void Update()
{
   GameObject player = GameObject.Find("ScrollingShipTextured");
   if(player)
   {

      float xpos = player.transform.position.x;
      float ypos = player.transform.position.y;

      float new_ypos = transform.position.y;
      if (new_ypos < ypos - 0.5f) new_ypos = ypos - 0.5f;
      if (new_ypos > ypos + 0.5f) new_ypos = ypos + 0.5f;

      transform.position = new Vector3(
         xpos - 0.7f,
         new_ypos,
         transform.position.z
         );
   }
}
```

Step 36: Test and Save

To test this, simply run the game and crash the ship into a rocket. If there's no error message this time, you were successful in fixing this bug. You may wish to clear all the old error messages in the Console Window first. To do this click the Clear button at the upper left of the Console Window.

In the next section, you'll make the game more interesting by adding shots.

VERSION 0.08: SHOTS

In this section, you're going to add horizontal shots that have the ability to destroy rockets and flying saucers. The scrolling ship shoots under player control. When the player hits the space bar a single shot is released.

You could use Blender to make the shot models, but it's not really necessary. You can use Unity to create shots as long and skinny capsules.

Before you go ahead with the shots, you'll do a small change to the lighting. In game development it's very common to be working on one thing only to discover that there's a simple change on something entirely different that will yield an improvement.

Step 1: Change the Directional light Y Rotation to 100.

This brightens up the playfield a bit and improves the look of the ship and the rockets. Now you're going to create the ship shots directly in Unity.

Step 2: GameObject – 3D Object – Capsule.

Step 3: **Rename** to **shipshot**.

Step 4: Position (**0**, **1**, **0**), **Rotation** (**0**, **0**, **90**), **Scale** (**0.015**, **0.03**, **0.03**).

If you wish, take a look at the capsule by focusing on it in the Scene panel. If the shot is hiding inside of the level geometry, move it up so you can see it. The next step selects a better color for it to make the shot contrast with the background.

Step 5: In the **Materials Folder**, create a **red** Material, **Smoothness 0.9**, name **ShotMat**.

Step 6: Assign **ShotMat** to **shipshot**.

So far all you have is a shot floating in space, not doing anything.

Step 7: Create a C# script with name **shipshot.cs**, and enter the following code for it:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class shipshot : MonoBehaviour
{
    public float shotspeed = 1.0f;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(0, shotspeed * Time.deltaTime, 0);
    }
}
```

Step 8: Assign **shipshot.cs** to **shipshot** and **test**.

The shot is moving, but not colliding with anything, so you'll add collision with the terrain.

Step 9: Create the **terrain** tag and tag **Level_1** with it.

Step 10: Add a **Mesh Collider** to **Level_1**.

Step 11: Add a **RigidBody** component to **shipshot**, and **uncheck Use Gravity** for it.

Step 12: Check Is Trigger for the **Capsule Collider** of **shipshot**.

Step 13: Add the following code to the **shipshot** class in **shipshot.cs**:

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "terrain")
    {
        Destroy(gameObject);
    }
}
```

You did this so that when a shot hits terrain it gets destroyed, just as you would expect.

Step 14: Test this as follows: turn off Maximize on Play, select shipshot in the Hierarchy, and watch when the shot gets destroyed. If you zoom out in the Scene panel, you'll get a better view.

If your shot starts out very close to the terrain, it all might happen too quickly. One way to help with testing this case is to use the step button. Here's how that works. Press play and quickly press pause after that. With the game paused, click on the step icon. You can also use the keyboard shortcut for it. The shortcuts for Play, Pause, and Step can be found in the Edit drop-down menu.

You're ready to launch shots from the ship.

Step 15: Make **shipshot** into a Prefab by dragging it into the **Prefabs** folder. Then **delete shipshot** in the **Hierarchy** panel.

Step 16: Replace the contents of scrollingship.cs with the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class scrollingship : MonoBehaviour
{
    public GameObject shotprefab;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(-0.3f * Time.deltaTime, 0, 0);
```

```
        if (Input.GetKey("w"))
        {
            transform.Translate(0, 0, 0.8f * Time.deltaTime);
        }
        if (Input.GetKey("s"))
        {
            transform.Translate(0, 0, -0.8f * Time.deltaTime);
        }
        if (Input.GetKeyDown("space"))
        {
            Instantiate(shotprefab,
               new Vector3(transform.position.x,
               transform.position.y,
               0.0f),
            Quaternion.AngleAxis(90, Vector3.forward));
        }

    }
}
```

The contents of this code are discussed below, but first, go ahead and test it.

Step 17: Select **ScrollingShipTextured**, and assign the **shipshot** Prefab to **Shotprefab** in the ScrollingShipTextured section of the Inspector by dragging or by using the bullseye icon.

Step 18: Play the game and press the space bar repeatedly to launch the shots.

The shots don't do any damage yet, but you can test that the shots get destroyed when they hit terrain. Now, take a look at the code in shipshot.cs. The Instantiate statement creates a shot every time the player presses the space bar. The initial location of the shot is the same as that of the ship, except that you're hardwiring the z coordinate to zero. That's not exactly great code, but it works. The initial launch angle is set in the Quaternion statement. You can change the angle by adjusting the first parameter, currently set to 90. Experiment with different values for the angle to get a sense of how that works. Quaternions are mathematical objects that encode rotations

using four numbers as an alternative to the more common Euler angles. You may search the internet for additional information about quaternions if you wish. Unity uses quaternions to store rotations of 3D objects.

You're finally ready to shoot at the rockets.

Step 19: Create a **shipshot tag** and **assign** it to the **shipshot** prefab.

Step 20: Add the following code to the end of the `OnTriggerEnter` function in rocket.cs:

```
if (other.tag == "shipshot")
{
   Destroy(gameObject);
   Destroy(other.gameObject);
}
```

This is basically the same code as for colliding rockets with the scrolling ship. You can now test this and try shoot down the rockets.

Testing may reveal a common problem in scrolling shooters: the shots keep going forever and destroy rockets at an unrealistic distance away from the ship. The next step addresses this.

Step 21: In **shipshot.cs**, replace the `Update` function with the following:

```
void Update()
{
   transform.Translate(0, shotspeed * Time.deltaTime, 0);
   GameObject player = GameObject.Find("ScrollingShipTextured");
   if (player)
   {
      if (player.transform.position.x -
          transform.position.x > 3.0f)
          Destroy(gameObject);
   }
}
```

This is fairly straightforward. The code checks to see if the shipshot is over 3 units away from the ship, and destroys the shot if it is.

Step 22: Test and **save**.

As you did previously, turn off Maximize on Play to watch the shots destroy themselves when they get too far away from the ship.

This game is much too easy. In the next section, you'll add some true enemies, flying saucers.

VERSION 0.09: FLYING SAUCERS

In Blender, it's easy to make flying saucers. After you do that, you'll animate their motion in Unity, have them shoot at the scrolling ship, and do the usual collision detection setup and scripting.

Step 1: In Blender, **select File** – **New** and hit the **Enter** key.

Step 2: Delete the **startup cube**.

Step 3: Add – **Mesh** – **UV Sphere**.

Step 4: Type **Numpad 1**, **Numpad 5**, **<Tab>**, and **View – View Selected**.

Step 5: Use Wireframe Viewport Shading.

Step 6: Type **a** to deselect all, then **b** and select the bottom half of the sphere. Do not include the vertices along the middle.

The next few steps shape the bottom half into a saucer.

Step 7: Type **s <Shift> z 2 <Enter>**.

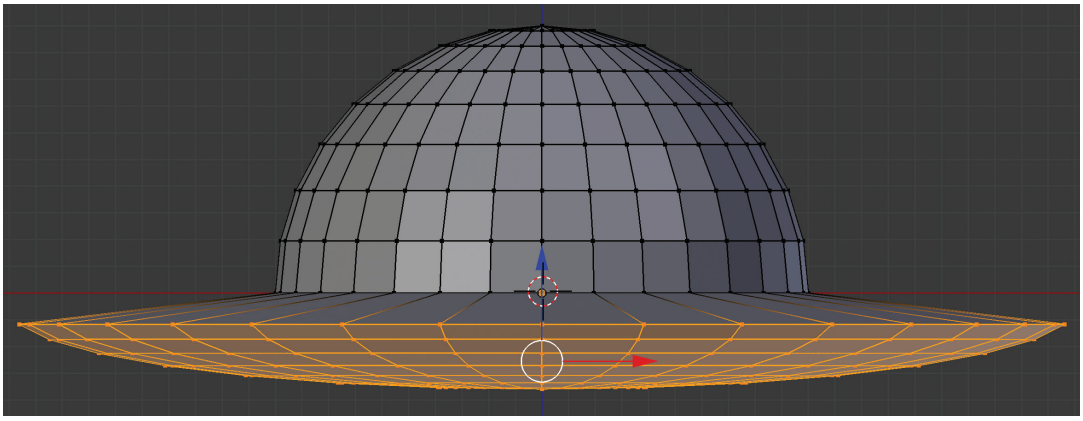
This step restricted the scaling to the axes other than z, i.e., x and y.

Step 8: Type **s z 0.3 <Enter>**.

Step 9: Type **g z 0.4 <Enter>**.

Step 10: Use **Solid Viewport Shading**.

You should now be looking at a flying saucer mesh as shown in Figure 10.33.



▲ **FIGURE 10.31** Simple flying saucer model in Blender.

Step 11: Save the blend file in Assets/Models as saucer.blend.

An advancing Blender user such as yourself could do these last 10 steps in about a minute or two. The keyboard shortcuts in Blender are a great way to become extremely fast at creating 3D models.

Step 12: In Unity, create a new **Material** in the Materials folder, call it **SaucerMat**.

Step 13: For SaucerMat, select **light grey** Albedo color, Smoothness of 0.9

Step 14: Select the **saucer** model in the Models directory, **disable Import Animation**, set **Normals** to **Calculate**, and **Apply**.

Step 15: Drag the **saucer** model into the **Hierarchy**.

Step 16: Select **Position (0, 1, 0)**, **Scale (0.04, 0.04, 0.04)**.

Step 17: Assign **SaucerMat** to the **saucer** object.

Step 18: Drag saucer into the Prefabs folder.

Step 19: Delete the original saucer object in the Hierarchy panel.

You now have a flying saucer prefab, though it still needs some work. The following steps implement collisions of saucers vs. ship shots and saucers vs. ship.

Step 20: Add a **Rigidbody** component to the **saucer prefab**. **Uncheck Use Gravity**.

Step 21: Add a **Box Collider** component to the **saucer prefab**. **Check Is Trigger**.

You now see the box collider outline in the Scene panel. By default, newly created box colliders surround the mesh with a snug fit.

Step 22: Change the **Size** of the box collider to **(2, 2, 1.2)**.

You reduced the size of the box collider to surround just the main hemisphere of the saucer. When in doubt, it's good to have colliders be smaller than the actual meshes. It is common to adjust collider settings later on when all the gameplay elements are available for testing.

It would be tempting to just use a mesh collider for the saucer, but that would be less efficient, and it would feel wrong. A mesh collider would precisely calculate when the geometry of the shot and the geometry of the colliding object intersect. This involved much more computation, depending on the complexity of the models involved. More importantly, experience has shown that simple colliders feel better during gameplay, provided they are adjusted properly to contain only the solid interior parts of models.

In this next step, you're going to write maybe just a little bit too much code all at once. Much of the code will be familiar to you, so it's not overly risky. You're going to put in the motion of the sauces and the collision code as well. If you wish, you may leave out the collision code at first, test, and then add it later.

Step 23: Create **saucer.cs** and assign it to the **saucer prefab**. Use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class saucer: MonoBehaviour
{
   public float radius = 0.2f;
   private float centerx;
   private float centery;
```

```
    private float saucertime;

    // Use this for initialization
    void Start()
    {
        saucertime = 0;
        centerx = transform.position.x;
        centery = transform.position.y;
    }

    // Update is called once per frame
    void Update()
    {
        saucertime += Time.deltaTime;
        transform.position = new Vector3(
            centerx + radius * Mathf.Sin(saucertime * 4),
            centery + radius * Mathf.Cos(saucertime * 4),
            transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "scrollingship")
        {
            Destroy(gameObject);
            Destroy(other.gameObject);
        }

        if (other.tag == "shipshot")
        {
            Destroy(other.gameObject);
            Destroy(gameObject);
        }
    }
}
```

This code makes the saucers move in a circular path using built-in trig functions. You're also doing collisions with shipshots and the scrollingship in the usual manner.

Step 24: Test this by placing some saucers into the **Hierarchy** panel. Then adjust their positions, run the game, shoot them and crash the ship into one.

It's tempting to just drag the saucer prefabs into the Scene panel directly. Don't do that! Unity has to guess which z-coordinate to use, and it often guesses something other than the 0 that you specified in the prefab transform. A z-coordinate other than 0 will break your collision detect code. You can try that out if you like.

You're continuing to follow the philosophy of testing your changes right away. Next, you'll create shots for the flying saucers, and then you'll have the saucers shoot them. The saucer shots will be the same as the ship shots, only they'll fly to the left and they'll have slightly different collision detect code.

Step 25: Drag a shipshot from the Prefabs into the Hierarchy panel.

Step 26: Rename the shipshot object in the Hierarchy panel to saucershot.

Step 27: Remove the Shipshot (Script) component from saucershot.

Step 28: Open shipshot.cs by double-clicking on it in the Scripts folder. Then, in Visual Studio do a Save As with the new name saucershot.

Step 29a: In saucershot.cs on line number 5, change the class name from `shipshot` to `saucershot`.

Step 29b: On line 7, change the initial value of `shotspeed` from `1.0f` to `-1.0f`.

The Update needs to be changed as well, because the code that destroys the shot if it's too far away on the right from ScrollingShipTextured no longer makes sense for alien shots. Instead you need to test if the shot is too far to the left. This is accomplished in the next step.

Step 29c: In the `Update` function, change `> 3.0f` to `< -3.0f` near line 22.

Step 30: Assign **saucershot.cs** to the saucershot object in the Hierarchy.

Step 31: Test this by watching the saucershot fly to the left when you play the game.

Step 32: Drag saucershot into the **Prefabs** folder.

Step 33: Delete the saucershot object in the Hierarchy panel.

Step 34: In **saucer.cs**, add the following line of code near the beginning of the `saucer` class:

```
public GameObject saucershot;
```

Step 35: In **saucer.cs**, add the following code section at the end of the Update function:

```
GameObject player = GameObject.Find("ScrollingShipTextured");
if (player)
   if (player.transform.position.x - transform.position.x < 3.0f)
      if (saucertime > 3.14159f / 2.0f)
      {
          Instantiate(saucershot,
             new Vector3(transform.position.x,
             transform.position.y, 0),
             Quaternion.AngleAxis(90, Vector3.forward));
          saucertime = 0.0f;
      }
```

Step 36: In the **saucer prefab**, click on the bullseye icon for Saucershot and assign the saucershot prefab. You may need to select the Assets tab when making that selection.

You need to give the saucershot its own tag.

Step 37: Select the saucershot prefab, create a saucershot tag, and use it to tag the saucershot prefab.

In theory, those saucers should now be shooting at you.

Step 38: Test by observing that the saucers are periodically shooting to the left.

The saucershots don't harm the ScrollingShip at all. Both the saucershot prefab and the scrollingship are set up for collision detection, so all that's missing is a few lines of code.

Step 39: In saucershot.cs, add the following code to the `OnTriggerEnter` function:

```
if (other.tag == "scrollingship")
{
   Destroy(gameObject);
   Destroy(other.gameObject);
}
```

Step 40: Test this by seeing if the saucershots destroy the scrollingship.

Step 41: Save.

It's time to take inventory of where you are. You have all the graphical elements, except for the bombs. There's some basic gameplay and control. The main things that are missing are scoring, audio, and populating the level with rockets and saucers. There are also bound to be additional changes to the code as you get more experience with playing the game.

In the next section, you'll do some level design.

VERSION 0.10: LEVEL DESIGN

For a change of pace, you'll do something that's technically easy, but artistically it isn't easy at all. Where are you going to put the rockets and saucers? It's really up to you, the designer.

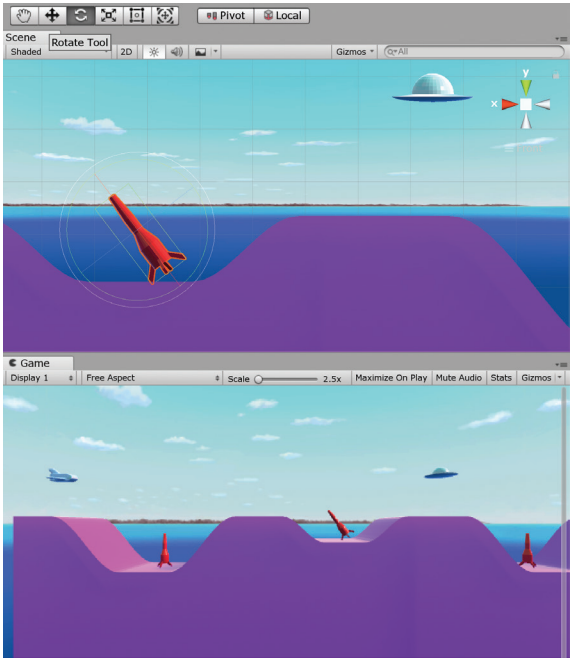
Unity does double duty as both a development environment and a level editor. For large, complex games, developers often build stand-alone level editor applications, but this won't be necessary for you in this game. You simply use your rocket and saucer prefabs and place instances into the scene wherever you want. It's up to you where to locate the saucers and rockets. However, as you do this, you're going to discover some things that will motivate you to make some changes to the code.

Step 1: Duplicate and drag five more rockets into the scene and test.

This is just a warm-up exercise to get you started. Remember to duplicate rockets in the scene already, rather than dragging from the Prefab folder.

Step 2: Select one of the rockets and rotate it using the rotate icon. Test it to see if it works.

You may wish to compare your Unity screen with Figure 10.32.



▲ **FIGURE 10.32** Rotated rocket.

To do the rotation, select the rocket and use the Rotate Tool from the icon strip at the top left of the Unity window, as shown in Figure 10.32.

Step 3: Test to see if the rocket collides with the terrain.

This is easy enough. Just rotate the rocket so it points at some terrain. Apparently, you didn't put in collisions between rockets and terrain, so the rocket just flies through it. The following step fixes this.

Step 4: In **rocket.cs**, add the following code to the `OnTriggerEnter` function:

```
if (other.tag == "terrain")
{
    Destroy(gameObject);
}
```

Step 5: Test.

Well, maybe you're going to see a problem now. Any rocket that gets initialized too close to the terrain gets immediately destroyed. How are you going to fix this? There's an old joke. A man goes into the doctor's office and says that it hurts when he raises his arm. The doctor's advice: Don't raise your arm. So, you could just avoid the problem by never placing the rockets too close to the terrain. This is a bit of a pain, so the following code avoids this issue. You'll put in a timer and only do the rocket vs. terrain collision detect after about a second after launch.

Step 6: Update **rocket.cs** to match the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class rocket : MonoBehaviour
{
    public float rocketspeed;
    private float flighttimer = 0.0f;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        GameObject player = GameObject.Find("ScrollingShipTextured");
        if (player)
        if (player.transform.position.x - transform.position.x < 0.5f)
        {
            transform.Translate(0, 0, rocketspeed * Time.deltaTime);
            flighttimer += Time.deltaTime;
        }
    }
```

```
    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "scrollingship")
        {
            Destroy(gameObject);
            Destroy(other.gameObject);
        }
        if (other.tag == "shipshot")
        {
            Destroy(gameObject);
            Destroy(other.gameObject);
        }
        if (flighttimer > 1.0f)
        if (other.tag == "terrain")
        {
            Destroy(gameObject);
        }
    }
}
```

You've added a private timer variable, initialized it to zero, and you only update it when the rocket is moving. In the `OnTriggerEnter` function you check to see if the rocket has been flying for a while, and only then do you do the terrain collision.

You might have noticed that the rockets fly forever if there's nothing in the way. This can't be good, so let's add some code to limit the life of rockets.

Step 7: Insert the following code in the `Update` function in rocket.cs:

```
if (flighttimer > 5.0f) Destroy(gameObject);
```

Step 8: Test this by watching what happens to a rocket after five seconds of flight.

The easiest way to watch is to turn off Maximize on Play and zoom out far enough in the Scene panel so you can see the tiny rockets as they fly for five seconds and then disappear.

There's another rather obvious problem. You don't have collision detect between ScrollingShip and the terrain. There are several options for this. Should the ship crash and burn when it hits the playfield? Maybe it should just bounce, or take some minor damage. You're going to follow the traditional route and destroy the ship whenever it touches terrain. While you're at it, you shouldn't let the ship fly over the tunnel that you created at the halfway point of the level.

Step 9: Add the following code to **scrollingship.cs**:

```
private void OnTriggerEnter (Collider other)
{
   if (other.tag == "terrain")
   {
      Destroy(gameObject);
   }
}
```

Step 10: Test crashing the ship into terrain.

Nothing happens when you try to crash into the terrain. The first thing to check is to see if ScrollingShipTextured has a trigger. It was supposed to be checked back in the Flying Rockets section. Well, the Box Collider does have the "Is Trigger" box checked, but there's no Rigidbody component. The next step fixes that.

Step 11: Add a Rigidbody component to ScrollingShipTextured and as usual, uncheck the Use Gravity checkbox.

Step 12: Test crashing the ship into terrain again.

It should work this time.

Step 13: Duplicate rockets until you have at least 20 rockets throughout the level and then test the game.

It's starting to be fun to play this game. You do have another gameplay problem. The player has the ability to just fly up and avoid all the obstacles. You noticed that earlier, but didn't actually do anything about it.

Step 14: Insert the following line of code in **scrollingship.cs** at the beginning of the `Update` function, immediately before the `Translate` call:

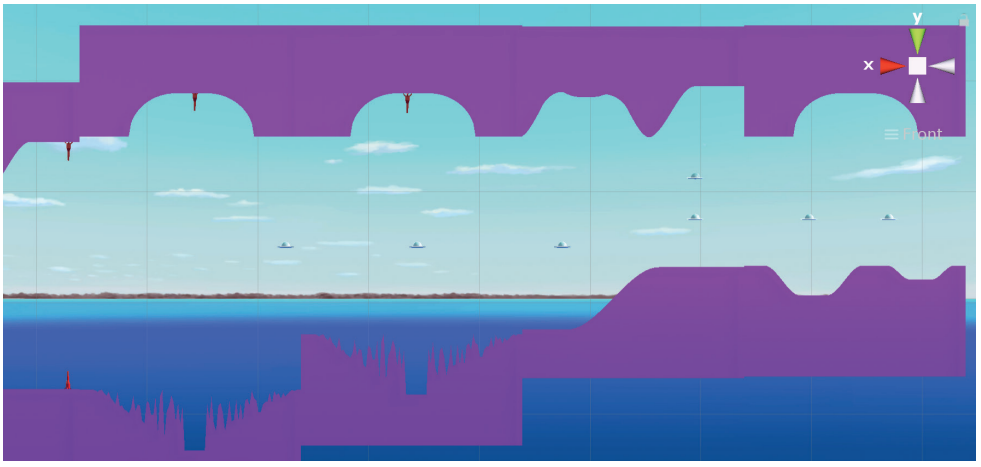
```
if (transform.position.y < 5.0f)
```

The official technical term for code such as this is a "horrible hack." The word goes back to the early days of coding when hacking was considered a good thing, and being called a hacker was the ultimate compliment. A horrible hack is bad code that works, typed in at the last possible moment when you're working on a deadline. Why is this code bad? Well, it's that 5.0 in there. The 5.0 is a "magic number." This code will always keep the ScrollingShip below an elevation of 5.0, regardless of the level design. Sometimes, at the end of a project, you do what you have to do to get the thing done quickly. You just hope that you don't have to deal with this bad code when you add another level later on.

Step 15: Put at least 10 saucers toward the end of the level.

Step 16: Test and Save

Is it fun? In a word, yes! Compare your layout with the one in Figure 10.33.



▲ **FIGURE 10.33** Level layout.

You're done with basic gameplay. There's no ending, and only one level, and many other missing features. Time to add some audio.

VERSION 0.11: AUDIO

Most of today's games have background music, or at least some kind of a background soundtrack. The very early classic games relied entirely on game-triggered sound effects to provide audio. In this section, you'll take it one step further and create a simple background soundtrack using Audacity and the looping feature in Unity.

Step 1: Open Audacity.

Step 2: Generate – Risset Drum…. – OK

Use the default settings, which are 100.0, 2.0, 500.0, 400.0, 25, and 0.8. If there are two Risset Drums in the Generate Menu, use the first one.

Step 3: Effect – PaulStretch…

Again, use the default settings of 10 and 0.25.

Step 4: View – Zoom – Zoom Out.

Step 5: Drag the mouse in the track to make a selection from time 0 to about 2.5 seconds.

Step 6: Effect – Fade In, **Effect – Fade Out**.

Step 7: Effect – Normalize…

Step 8: Transport – Playing – Loop Play.

This is the effect you want, a rumbling, pulsating sound effect.

Step 9: Press the **Stop** icon.

Step 10: File – Save Project with the name rumble.aup in the Sounds folder of your game.

Step 11: File – Export – Export Selected Audio … and use the name rumble.wav.

Step 12: Back in Unity, Preview the rumble sound in the Sounds Asset folder.

There are three items in the Sounds folder, the rumble.aup Audacity file, the rumble_data folder and rumble.wav file. You'll be using the .wav file.

Step 13: Drag the **rumble** sound to the **Main Camera**, and **check** the **Loop** box in the Inspector in the Audio Source section.

Step 14: Test it by playing the game.

If you play the game now, you should hear the rumble effect looping in the background.

Step 15: Select the Sounds folder in the Assets panel.

Step 16: Assets – Import New Asset…, and then navigate to cexplo.wav from the ClassicVerticalShooter project. Repeat for cshot.wav.

These two sound effects may not be perfect, but they'll be good placeholders for now. You'll start with the shot sound.

Step 17: Select **ScrollingShipTextured**, and add an **Audio Source** component. **Uncheck Play on Awake**.

Step 18: In **scrollingship.cs**, make the following changes.

In the `GetKeyDown` section of the Update function, insert the line

```
gameObject.GetComponent<AudioSource>().Play();
```

Step 19: In the Inspector for **ScrollingShipTextured**, set the AudioClip property to **cshot**.

Step 20: Test.

You now hear the familiar shot sound when you shoot missiles with the space bar.

Step 21: In **shipshot.cs**, make the following changes.

After the class declaration near the top of the file, insert the line

```
public AudioClip clip;
```

In the `OnTriggerEnter` function, insert the line

```
AudioSource.PlayClipAtPoint(clip, gameObject.transform.position,
                                                        1.0f);
```

immediately before the Destroy statement.

Step 22: Assign cexplo to the clip variable in the inspector.

This is similar to the technique you used in the last project. It's necessary to use the `PlayClipAtPoint` function because the `PlayOneShot` function needs the object to be alive while playing the sound, and as you can see, the shot is about to be destroyed. You can test this by shooting missiles into terrain.

Step 23: Repeat Steps 21 and 22 to add the **cexplo** sound effect for **all the collision events in saucershot.cs, rocket.cs** and **saucer.cs**.

Step 24: Test and Save.

In the next section, you'll wrap things up by adding scoring.

VERSION 0.12: SCORING

You're going to keep the scoring as simple as possible. The player gets one life, there's just one level. There's an ending and a game over message. Finally, you'll put in scoring for destroying rockets and saucers.

Step 1: Select GameObject – Create Empty and rename it to **scoring**.

Step 2: Create a script with name **scoring**, assign it to the **scoring** object, and use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class scoring : MonoBehaviour
{

    public static int score;

    // Use this for initialization
    void Start()
    {
        score = 0;
    }
```

```
    // Update is called once per frame
    void Update()
    {

    }
    private void OnGUI()
    {
        GUI.Box(new Rect(10, 10, 120, 30), "Score:    " + score);

        GameObject player = GameObject.Find("ScrollingShipTextured");
        if (!player)
        {
            GUI.Button
            (
                new Rect(Screen.width / 2 - 200,
                Screen.height / 2 - 50,
                400, 100), "Game Over"
            );
        }

        if (player)
            if (player.transform.position.x < -24.0f)
            {
                GUI.Button(
                new Rect(Screen.width / 2 - 200,
                Screen.height / 2 - 50,
                400, 100), "The End"
                );
            }
    }
}
```

We put in two buttons to make a minimal attempt at game structure. The "Game Over" message tells the player to stop playing. The only way to play another game is

to exit the program and try again. The "The End" button is the reward for surviving the entire level. The magic number −24.0f was determined by placing an object a little past the end of the level and looking at the x position of it.

The coordinate system for this game turned out to be the reverse of what you might expect. As the ship makes progress along the level the x-coordinate decreases. The test for reaching the end is thus a check to see if the x-coordinate is less than −24.0f.

Step 3: Add the following line to saucer.cs in the `OnTriggerEnter` function in the shipshot section:

```
scoring.score += 900;
```

Step 4: Repeat Step 3 for rocket.cs and a score of 400.

Step 5: Test and **save**.

The scores of 900 and 400 are reflections of the difficulty of hitting saucers vs. rockets. It seems that saucers are more difficult to hit. To test the Game Over message is easy enough, but to get to the end could be a challenge. You can always just set the x-coordinate of the ship to −23 and bypass everything! After testing like this, don't forget to put the ship back at its initial position. While you're at it, try placing the initial ship position a bit farther to the left.

VERSION 1.00: RELEASE AND POSTMORTEM

Our fourth classic project turned into quite a game. It's not ready for commercial release, but it's a start. There's a lot of fun to be had playing the game the way it is, but of course the best part is this: Because you built it from scratch, you have a good understanding of how it all works. You can make changes and improve it (or make it worse) with just a few clicks of the mouse, or a couple of changes to the code.

This project shows how to make a 2D game using 3D tools. The development of the game went very smoothly. There were a few bugs along the way, but that's always going to happen in game development.

The worst problem is obvious: The game isn't finished yet. It would really help to build several levels with graphic and gameplay variety. To only give the player one life also seems very harsh. The bomb weapon from the original design was dropped from production, no pun intended.

In short, the game is playable, looking good, and you likely learned a few things. In the exercises, you are going to explore some new directions.

EXERCISES

1. Use Blender to build a new scrolling ship with the wings near the top of the fuselage. Use texture painting to give it a polkadot texture. Put the new ship into the game.
2. Take the ship from Exercise 1 or the original from the game, load it into Blender, and modify the mesh by extruding a few faces on the back of the ship to create an exhaust. Scale the exhaust faces to make them slightly larger.
3. Build a new and different level using the techniques from this chapter. Save it as Level_2.blend. Create a new Scene in Unity and use Level_2 as your playfield in that scene.
4. Create a bomb in Blender using the same techniques that were used to make the rocket. Integrate the bomb into the game and launch the bomb from the scrolling ship using the "b" key. Use Gravity to have the bomb fall to the ground, and have the bomb collide with terrain, rockets and saucers.
5. Create two new sound effects by experimenting in Audacity. Export the sound effects into the Sounds folder and use them in the game.
6. Create level and lives displays in scoring.cs. Initialize the level to 1 and lives to 3. Then do the next exercise.
7. Use a state machine similar to the one in the Classic Vertical Shooter to implement Game Over and Press Start. When the scrolling ship collides with something, instead of going to Game Over, decrease the lives counter and restart the current level.

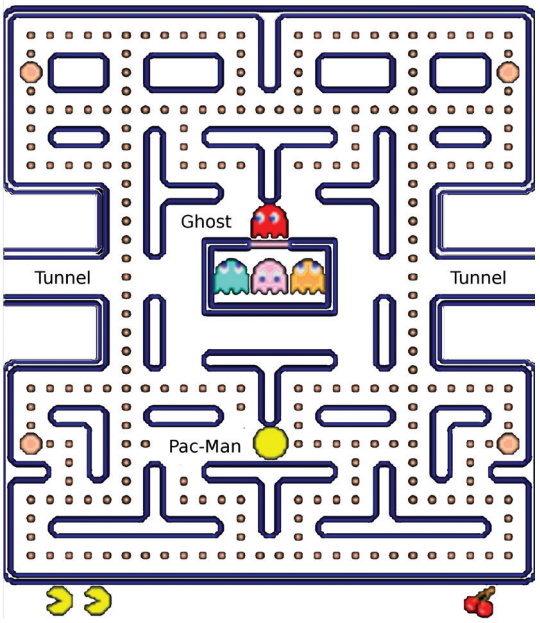
8. At the end of level 1, go on to level 2 from Exercise 3 and have the ending of the game at the end of level 2 instead.
9. Instead of having an ending, go back to level 1 after level 2. Increase the difficulty of the game by making the rockets tougher to avoid and by having the saucers shoot more frequently.
10. Make the saucer shots home in on the scrolling ship on higher levels.
11. Create a Particle System for an exhaust of the Scrolling Ship.

CHAPTER

11 Pac-Man

Pac-Man (Namco, 1980) changed everything. It introduced a completely new game mechanic, was almost entirely nonviolent, and really brought video games to a worldwide mass audience, including women, adults, seniors, and children. *Pac-Man* was created by Namco in Japan and first released in 1980. Official credits weren't given in those early days of game development, but Toru Iwatami is now recognized as the person most responsible for creating this iconic and hugely influential game. Figure 11.1 shows a level diagram of the first maze.

FIGURE 11.1 *Pac-Man* maze.



THE FIRST MAZE GAME

Was *Pac-Man* really the first maze game? The answer depends on how you define "maze game." Sega's *Head On* from 1979 has some similarities to *Pac-Man* but it's a bit of a stretch to put the two games into the same category. *Pac-Man* is definitely the first well-known arcade maze game. The gameplay is deceptively simple, requiring no buttons and just a single joystick to control the main character.

It's instructive to look at gameplay footage of the original arcade *Pac-Man*. Countless videos of this can be found on the internet and it's worth looking at one or two before reading the rest of this chapter.

In *Pac-Man*, the player moves the character around the maze to avoid the four enemies. Three brilliant and novel design elements in the game are the tunnels, the power pellets, and the bonus fruits. The tunnels make it easier for the player to escape when he's cornered. The power pellets let the player fight back instead of getting chased all the time.

The bonus items, mostly fruits such as cherries and apples, are optional rewards that appear at a fixed spot for a limited time, tempting the players to risk their lives to get a few extra points. The bonus fruits aren't really essential to the game, but they add color, and having an extra reward out there to lure greedy players is a fun way to add depth to most any arcade game.

CUTSCENES

Pac-Man isn't just a maze game. It also introduced cutscenes as a way to advance the story in video games. They are noninteractive and, in arcade games, they are necessarily brief. Today's much longer cutscenes need to be skippable, but in these early arcade cutscenes the players had no choice but to watch them in their entirety.

The real hidden purpose of cutscenes to an arcade gamer is to provide a short period of rest between intense periods of gameplay action. You might get bored when watching the same cutscene too many times, but getting a few seconds of respite is always appreciated.

It only took a few years for the game industry to respond by going hog-wild with cutscenes, eventually culminating in million-dollar budgets that sometimes eclipsed the budgets for the rest of the game, or so it seemed.

Cutscenes have even been used as an anti-piracy measure. The short but plentiful cutscenes in 1996's *Gubble*™ were used as uncompressed filler on the CD-ROM to make the game artificially large, thus harder to pirate and download using a slow Internet connection.

PAC-MAN FEVER

Pac-Man had a huge cultural impact, especially in the United States. Soon after the release of the game itself, there appeared an animated television series, t-shirts, and the hit pop-song "Pac-Man Fever." Amazingly, Ken Uston's strategy guide, *Mastering Pac-Man*, sold over a million copies in the '80s. Video games had reached mainstream popular culture, virtually overnight.

ENDING RULE

Our next classic game design rule is somewhat of an oddity, because most classic games, and all games featured in this book, including *Pac-Man*, break it!

Rule 7: Ending Rule: Make an ending.

Just about all classic coin-op games in the classic era don't have a designed ending. The strange thing is that, due to programming limitations, a few of the games had what's now called a "kill screen," including *Pac-Man*. Kill screens kill the player off due to a programming or design bug, effectively ending the game. If you haven't seen the *Pac-Man* kill screen, go and search for it online to take a look.

The main point is that the designers of that era simply didn't bother to design an ending for their games, which was a mistake. This was a great example of industry-wide group-think, where everybody thought it was OK to have the games go on "forever." Even stranger was the general feeling that games without an ending were the "standard" way of designing arcade games. It was something that arcade players had come to expect, mainly due to the publicity surrounding marathon gaming sessions on *Asteroids* and *Missile Command*. There was a certain mystique surrounding people who had "mastered" a particular game, and thus could play it as long as they wanted, effectively "owning" the machine.

The downside of not having an ending is clear. The top scores become more a measure of endurance than skill, violating the Score Rule. Experts lose interest when the

game just goes on and on the same way, breaking the Experts Rule and the Difficulty Ramping Rule as well.

PAC-MAN AI

Here is where it gets really interesting for game designers. Just how do those ghosts decide where to go? In the context of game design, the logic behind character behavior is called *artificial intelligence*, or AI. First, consider the basics of *Pac-Man* AI. The ghosts go at constant speed and usually don't turn around. They switch between two modes, chase and scatter. When they chase, they use their own individual rules to decide which way to turn at an intersection. When they scatter, they simply aim to go to their individual target location. Each ghost has a target in its own corner.

There remains the question of which way the ghosts should turn when they get to an intersection. If they all turn towards the player, they would all behave the same way and as a result, they could be bunched together like a flock of sheep. If the ghost behavior were truly intelligent, the player would have no chance because the ghosts could simply coordinate their efforts to trap the player. The approach taken by Toru Iwatani is to make all four ghosts aim at different yet sensible target locations.

The exact details of chase mode for the four ghosts can be found online. To summarize, the red ghost always aims at the player, the blue and pink ghosts aim at spots near the player, and the orange ghost only aims for the player when he's far away from the player; otherwise, he goes into scatter mode, where he aims at his starting position.

Of course, when the player has the power pellet, the ghosts immediately switch to "run away" mode.

AI programming in *Pac-Man* was done in assembly language, the preferred programming technology of arcade games at the time. Because of this, the artificial intelligence of the ghosts is nothing more than a few carefully crafted assembly language instructions.

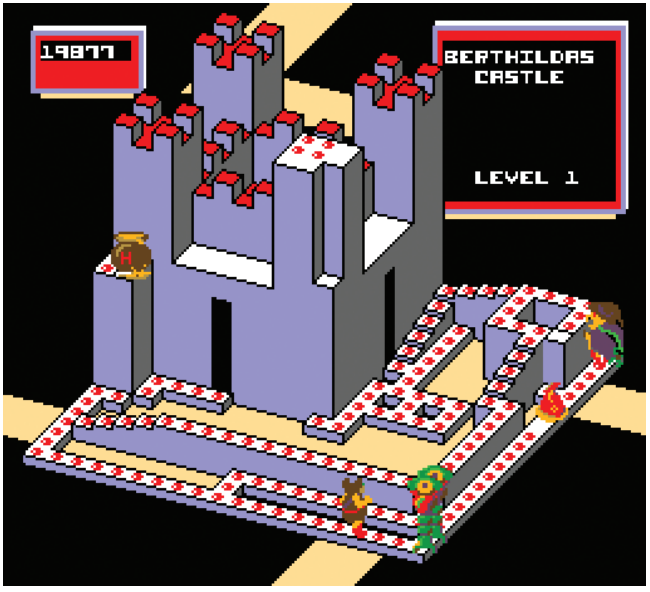
Modern path-finding algorithms have largely supplanted these early AI efforts. The days of writing AI code in assembly are history, but it's still interesting to study

the old techniques. They continue to be useful and should be in every game designer's AI arsenal.

PAC-MAN SEQUELS AND MAZE GAMES

In contrast to the earlier arcade mega-hits, for *Pac-Man* the sequels were plentiful and hugely successful, especially in the '80s. Namco's official sequels included *Ms. Pac-Man*, *Super Pac-Man*, *Jr. Pac-Man*, and *Pac-Mania*. The arcade game industry adopted the new maze game category with gusto and released games such as *Mr. Do* (Universal, 1982), *Dig Dug* (Namco, 1982), *Lady Bug* (Universal, 1981), and *Pepper II* (Exidy, 1982), just to mention a few.

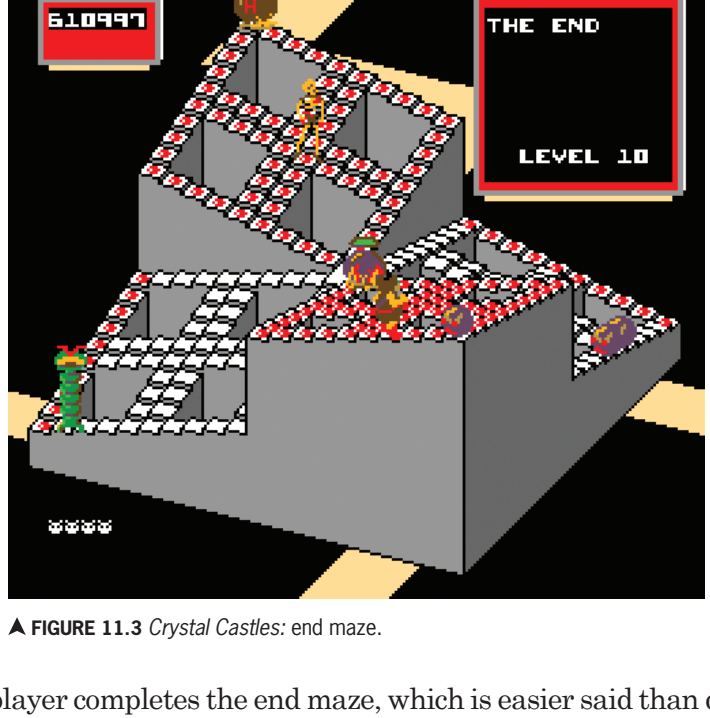
The move to 3D maze games started in 1983 with Atari's *Crystal Castles*. A screenshot from *Crystal Castles* is shown in Figure 11.2.



▲ **FIGURE 11.2** *Crystal Castles:* Berthilda's Castle.

Crystal Castles was designed and programmed in 1982 and 1983 by the author of this book, Franz Lanzinger. In this game, the player controls Bentley Bear with a trackball to collect gems from isometric castles. The game achieved some notoriety for being the first coin-op nonracing game with a designed ending.

The ending in *Crystal Castles* illustrates many of the eight rules of classic game design, especially Rules 5 through 8. The unique and strange end maze is shown in Figure 11.3.

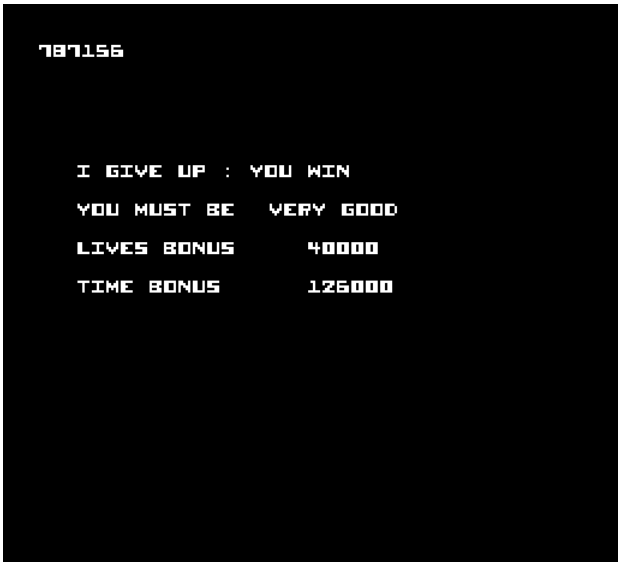


▲ **FIGURE 11.3** *Crystal Castles:* end maze.

After the player completes the end maze, which is easier said than done, there are some surprise bonus scores, as shown in Figure 11.4.

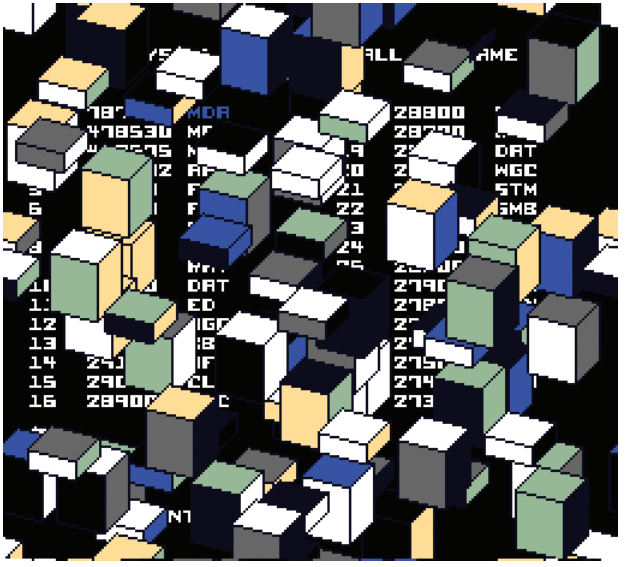
A hidden timer is used to calculate the time that it took to get to the end. Then a bonus score is awarded. The faster the player finished the game, the higher the bonus score, using a simple linear equation. There's also a bonus for any unused lives at the end. The game selects one of several built-in congratulatory messages based on the lives bonus.

Multiple endings are always problematic, especially in large console games, because they imply that players should replay the game over and over if they wish to see all the endings. This can be boring to the players unless the game has good replayability.



▲ **FIGURE 11.4** *Crystal Castles:* scoring at the end.

Following the score display, there's a short final animation that simply draws random boxes on the screen, as shown in Figure 11.5. This ending had to fit into just a few lines of code, so it reused the existing functions for drawing the mazes.



▲ **FIGURE 11.5** *Crystal Castles:* boxes gone wild at the very end.

It is very difficult to get to the ending in Crystal Castles. Only a few of the top players were good enough to get to the end maze, much less finish it. It's even more difficult if not impossible to get to the end when the trackball control is replaced with a joystick.

In 1996, a new independent game development company, Actual Entertainment, was formed by Franz Lanzinger, Mark Robichek, and Eric Ginner to make an unofficial sequel to *Crystal Castles*. The resulting game series, *Gubble*, continues to explore and expand the maze game category. The last maze of the original *Gubble*, right before the ending, is shown in Figure 11.6.

▲ **FIGURE 11.6** *Gubble.*

If you look closely, you'll find the letters "JoeC" as part of the playfield. Many maze games, starting with *Mr. Do*, used this technique to display text, anything from the name of the game itself to initials or even subliminal messages. JoeC stands for Joe Cain, one of the developers of *Gubble*. *Crystal Castles* used this technique in several places, and it even displayed the initials of the current high score holder in this way on the first maze.

Pac-Man is the quintessential classic game. It's simple yet deep. It and its official and unofficial sequels live on decades later. It's fun. Next time you see an original Pac-Man arcade machine, insert a coin, and be amazed at how great it feels to grab a real arcade joystick and be Pac-Man.

CHAPTER 12

Classic Game Project Five: Maze Game

It took quite a bit of effort to make the first four classic game projects, so now we're going to make something a little different and smaller, a maze game. Don't forget Rule 1: Keep it simple. It's surprising how much fun the deceptively simple games can be.

DESIGNING A MAZE GAME

You'll start, as always, with the playfield, in this case a maze. You'll be using Blender to make the maze, so there's no need to sketch the maze right now. The main character is going to be a sphere. In order to keep things simple, you don't want to spend a lot of time creating animated characters. Instead, you'll use some of Unity's built-in shapes and get on with making the game fun. There's a long history of successful classic and modern games that use abstract shapes as characters, so that's reasonable justification for "going abstract" here as well.

Following the classic maze game design pattern, the main character is going to collect things in the maze while trying to avoid enemies. The enemies move around and are trying to attack the player. In addition to the main character, you'll need to decide on designs for enemies and things to collect. In keeping with an abstract theme, the enemies are going to be tumbling cubes and the things to collect are smaller spheres. It doesn't get much simpler than that. If there's a need for differentiating enemies there's always color, size, and basic animation available.

This isn't going to be a direct *clone* of *Pac-man*. Your goal is to make an original game in the same general category as the arcade maze games of the '80s. Cloning famous games can be educational, but it's even more instructive to make an original game where you don't know ahead of time how it's going to turn out.

VERSION 0.01: THE MAZE

Step 1: Start up **Unity** and create a **new 3D project** with the name **ClassicMaze-Game**.

Step 2: Create the following folders in the Assets panel: **Materials**, **Models**, **Prefabs**, **Scripts**, and **Sounds**.

Step 3: Save and **exit**.

This is the basic setup for starting a new project that was used in some of your previous projects. It's not really necessary to exit Unity, but it's a good habit to exit frequently to test that your saves are working correctly. Next, you'll use Blender to make the maze.

Step 4: Start Blender.

Step 5: Type s Shift-z 8 <enter>.

This step scales the starting cube by a factor of 8 in the x and y, but not in the z direction.

Step 6: Select View – View Selected in the 3D View Menu.

Step 7: Press <Tab> to go into edit mode.

Step 8: Type numpad-1 and **numpad-5** to get to Front Ortho view.

Step 9: Type **z** to select **wireframe** viewport shading.

Step 10: Get into Face Select Mode.

Face Select mode is chosen by clicking on the third cube-shaped icon at the bottom of the 3D view. The Face Select icon looks like a cube with the front face highlighted in orange.

Step 11: Move the mouse into the 3D panel, then **Type a** to deselect everything.

Step 12: Type b to get to border select mode. Select the top line of the rectangle.

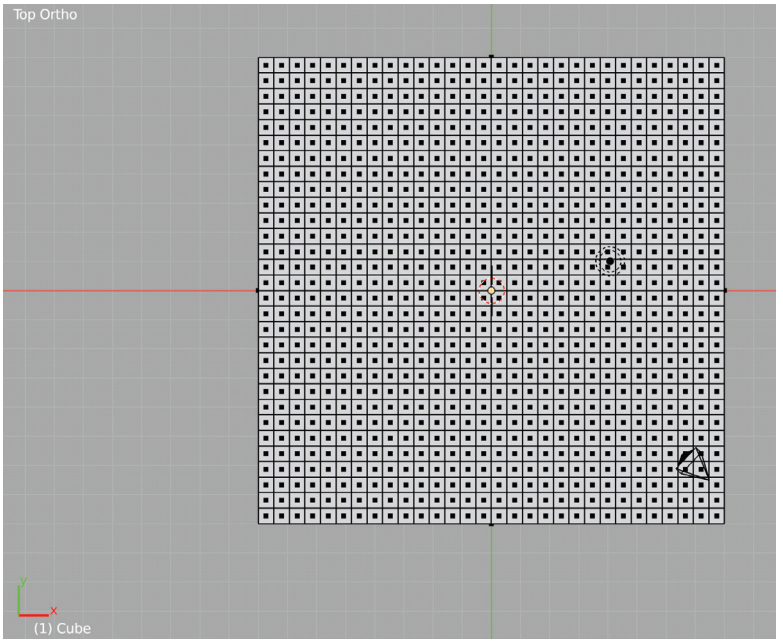
Step 13: Select **Mesh – Edges – Subdivide** and set **Number of Cuts** to **29**.

This step created a 30 x 30 grid on the top of the block.

Step 14: Type numpad-7 to go into Top Ortho view.

Step 15: Type **z a** to exit wireframe mode and deselect everything again.

Your 3D viewport should now look like Figure 12.1.



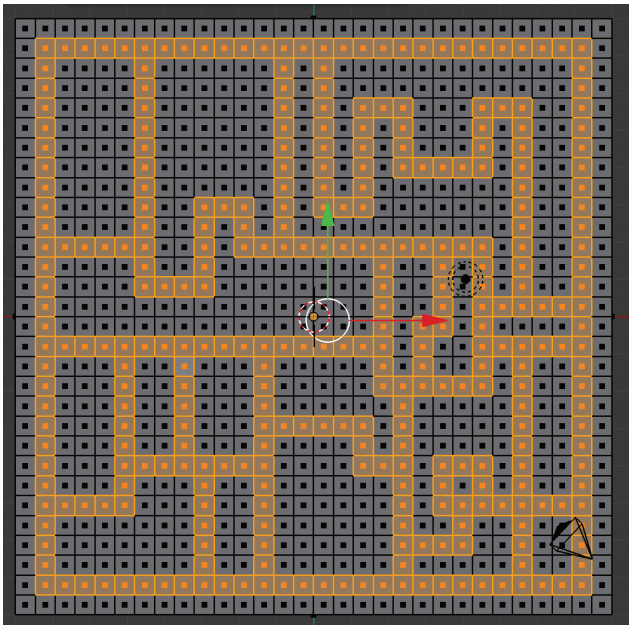
▲ **FIGURE 12.1** Grid for classic maze game.

Step 16: Select File – Save with name **mazegrid.blend** in **Assets/Models**.

You've just created a block with a 30 x 30 grid on top. This prepared you for making the maze. You'll select faces from the grid in order to extrude them later on.

Step 17: Press **<Shift>right-click** to select a few faces for your pathway. Then repeatedly use border select mode with the **b** key to add groups of faces to your selection.

Compare your creation to Figure 12.2. Your selection doesn't need to exactly match that figure, but you should have something similar. You can use Undo while doing this if something doesn't go quite right. You can <Shift>right-click on a selected face to unselect it.



▲ **FIGURE 12.2** Selecting the path.

The next two steps do the extrusion where you take the path and push it into mesh. Think of it as carving a path out of a large block of granite.

Step 18: Type number pad 1 to go back to the Front Ortho view.

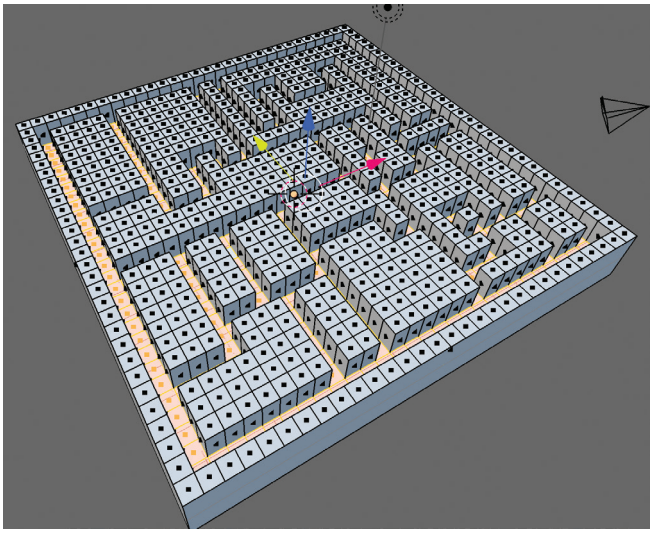
In the following step, be sure to type the minus sign in front of the "1."

Step 19: Type e -1 <enter> to extrude the path down into the mesh by 1 unit.

Step 20: Type numpad-8 (three times) numpad-4 (two times) numpad-5.

Step 21: Use the **numpad-minus** and **numpad-plus** keys to zoom in and out. Alternatively, you can use the mouse scroll wheel.

Your screen should now look like Figure 12.3.



▲ **FIGURE 12.3** Extruded maze path.

This maze is somewhat experimental. You can't expect to make a great maze before having a game to test with. Once you have some basic gameplay, you'll take another shot at making the real maze.

Before you bring this maze into Unity, there's a little bit of housekeeping to do. In order to have a different color or texture for the maze path as opposed to the maze walls, it's best to separate the two into different meshes. This may not be the most efficient way of doing things, but it's easy and simple.

Step 22: Type x – faces.

Because you still had the path selected, it was easy to delete it. The floor will later be replaced by a single large plane in Unity.

Step 23: Select File – Save As… using the name **maze_proto.blend**. **Exit** Blender.

It's time to go back to Unity and see what this maze looks like there.

Step 24: In Unity, s**elect maze_proto** in the Models folder.

Step 25: In the Inspector, click on **Animations**, and **uncheck Import Animation**.

Step 26: Click on **Apply**.

Step 27: Click on **Model**, and use **Calculate** for **Normals** and **Apply**.

Step 28: Drag **maze_proto** from the Assets panel to the **Hierarchy panel**.

Step 29: Set the **Position** to **(0, 0, 0)**, if necessary.

Step 30: Use **Top perspective view** in the Scene panel.

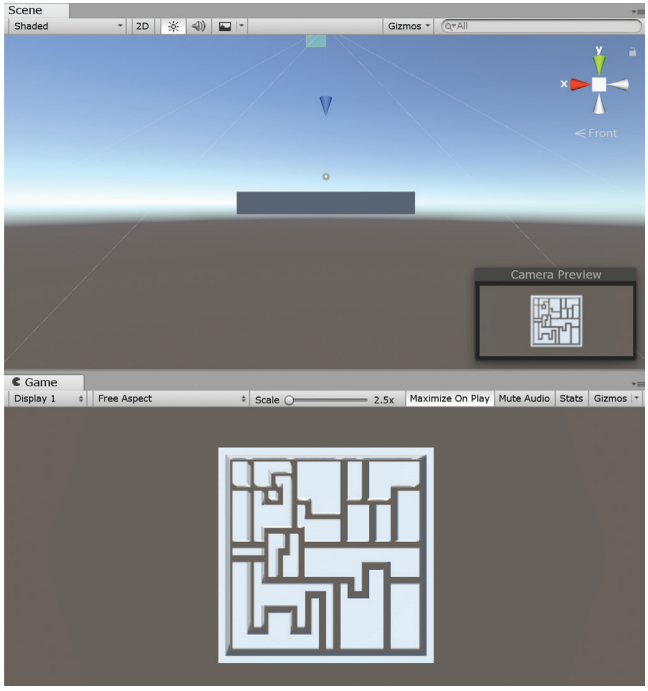
Step 31: Press **f** to focus the Scene panel onto maze_proto.

Step 32: Move the **Main Camera** to **Position (0, 20, 0)**, **Rotation (90, 0, 0)**.

The x rotation of 90 points the camera down, and the y position of 20 moves it up and away from the maze. That's where you want to have the initial position for the camera. Later on, you'll move the camera closer and scroll it to follow the main character around.

Step 33: Use **Front perspective view** in the Scene panel.

Your Unity Scene and Game panels should look like Figure 12.4.



▲ **FIGURE 12.4** Setting up the maze prototype in Unity.

If you look carefully, you'll notice that the maze is rotated 180 degrees compared to the view in Blender. You are going to ignore this issue for now. When the time comes to design the real maze, it'll be easy to correct this by rotating the maze or the camera in the inspector.

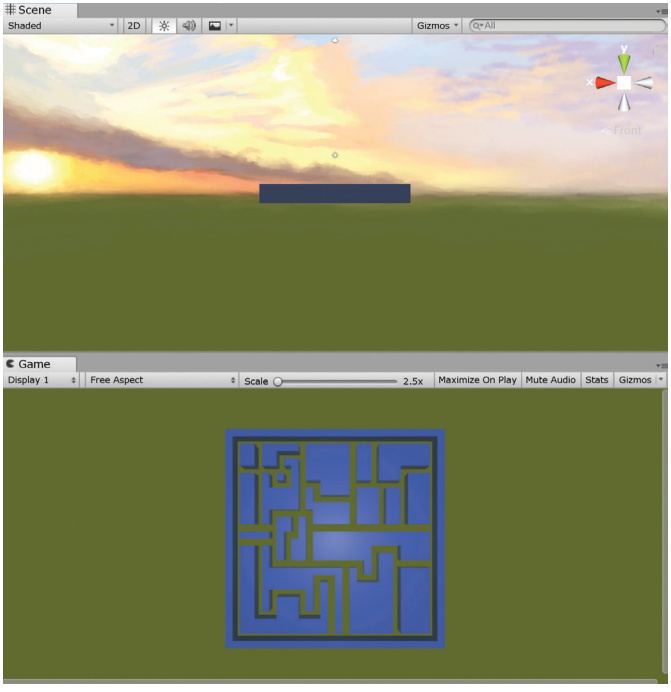
It's time to add better lighting and a material for the maze.

Step 34: Create a **Point Light** at **Position (0, 5, 0)**. **Delete** the default **Directional Light**.

Step 35: In the **Assets – Materials** folder, create a **light blue material**, rename it to **mazemat**, and assign it to **maze_proto**.

Step 36: Add a **skybox** like you did for the scrolling shooter project. Use the **Newdawn1** Skybox or another skybox of your choosing.

This Skybox is there just for decoration and doesn't affect gameplay. Compare your screen with Figure 12.5.



▲ **FIGURE 12.5** The sun rises on the maze.

Almost any skybox would work here, so feel free to substitute another one from the Asset Store. And yes, it is possible to make your own skyboxes. If you search the internet for "making your own skybox Unity" you'll find instructions on how to do that.

The maze needs a floor.

Step 37: In the Hierarchy Panel, select **Create – 3D Object – Plane**, and rename it to **floor** in the Inspector panel.

Step 38: Change the **Scale** of the floor to **(2, 1, 2)**.

Step 39: Create a **material** for the floor, make it **dark blue**, name **floormat, assign** to **floor**.

The maze is looking kind of dark now, so turn up the range of the light.

Step 40: Set the **Range** of the **Point light** to **20**.

There's a lot more to creating good lighting than adjusting the range of the one light. It's fun to put several lights into the scene and to experiment with colors, ranges, and intensities. Feel free to do this, if you like.

Step 41: Test and **Save**.

There's not much to test here, but it's still worthwhile to make sure you can run the game, even though it's just a static look at the maze. The prototype maze is complete and set up in Unity. You are now ready to create the maze characters.

VERSION 0.02: THE PLAYER

The player will be a sphere. While you might get more control over the mesh by making it in Blender, the built-in sphere in Unity is fine, so go ahead and use that.

Step 1: In the Hierarchy panel, create a Sphere and rename it to **player**.

Step 2: Create a **green player material,** name **playermat** and assign it to the **player**.

Step 3: Change the player **Position** to **(0, 2, 0)**, **Scale (0.4, 0.4, 0.4)**.

You just placed the player hovering above the maze for now.

Step 4: Add Component – Physics – Rigidbody.

Yes, you're going to leave Gravity for the player.

Step 5: Create a **player.cs** script and assign it to **player**. Use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class player : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    float factor = 10.0f;
    // Update is called once per frame
    void Update()
    {
        if (Input.GetKey("right"))
        {
            GetComponent<Rigidbody>().AddForce(Vector3.right * factor);
        }
        if (Input.GetKey("left"))
        {
            GetComponent<Rigidbody>().AddForce(Vector3.left * factor);
        }
        if (Input.GetKey("down"))
        {
            GetComponent<Rigidbody>().AddForce(Vector3.back * factor);
        }
        if (Input.GetKey("up"))
        {
```

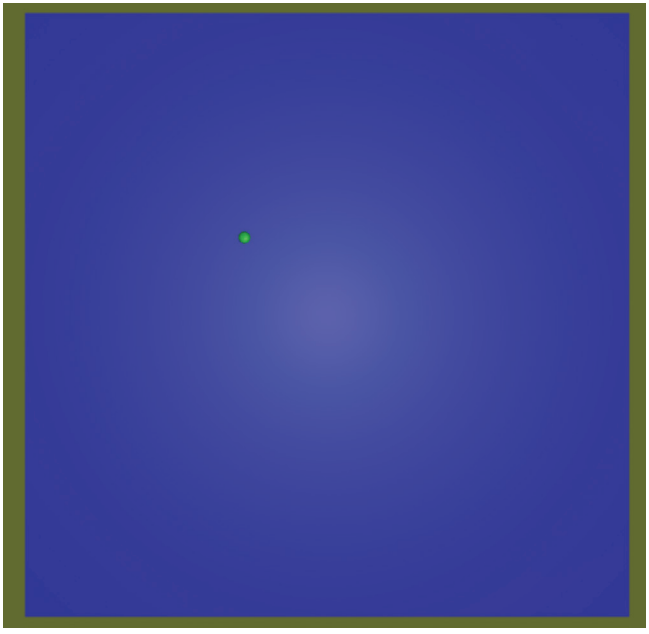
```
            GetComponent<Rigidbody>().AddForce(Vector3.forward * factor);
        }
    }
}
```

This code lets you control the player with the four arrow keys on your keyboard. To test this out, do the following steps:

Step 6: Disable the **Mesh Renderer** for **maze_proto** by unchecking the box next to "Mesh Renderer" in the Inspector panel.

Step 7: Run the game.

Your game panel should look like Figure 12.6. Press the four arrow keys and test that you can control the player.



▲ **FIGURE 12.6** Testing the player control.

Step 8: Stop running the game.

Step 9: Enable the **Mesh Renderer** for **maze_proto**.

Step 10: Add Component – Physics – Mesh Collider.

Step 11: Run the game.

This time you turned on the renderer for maze_proto and added a mesh collider component. This allows you to move the ball around in the maze. If the player starts out on top of a blue wall, slowly move it into a maze path and it will drop down.

Step 12: Stop running the game.

In the next step, you'll initialize the player in the maze rather than floating above it.

Step 13: Change the Position of the **player** so that the player is initialized in the maze. Use a y position of 0.5.

Depending on your maze, you may need to adjust the x and z position. A good y position is 0.5. To find where to initialize the player you can use the Scene panel with at Top Iso view, select the player, and then move the player around with the Move Tool until the player is at a good location. For the maze from the book, the new initial position is (0, 0.5, 0.8).

When you're done making this adjustment, don't forget to test.

Step 14: Test the player control again.

Step 15: Save.

In the next section, you'll create the enemies.

VERSION 0.03: NASTY ENEMIES

The enemies are going to be cubes, but you're going to use a sphere collider for them. This makes the cubes appear to be tumbling around.

Step 1: In the Hierarchy panel, create a cube, rename it **nasty enemy**.

Step 2: Make a **red material** for the **nasty enemy** and name it **nastymat**.

As usual, put the material into the Materials folder and don't forget to assign the material to the nasty enemy.

Step 3: Change the **Scale** for the nasty enemy to **(0.4, 0.4, 0.4)** and **place it onto the maze**.

You use the same technique you used for the player. The y position should be 0.3, whereas x and z are adjusted so that the nasty enemy is on a path rather than hidden inside the maze mesh.

Step 4: Add Component – Physics – Rigidbody.

Step 5: Test.

Here is where the real fun begins. The nasty enemy isn't at all nasty yet, but just sits there, but if you crash the player into the nasty enemy it reacts and bounces away. In the next step you'll use a neat trick to make it tumble.

Step 6: Remove the **Box Collider**, and **add** a **Sphere Collider** in **nasty enemy**.

To remove the Box Collider, click on the gear icon at the right and select Remove Component. Feel free to test this, if you wish. The enemy is now tumbling. The next step makes the enemy move towards the player.

Step 7: Create a nastyenemy.cs C# Script for the nasty enemy and use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class nastyenemy : MonoBehaviour
{
    public float factor = 5.0f;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
```

```
        Vector3 dir = new Vector3(0, 0, 0);
        GameObject player = GameObject.Find("player");
        if (player)
            dir = player.transform.position - transform.position;
        GetComponent<Rigidbody>().AddForce(dir * factor);
    }
}
```

This code is short and effective. At the bottom of the Update function, you can see an AddForce function call. The variable `dir` is a vector that stores the 3D direction of the force that's going to be applied. The direction is computed to be a vector from the enemy position to the player position. In other words, this function tells the enemy object to move directly towards the player, and to do so with a force proportional to the distance between them.

Step 8: Test.

When you run the game now, the nasty enemy follows the player around like a dog on an elastic leash. Although you're using a modern physics engine to implement it, the resulting movement of the nasty enemy is actually the same as the bowling balls from *Crystal Castles*, written over 35 years ago.

Step 9: Drag nasty enemy into the Prefabs folder.

Step 10: Use the Top Iso view in the Scene panel and duplicate the nasty enemy a few times.

Step 11: Experiment with different starting positions for the enemies.

Step 12: Create a light blue easy enemy material, and assign it to one of the nasty enemy instances in the Hierarchy. Set the factor to 2 for the easy enemy, and test it.

It's now time to add collision detection for the enemy vs. player. This has to be done a little differently than in the past because you're using the physics engine.

Step 13: Replace the nastyenemy.cs file with the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class nastyenemy : MonoBehaviour
{
   public float factor = 5.0f;
   private Vector3 initPosition;

   // Use this for initialization
   void Start()
   {
      initPosition = transform.position;
   }

   void RestorePosition()
   {
      transform.position = initPosition;
   }

   // Update is called once per frame
   void Update()
   {
      Vector3 dir = new Vector3(0, 0, 0);
      GameObject player = GameObject.Find("player");
      if (player)
         dir = player.transform.position - transform.position;
      GetComponent<Rigidbody>().AddForce(dir * factor);

   }

   private void OnCollisionEnter(Collision collision)
   {
      if (collision.gameObject.name == "player")
```

```
        {
            collision.gameObject.transform.position =
                new Vector3(0, 0.5f, 0.8);
            RestorePosition();
        }
    }
}
```

This code is fairly straightforward. When the player collides with an enemy, both the player and the enemy are sent back to their starting positions. You may need to change the numbers in the code at the end that repositions the player. This is a bit abrupt, but it's good enough for now. We do have a problem with the enemies and the player getting launched away from the maze as a result of movement that is too fast. This is a bug you'll live with for now.

Step 14: Test and **save**.

You have a pretty good framework for creating a bunch of levels. The enemies aren't exactly smart, but that's OK. It can be fun to try to outwit a bunch of dumb enemies. In the classic era, it wasn't possible to have very sophisticated character movement code and yet it didn't matter because it was still possible to tune and balance these games and end up with something incredibly fun and exciting.

In the next section, you'll be adding dots.

VERSION 0.04: DOTS

You still don't have a game. All that's missing is a goal for the player.

Step 1: In Unity, create a **Sphere**, **rename** it to **dot**, **Position (0, 0.3, 0)**, **Scale (0.2, 0.2, 0.2)** and put it **on the maze**.

Step 2: Create a **white** material for it.

Step 3: Create **dot.cs** and assign it to the dot object. Insert this:

```
private void OnTriggerEnter(Collider other)
{
```

```
    if (other.name == "player")
    {
        Destroy(gameObject);
    }
}
```

Step 4: Select the dot object in the Hierarchy and look at the Sphere Collider in the Inspector panel. **Check Is Trigger**.

Step 5: Test that the dot disappears when colliding with the player.

Step 6: Make a dot prefab in the usual manner.

Step 7: Put three dots onto the maze path near the initial player position by dragging two more instances from the Prefabs directory into the Hierarchy. Check that the Y coordinate is still 0.3 for all the instances.

Step 8: Test and **save**.

The three dots should disappear when the player runs into them. Later, during development, you'll put many dots out there, but for initial testing and development it's better to just have a few dots. The next section shows how to put in some basic sounds.

VERSION 0.05: AUDIO

The sound design for this game is fairly simple, with one new twist. In the previous projects, the sound effects were typically triggered by collisions. This game doesn't have very many collisions, so how about you put in sound effects for each of the four arrow key controls?

The plan is to go into Audacity and make four similar sound effects for the four arrow keys, a nice happy sound for when the player picks up a dot, and a sad sound when the player dies.

Step 1: Start Audacity.

In the next step, to get the NTSC frames time scale, you'll click on a small arrow next to the Duration display.

Step 2: Select **Generate – Chirp…**, set Duration to **5 NTSC frames**, **Sine** Waveform, **Frequency 440, 1320**, **OK**. Press **Play** to test it (the green arrow).

Your waveform should be about 0.17 seconds long, just right for a short sound effect. The NTSC frame time scale allows for quick entry of short time durations.

Step 3: File – Save Project As … arrowkeys.aup in the Sounds folder in Assets of the ClassicMazeGame project.

Step 4: File – Export – Export as WAV with name **arrowup.wav** in the same folder.

Step 5: Effect – Change Pitch … -3 semitones. Press **Play** to test it.

Step 6: File – Export – Export as WAV with name **arrowdown.wav** in the same folder.

Step 7: Undo Step 5 by **Edit – Undo Change Pitch**, then **Effect – Change Pitch… -1 semitones**. Press **Play** to test it.

Step 8: File – Export – Export as WAV with name **… arrowright.wav**.

Step 9: Effect – Change Pitch … -7 semitones. Press **Play** to test it.

Step 10: export to **arrowleft.wav**.

The chirp sound effect is a quick way to make the iconic beeps that were so common in classic video games. There are just two more effects for collisions.

Step 11a: Select – All, Edit – Delete.

Step 11b: **Generate – Chirp**, **Sine Waveform**, **Frequency 440 – 5000**, **Duration 15 NTSC frames**. **Test**.

Step 11c: Select File – Export – Export as WAV with name **dot.wav**.

Step 12a: Select – All, Edit – Delete.

Step 12b: Generate – Chirp, **Sawtooth Waveform, Frequency 1000 – 50, Duration 15 NTSC frames. Test.**

Step 12c: Select File – Export – Export as WAV with name **enemy.wav**.

There might be a method to this madness. The rising frequency in the chirp makes a happier sound than a falling frequency shift. To get an authentic retro sound, it helps to use Sawtooth and Sine wave forms. Those simple wave forms were easily generated and commonly available in the early sound chips.

Step 13: In Unity and **test** the **sound effects** in the Sounds folder.

Step 14: In **dot.cs**, insert the following two lines code:

```
public AudioClip dotsound;
AudioSource.PlayClipAtPoint(dotsound, transform.position, 1.0f);
```

The first line goes before the `Start` function, the second line goes before the Destroy function call in the `OnTriggerEnter` function.

Step 15: For the dot prefab, assign the **dot** sound for the **Dotsound** property in the Inspector using the bullseye icon at the far right and then **test** the sound in the game.

Step 16: Add the **enemy sound** in a similar manner to Steps 14 and 15. **Test**.

Step 17: Insert the following **line of code** to **player.cs** immediately before the `Start` function:

```
public AudioClip aleft, aright, aup, adown;
```

This line declares four variables all at once. It's usually a good idea to give each variable its own line, but here the four variables are very similar, so it's OK to group them together like this. Your goal is to make your code maintainable and clear. That is more important than saving space.

Step 18: Insert the following code to player.cs at the beginning of the Update function:

```
if (Input.GetKeyDown("right"))
    {
        AudioSource.PlayClipAtPoint(aright, transform.position);
    }
    if (Input.GetKeyDown("left"))
    {
        AudioSource.PlayClipAtPoint(aleft, transform.position);
    }
```

```
if (Input.GetKeyDown("up"))
{
   AudioSource.PlayClipAtPoint(aup, transform.position);
}
if (Input.GetKeyDown("down"))
{
   AudioSource.PlayClipAtPoint(adown, transform.position);
}
```

Step 19: Assign the four arrow sounds to the associated properties for the player.

This code looks similar to the other half of the Update function, but there's an important difference. The `GetKeyDown` function tests for a transition of the key from *not-pressed* to *pressed down*. The `GetKey` function just tests to see if the key is pressed, so it keeps doing the `AddForce` function calls every frame whenever the particular key is pressed.

Step 20: Test and **save**.

Much more can be done with sound in this game, of course. The reader is encouraged to experiment with additional sound effects and background sounds.

The next section adds scoring, levels, and difficulty ramping.

VERSION 0.06: SCORING AND LEVELS

Scoring is easy, but adding levels takes effort. The tricky part is to restore the dots for the next level. You're going to dive in and do this all at once in the following steps.

Step 1: Select GameObject – Create Empty, rename it to **scoring**.

Step 2: Create the **scoring.cs** script for **scoring** and use the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class scoring : MonoBehaviour
{
```

```
public static int score;
public static int lives;
public static int dots;
public static int totaldots;
public static int level;
public static bool initlevel;

// Start is called before the first frame update
void Start()
{
   score = 0;
   lives = 3;
   totaldots = 3; // update when changing number of dots in level
   dots = totaldots;
   level = 1;
   initlevel = false;
}

// Update is called once per frame
void Update()
{
   if (dots == 0)
   {
      initlevel = true;
      level++;
   }
   if (dots == totaldots) initlevel = false;
}

private void OnGUI()
{
   GUI.Box(new Rect(60, 30, 90, 30), "Score:   " + score);
   GUI.Box(new Rect(Screen.width - 130, 30, 90, 30),
       "Lives:  " + lives);
```

```
        GUI.Box(new Rect(Screen.width / 2 - 100, 30, 200, 30),
            "Dots: " + dots);
        GUI.Box(new Rect(60, Screen.height - 50, 90, 30),
            "Level:  " + level);
    }
}
```

Step 3: Replace the code for **dot.cs** with the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class dot : MonoBehaviour
{

    public AudioClip dotsound;
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (scoring.initlevel == true) Revive();
    }

    void Suspend()
    {
        gameObject.transform.position = new Vector3(transform.
                        position.x, 20.0f, transform.position.z);
    }
```

```
    void Revive()
    {
        gameObject.transform.position = new Vector3(transform.
                            position.x, 0.3f, transform.position.z);
        scoring.dots++;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.name == "player")
        {
            AudioSource.PlayClipAtPoint(dotsound,
                transform.position, 1.0f);
            Suspend();
            scoring.dots--;
            scoring.score += 10;
        }
    }
}
```

Step 4: In **player.cs** replace the `Start` function with this code:

```
void InitPosition()
{
    transform.position = new Vector3(0.0f, 0.3f, 0.5f);
    GetComponent<Rigidbody>().velocity = new Vector3(0, 0, 0);
}

// Use this for initialization
void Start()
{
    InitPosition();
}
```

Step 5: Insert the following line at the end of the `Update` function in **player.cs**:

```
if (scoring.initlevel) InitPosition();
```

Step 6: Read the new code, try to understand it and test it.

The essential idea behind this code is to suspend the dots instead of destroying them. They are suspended by placing them behind the camera by setting the Y coordinate to 20, which makes them invisible. This makes revival really easy. You simply change the Y coordinate back to 0.3. This code also adds level advance when all the dots are collected, and it adds 10 points to the score whenever a dot is collected. The lives counter doesn't work just yet, but you can see the display for it.

Next is a game over screen. You'll do this just like in the last chapter.

Step 7: Insert the following code into the `OnGUI` function in **scoring.cs**:

```
GameObject player = GameObject.Find("player");
if (!player)
{
      GUI.Button (new Rect (Screen.width/2 - 200,
                            Screen.height/2 - 50,
                            400, 50),"Game Over");
   }

   if (scoring.level == 3)
   {
      GUI.Button (new Rect (Screen.width/2 - 200,
                            Screen.height/2 - 50,
                            400, 50),"The End");
   }
```

Step 8: Insert the following code at the end of the `Update` function in **player.cs**:

```
if (scoring.lives == 0) Destroy(gameObject);
if (scoring.level == 3) Destroy(gameObject);
```

This code destroys the player when he's out of lives or when the player reaches the ending. Yes, the ending is hardwired at level 3! That's too soon for the real game, but it's OK for now when you're testing.

Step 9: Insert the following line of code into **nastyenemy.cs** into the collision section where the player gets repositioned to the start:

```
scoring.lives--;
```

Step 10: Test the ending and the game over screens.

Now a one-liner for adding difficulty ramping.

Step 11: Replace the `AddForce` line at the end of the `Update` function in nastyenemy.cs with the following:

```
GetComponent<Rigidbody>().AddForce(dir * factor
   * (0.6f + 0.2f * scoring.level));
```

This code ramps the force applied to the enemies, making them more aggressive on higher levels.

Figure 12.7 shows a screen capture when testing the game at level 1.

▲ **FIGURE 12.7** Testing the classic maze game.

Step 12: Test and **save**.

You now have a framework for making the game into something special, your very own creation. The next and final development section gives you some pointers on where to take things from here.

VERSION 0.07: TUNING

It may not seem like it, but you're almost ready to release this game. The first step will implement a scrolling camera, very much like you did in the scrolling shooter project. It also reveals the 3D nature of this game.

Step 1: Create a **camera.cs** script for the Main Camera with the following `Update` function:

```
void Update()
{
   GameObject player = GameObject.Find("player");
   if (player)
   {
      transform.position = new Vector3(
         player.transform.position.x,
         transform.position.y,
         player.transform.position.z
         );
   }
}
```

Step 2: For **Main Camera**, set the **Field of View** to **40**, and the **Position** to **(0, 10, 0.8)**.

Your x and z coordinates should match the starting coordinates of the player.

Step 3: Test.

You can now see that the game looks pretty good this way, but there is a problem with the enemies. They are cutting into the maze walls because of the strange

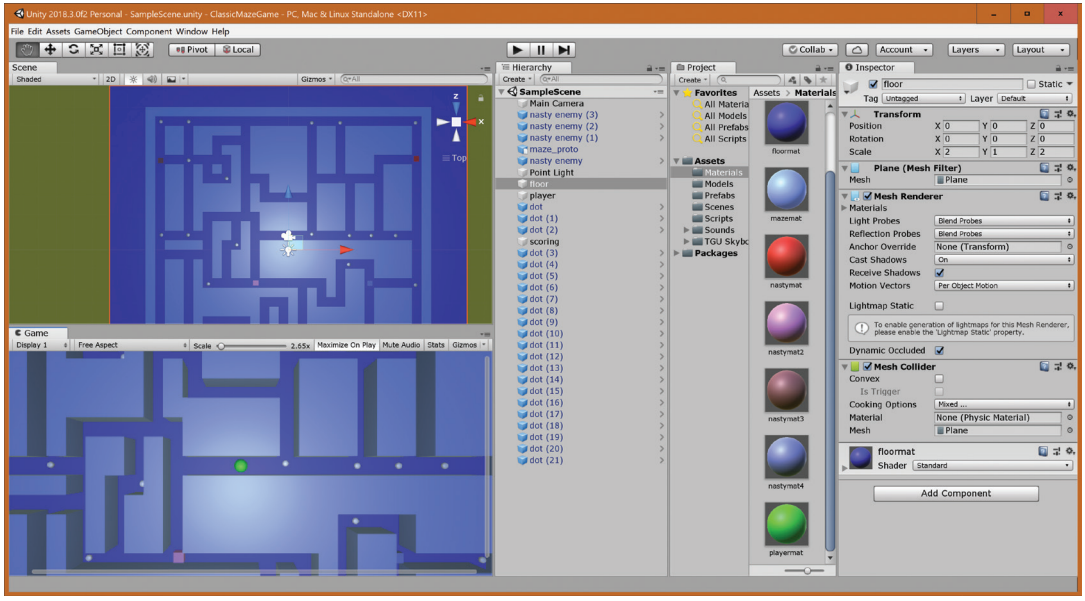
combination of using a sphere collider for a cube. The problem is easily remedied by adjusting the Scale of the object and the radius of the sphere collider.

Step 4: For the nasty enemy prefab, make it a bit smaller by changing the Scale to (0.3, 0.3, 0.3). Also set the Radius of the Sphere Collider to 0.7. Test the effect of this change.

Step 5: Increase the number of dots to at least 20 and spread them throughout the maze. Modify the code to handle this.

Step 6: Make the four enemies different colors.

Your Unity screen should look similar to Figure 12.8.



▲ **FIGURE 12.8** Classic maze game in Unity.

Step 7: Test it now. Can you get to the end?

The game is quite challenging now. If the game feels too hard to you, make the enemies slower, or change the ramping equation. The game is ready for release, or is it? There's always more to be done. The most obvious next step would be to make new mazes in Blender. Maybe there's a bug or two lurking in the game, ready to be

discovered. There's definitely room for exploration with the enemy AI. Check out the exercises at the end of this chapter for more ideas on how to expand the game.

VERSION 1.00: RELEASE AND POSTMORTEM

Finishing a game is just the first step in the release process. For commercial projects, the choice of target platforms, distribution, and marketing are just as critical to a game's success as the game itself. This classic maze game can be viewed as an experimental side project. As such, it can be released to friends and family. It's great to see their reactions. If you're lucky, they'll inundate you with new ideas, some of them good, some not so good, and some won't be new at all because you thought of them yourself earlier. Take notes of all the ideas, and try out the ones that are easy to try out.

This classic maze game was surprisingly easy to make. The original choice of using abstract characters really paid off in terms of speed of development. The technique of using placeholder graphics and sound to develop gameplay first is extremely useful. It's easy to replace graphics or sound after the gameplay is developed. Making your final art and sound first, and then trying to make a fun game out of it is much riskier and potentially very expensive.

The project turned into an original maze game without even trying all that hard. There's very little code, just a maze, a few characters, and some simple sound effects. The game is fun just the same, just as many of the classic arcade games of the seventies and eighties.

You took some serious shortcuts in development, and it shows in a few spots. People expect to see a bunch of different mazes, not just a single maze. The abrupt warping of the player when he dies is jarring, and the game really could use more characters. For a prototype, experimental game, these shortcuts are acceptable. It's up to you to take the development to the next level. After working through all of the projects, you know enough to do just that.

EXERCISES

1. Use Blender to build a 30 x 30 maze. Use the same technique as your first maze, but experiment with different paths. Put in an open area, or a long spiral with a dead end. Then integrate the new maze into the game with the name Maze_2.
2. Use Blender to build a 20 x 20 maze and a 50 x 25 maze. Stretch the maze in Blender so that the width of the maze path matches the prototype maze. Integrate both mazes into the game. Do this by putting both mazes into the scene and moving the player from the first maze to the second when the player removes all the dots from the first scene.
3. Replace the main character graphics with a monkey head. Use Blender to make the monkey head. Optional: Rotate the monkey head to face the current direction of movement.
4. Put in a state machine for the game. When the player dies, have all enemies and the player go back to their starting positions, and put in a three-second delay before gameplay starts again. Optional: During the delay, smoothly move the camera to the initial position.
5. Put in a new character, similar to a bonus fruit in *Pac-Man*. Use Blender to make graphics for it, have it appear at a fixed spot for a limited time, and make it worth 1000 points.
6. Create four different character shapes for your four characters. Do this by adjusting the x, y, and z scales of the renderers, or by replacing the shape with a capsule or a cylinder.
7. Modify nastyenemy.cs to have the enemy aim at a 2D fixed offset from the player. Make the offset variables public vars and assign different offsets to the different enemies.
8. Record yourself saying "waka waka" and take the resulting audio file into Audacity. Modify the recording in Audacity using one or two effects. Then save the sound with the name waka.wav and put it into the game as a looping background track.

Epilogue

It's time to review your achievements. You took a closer look at some of the most influential games from the classic arcade era and used them as inspiration for your own creations. You built five classic games from scratch. You got a taste of what it's like to be a game developer. Last, and certainly not least, you got a step-by-step practical introduction to Unity, Blender, GIMP, and Audacity.

SO MANY GAMES, SO FEW PAGES

Tough choices had to be made when deciding which classic video games to feature in this book. What about *Asteroids*, *Missile Command*, *Defender*, *Q*bert*, and *Crystal Castles*? And then, of course, we shouldn't forget the racing games, video pinball, *Tempest*, and *Venture*.

Some major aspects of game design had to be mostly ignored due to time and space constraints. The worst of these omissions are multiplayer, theming, and story development. Entire books can be and have been written about these subjects.

Multiplayer was typically ignored or handled poorly in the early years of video game development, with the exception of *Pong*. Oddly, the first and most famous arcade game from that time period was a two-player simultaneous game, only to have single-player games dominate the video game scene for dozens of years until the emergence of arcade fighting games.

Story-telling was present in the classic era, but for the most part the hardcore players didn't care about the stories. If there was a story at all, it was often tacked

on by the marketing department and not really an integral part of the game. Good stories make it easier to sell games, but in games such as *Pac-Man* or *Space Invaders*, the game doesn't depend on the player knowing anything about the story. While there are plenty of story-driven games nowadays, back in the day, the stories tended to be ignored by the true gamers. The players cared mainly about how far into the game they could play and their high scores. Most games didn't have an ending, which makes for a poor story.

NOVELTY

Here's the last and possibly most important rule of all:

> Rule 8: Novelty Rule: Make it new!

In the '70s and '80s, novelty was king. It was taken to an extreme by the coin-op industry, especially Atari coin-op. Atari had an internal edict that forced all new coin-op products to be as different as possible from everything that came before. Sequels did happen, but because they usually performed below expectation—a prime example being *Asteroids Deluxe*—Atari management concluded that novelty was an essential ingredient when trying to develop a hit game.

Of course, there were a few really big exceptions, such as *Ms. Pacman* and *Stargate*, but in general, the public clamored for novelty, and the industry responded.

Decades later, novelty is much more difficult to achieve, and sequels are often more successful than the originals. But let's not forget that novelty can really add to the fun, especially when you combine it with good design and high-quality engineering.

HOW MODERN GAMES ARE INFLUENCED BY THE CLASSICS

It may be amusing to look at these quaint old games, but do they still matter? Isn't it all just ancient history? Modern games may seem much more advanced, larger, or

even better (whatever that means) than the old classics, but if you take a closer look, you'll see the influence of the classics in almost every mainstream modern game.

The single most successful game category during the past decade is without a doubt the First-Person Shooter or FPS. When you play one or two of these games, you'll immediately see the connection with the classic 2D shooters. When you summarize it in one sentence, it's essentially the same game mechanic: Shoot them before they shoot you. Modern FPS games implement this with cutting-edge 3D and an epic and complex story thrown in to keep you entertained and motivated in the process.

The basic lessons of classic game development still apply today and for the foreseeable future: Fix your bugs early, make the game fun first before spending too much time polishing the art, and most importantly, test and play your game.

The variety of the classic game spectrum is striking. The game designers from that era were continually innovating and weren't afraid to put crazy new features into their games. This happened, in part, because the designers worked in isolation on different continents and hidden away in secret labs without an internet connection. Nowadays, huge economic pressures at major game studios make it riskier for them to do something radically different. Fortunately, there's a healthy community of independent game developers who aren't quite as afraid of change.

It's a useful exercise to try to design a game that has little or nothing in common with any classic video game. Consider the "null game." It's a game where every player ends up with a score of zero regardless of what happens during the gameplay. It's mostly a theoretical construct, kind of like the old joke game "52 pickup." Two similar games are the "you win" game and the "you lose" game. In these games, there's also no gameplay and the winner, i.e., the score, is predetermined.

You're probably thinking that it's completely crazy to even talk about the "null game," or the "you win" game. But, if you think about it, when a casual player buys some AAA FPS console title because a friend told him it's great, plays it for a few hours and then gives up, isn't that a null game? Didn't that player just treat the game as an interactive experience rather than the game it was designed to be?

From a strictly monetary perspective, the single-player classic arcade games were "you lose" games in the sense that you always lost a quarter. But of course that's ignoring the fairly high importance placed on the numerical score by the classic designers. The classic designers suffered somewhat from the delusion that their players cared about the score. In reality, most players played for fun and didn't really care about the score. Eventually, many of the home console games caved and dropped numerical scoring. Then, to cater to the more fanatic players, they brought scoring back disguised as achievements.

Pretty soon, another 30 years will have flown by and game design will likely have evolved in unpredictable directions. But, games are games. It's my humble opinion that the lessons learned in this book will continue to be useful to future generations of game designers.

Appendix I: Introduction to C# for Beginners

This appendix is a short and quick introduction to C# to help prepare you for the programming parts of this book. Feel free to skip this appendix if you're an experienced C# programmer.

PROGRAMMING IS EASY

Programming is the way humans talk to computers and make them do what they want. Computers can seem to be very stupid, but they're fast, and they have a very good memory. Programming has evolved over the years from hitting toggle switches to creating punch cards to writing text files using increasingly sophisticated computer programming languages.

Programming is easy because computers do exactly what we tell them to do, no more and no less. In that sense, programming is very much like playing the piano. It's easy to hit the keys on the piano and to make the individual notes sound good, but to string the notes together and play piano at a professional level takes years of practice and dedication.

To get started with programming, you need to first learn some of the basic vocabulary. Programming involves the writing of *code*. Code is just another word for the programs, which in turn are text written using the rules of a particular programming language.

Code is broken up into statements. Each statement tells the computer to do something or to set up something for use in other statements. Optionally, there can be

comments that don't affect anything but are there to annotate and document. The program is the collection of all the statements, usually spread out over several text files. In C#, the text files have the .cs extension rather than .txt but they are still text files. Once you're done writing the program you can run it. You hope that the program does what you intended. Very large programming projects can have millions of lines of code written by hundreds of programmers. Fortunately, the classic game programming projects from this book are much smaller than that, encompassing at most a few hundred lines written by you.

INTERPRETED OR COMPILED?

C# (pronounced C sharp) is the programming language used in this book. It is a dialect of the C programming language, so if you're familiar with C or other dialects like Java, PHP, Perl, or C++, you'll notice the similarities.

C# is usually an interpreted language, as opposed to C and C++, which are compiled. The difference between interpreted and compiled is that an interpreted program can be run immediately after a change is made to the program, whereas a compiled program needs the time-consuming additional step of building an executable version of the program before running it.

The main advantage of an interpreted program is that you don't have to wait for a compilation step before running the program. Also, interpreted programs allow for the possibility of changing the program "on the fly," i.e., right in the middle of program execution. Compiled programs tend to run faster, but in recent years this has become less of an issue due to vast improvements in processor speed and the fact that the processors tend to be so busy doing other things that the performance of the scripts is usually unimportant. This is one reason why the developers of Unity chose interpreted languages rather than compiled languages.

After this general introduction, it's time to look at the basic elements of programming.

NUMBERS AND STRINGS

Numbers are the real foundation of programming games. That's because numbers are used to count things, to measure the locations of objects, to describe the graphics, audio, and even the gameplay logic in your games. Strings are sequences of characters like "Hello World!" for example. In game programming, strings are typically used for messages, names of characters, or anything else that needs to be displayed for the player.

There are two main categories of numbers in games: integers and floating-point numbers.

Integers are whole numbers without fractions, for example 3, 1892, or -17. Integers can be negative, positive, or zero. You can write +3 or just 3 to represent the positive integer three.

In programming, integers are used to count things. In game programming, integers typically keep track of the score in a game, or statistics such as lives or levels.

Floating point numbers are numbers with fractional parts, such as 3.452 or -12000.031. Fractions such as ¼ are not directly used in most game code. Instead, you would use 0.25. Floating point numbers are used to represent the approximate position, speed, or length of objects, for example. Floating point numbers can also be entered into code with an exponent using the letter "e." For example, we can write 1.425e20, which is an abbreviation of 1.425 times 10 to the 20th power.

Yes, there are limitations to both integers and floating point numbers because computers are finite machines. It would be inefficient to allow for really huge numbers because they are rarely used in practical applications. In C#, it is possible to go up to 64 bits worth of precision for integers and floating point numbers. In this book, you'll only use 32-bit integers and 32-bit floating point numbers. 32-bit signed integers have a range of about minus two billion to plus two billion. You need to be aware of this if you plan to count things that might get larger than that. If your integers are going to be less than 9 digits long, you're fine with 32 bits. For 32-bit floating point numbers, the limit is 10e38 with seven digits of precision.

It's important to realize that in code there is a difference between 3 and 3.0. The "3" by itself is the integer 3. "3.0" is a floating point number. In math and science, those two numbers are considered to be exactly the same, but not in computer code. The difference shows up when considering expressions such as 9/4. Because both 9 and 4 are integers, the result after the division is also an integer, rounded down to the closest integer, which happens to be 2. However, 9.0/4.0 results in the floating point number 2.25, just what you would expect.

There's one more issue when it comes to floating point numbers in C#. You'll notice that C# game code often has the letter "f" at the end of floating point numbers, for example 3.14159f rather than 3.14159. The f is an indicator that the number uses up 32 bits rather than the C# default 64 bits. The built-in Unity functions tend to use 32-bit floating point numbers, so in order to be compatible with them all code in this book uses 32-bit floats.

Strings are sequences of characters. In Unity the characters are Unicode, the current standard for encoding international characters. Back in the very old days, when memory was at a premium, characters used 6 bits, which was just enough for 26 letters, 10 digits, and a few special characters. Unicode characters use a variable number of bits, ranging from 8 to 32.

Strings are typed into code by surrounding the characters by double quotes, "Hello World!" for example. This string has 12 characters. That's right, the space counts as a character. You will sometimes see escape sequences in strings such as "This is line 1\nfollowed by line 2." The backslash allows for the entry of special characters such as the new-line character. The string "a\nb" has three characters: a, the new-line character, and b.

VARIABLES AND VARIABLE NAMES

Numbers are stored in variables. Variables have names such as "position" or "color." Unlike mathematicians and scientists who tend to use single letters for variables, programmers often use longer variable names. This helps in remembering what all those variables are supposed to represent.

In C#, variable names must start with a letter and capitalization matters, i.e., "Length" and "length" aren't the same variable. Variable names may not contain spaces, so this is why you'll see variable names such as BallSpeed or MyLongVariableName. Other ways of writing variable names you might encounter are ball_speed or ballSpeed. It's important for you to develop a good eye for spelling and capitalization. Countless hours of programmer productivity have been lost because of misspelled variable names.

In many programming languages, including C#, variables must be declared before they can be used. A variable declaration is a statement that tells the program some initial information about the variable. Think of variables as cardboard boxes with giant labels on them. The labels are the variable names, and the contents are the variable values. The declaration is a statement that puts the label onto the box. The declaration also specifies the type of items that are allowed into the box. Here are some samples of variable declarations in C# followed by some code that uses them:

```
int Score;
float Speed;
string PlayerName;
bool isAlive;

Score = 0;
Score = Score + 100;
Speed = 54.9f;
PlayerName = "Joe";
isAlive = true;
```

Boolean variables are used in programming logic and can take on two values: "true" and "false."

Sometimes it's convenient to initialize variables at the same time as declaring them. In C#, this is done, for example, as follows:

```
int Score = 100;
float Speed = 70.0f;
```

```
string PlayerName = "Joe";
bool isAlive = true;
```

WHITESPACE

Whitespace is a programmer's term for spaces, tabs, and linebreaks. In most, but not all programming languages, including C#, all whitespace is equivalent. So, for example:

```
x = 2 + 2;   y=2+x;
```

and

```
x=2+2;

y=2+x;
```

have exactly the same meaning. It takes a little practice to learn where it's OK to insert whitespace. For example, the following statements are *not* the same:

```
MyVariable  = true;
My Variable = true;
```

This is because variable names are not allowed to contain whitespace. The second line is invalid and would cause a compiler error. It is OK to insert whitespace between parts of arithmetic expressions, for example: `X     +     2`

Whitespace is useful for making your code look nice. It's a good idea to avoid tabs because tab settings can change, thus making pretty code look ugly simply by changing the tab settings. It's best to avoid this problem by using spaces instead of tabs in your code.

STATEMENTS AND SEMICOLONS

Statements are usually groups of expressions ending with a semicolon. For example,

```
X = 2;
```

is an example of a simple statement. Why do we have that semicolon at the end? Well, periods are used in numbers and complex variable expressions, so the next best thing is a semicolon. We need the semicolon to separate statements from one another. For example,

```
x = 2; y = 3; z = 4;
```

is a single line of code with three statements in it.

COMPUTATIONS

Computers are really just fancy programmable calculators. Let's learn how to add, subtract, multiply, and divide.

```
x = 2+3;
x = 12-3;
x = 2*12;
x = 7/2;
x = 7.0f/2.0f;
```

Those are the four common computations. The only strange one is multiplication, which is usually done with the star special character on your computer keyboard, or Shift-8. The results of the above computations are 5, 9, 24, 3, and 3.5. The 7 divided by 2 results in a 3 because the inputs are integers.

FUNCTIONS AND FUNCTION CALLS

Functions are a very powerful way to group computations together. Here is an example:

```
void DoubleAndIncrementScore()
{
   score = score * 2;
   score++;
}

score = 1;
```

```
DoubleAndIncrementScore();
DoubleAndIncrementScore();
DoubleAndIncrementScore();
```

This code fragment doubles and increments the score three times. The function definition resides between two curly brackets; the three function calls can occur anywhere else in our code. The function calls change the score from an initial value of 1 to 3, then 7, and finally 15.

You are now ready to watch the following video:

https://unity3d.com/learn/tutorials/topics/scripting/variables-and-functions

This video will give you additional basic examples of functions and variables. Feel free to explore some of the many other video tutorials available at the Unity website, for example *https://unity3d.com/learn/tutorials/s/scripting*

LOOPING

Loops are a great way to do repetitive task. Here is a quick example:

```
int score = 1;
for (int i=0; i<4; i++)
{
   score = score * 2;
}
```

This sequence of code sets the variable `score` to 1, then doubles it four times. The final value of `score` is 16. The variable `i` is set to 0 at the beginning of the loop and is incremented as long as it stays less than 4. You can use the index variable in the loop, for example like this:

```
int score = 1;
for (int i=0; i<4; i++)
{
   score = score + i;
}
```

The final result of this computation is 1+0+1+2+3 = 7.

LEARNING TO CODE

You're now almost ready to start coding. The only way to really learn how to code is to code. A great way for beginners to learn is to follow along with the step-by-step instructions throughout the book. Don't yield to the temptation of just cutting and pasting the code from someplace rather than typing it in. Only by typing each and every line yourself will you experience the joys, thrills, and spills of programming.

If you're a poor typist, stop reading right now and spend a few hours learning the basics on how to touch-type. If you're hunting and pecking with two fingers, you're needlessly handicapping yourself. Most good professional programmers can type at least 50 words per minute. Some are ridiculously fast and can type code faster than you can read it. A famous game developer with over 30 years of coding experience was asked recently which programming course he had found most useful. He immediately answered that it wasn't a programming course at all, but rather the typing course he took as a kid at a local vocational school. These days it's easy to find free typing tutorials and lessons online. Even if you're not aiming to become a professional programmer, touch-typing is a valuable skill if you plan on using a computer keyboard with any frequency.

You might consider yourself an accurate typist, but nobody's perfect. Typos are a fact of life for all programmers. Even if you're a top-notch typist, you're not going to be 100% perfect. All it takes is one single unlucky typo, and your amazing program turns into something completely broken. This isn't like writing an email, where a typo here or there doesn't really matter. Fortunately most typos result in an error that is automatically detected by the programming environment. Sometimes though, a simple typo can result in a bug that can only be found and fixed via extensive testing.

THE CODE IN THIS BOOK

The code in this book is designed to be accessible to beginners. There are no advanced coding concepts here, just some assignment statements, loops, a little bit of easy math, and a few functions here and there. There are places that might be

puzzling to a beginner. That's quite alright. Your goal isn't to understand every line of code immediately, but rather to follow along as best you can and to improve your understanding gradually. After reading this appendix, you are ready to dive in and do the programming steps in this book. If you're new to it, it'll take some patience and perseverance, but there's no feeling quite like writing code, fixing the inevitable bugs, and then having it do exactly what you want.

Appendix II: Eight Rules of Classic Game Design

Rule 1: Simple Rule: Keep it simple.

Rule 2: Immediate Gameplay Rule: Start gameplay immediately.

Rule 3: Difficulty Ramping Rule: Ramp difficulty from easy to hard.

Rule 4: Test Rule: Test the game to make sure it's fun.

Rule 5: Score Rule: Score equals skill.

Rule 6: Experts Rule: Keep experts interested.

Rule 7: Ending Rule: Make an ending.

Rule 8: Novelty Rule: Make it novel.

Appendix III: About the DVD

The DVD contains project files used for creating the games in this book. Please refer to the README file on the DVD for a detailed listing of the contents and further instructions on how to navigate the project files.

The DVD contains image files that correspond to the figures in the book. These image files are provided as an additional resource to the reader.

Last but not least, check out the reference gameplay videos of the projects in this book, and the PC and Mac executables. You might find it helpful (though it's a bit of a spoiler) to look at the videos and play the games before, during, or after you go through the step-by-step creation process.

The files on the DVD are compatible with most Windows and Mac computers.

INDEX

A

Absolute Grid Alignment, 198
"Add Component" box, 17
`AddForce` function, 58, 262
AI. *See* artificial intelligence
`aliencounter` variable, 157
alien death sequence, 149–153
aliens, 126–133
alienscript, 136, 140, 152
alien shots, 133–138
alpha, 113, 116, 118
arcade shooter genre, 167–168
arcade video games, 2–4, 41
Arkanoid, 74
arrowprefab, 123–125
artificial intelligence (AI), 239–240
assets, 33
Assets panel, 19, 33
Asteroids, 3, 40, 106, 238
Audacity, 61–62, 259
  background soundtrack using, 229
  making sound effects, 28–31
  open-source tool, 4–6
audio, 61–64, 229–231, 259–262

B

background soundtrack, using Audacity, 229
ball, 57–59
ball movement, 83–86
`BallRelaunch`, 60, 98, 101
BallScript, 63–65, 83–84
BallScript.collflag variable, 96
`BallScript.launchtimer` variable, 89
Bitmasters, Day One, 44–45
black box testing, 135
Blender, 186
  flying saucer model in, 217–218
  initial screen, 26
  making 3D objects, 25–28
  open-source tool, 4–6
bonus score, 241
Boo languages, 17
boolean variables, 280
`bool` variable, 96
Bounce material, 38
box collider, 59, 60, 66, 87, 132, 137, 219, 227, 255
box modeling, 179
Breakout, 106
  classic arcade video games, 3, 71
  coin-op version, 72
  sequels, 74–75
Breakout, Atari, 71–74
BrickMaker object, 91–93
bricks, 91–95
"bricks balls game" videos, 75
Bucket Fill tool, 24
built-in trig functions, 221
Bushnell, Nolan, 41

C

C# (C sharp), 17–21, 277
  keywords, 21
camera, 50, 194, 230
capping the score, 105–106
CGD. *See* Classic Game Design
checkpoints, 166
chirp sound effect, 260
classic arcade video games, 2–4, 67
Classic Brick Game
  ball movement, 83–86
  bricks, 91–95
  collisions, 86–91
  playable, 95–96
  player, 80–82
  playfield, 77–80
  postmortem, 102–103
  release, 102–103
  scoring, 96–98
  title screen, 99–102
Classic Game Design (CGD) rule, 11
  difficulty ramping rule, 73, 106

ending rule, 238–239
experts rule, 166–167
immediate gameplay rule, 43–44
score rule, 105–106
simple rule, 43
test rule, 73–74
Classic Paddle Game
audio, 61–64
ball, 57–59
first release, 67–68
gameplay screenshot of, 66
paddles, 52–57
playfield, 46–52, 59–61
postmortem, 68–69
scoring, 64–66
Classic Vertical Shooter. *See* vertical shooter
code, 276–277, 284–285
coin-op games, 2, 44
coin-op, Real Atari, 44
coin-specific features, 3
collisions, 86–91
color coding, 18
commercial game projects, 52
compiled programs, 277
computations, 282
Computer Space, 40–41
"configuring Blender for laptop", 9
console games, 3
Console panel, 14
const declarations, 141
consumer group, 44
`CreatePrimitive` statement, 93
Crystal Castles, 240–243, 256
`Custom Icon Size`, 23
cutscenes, 237

D

Dabney, Ted, 41
Debug.Log statement, 100
DEC, 41
`Default` theme, 23
Defender, 3
demo application, in Unity, 31–39
design elements, *Pong*, 42
difficulty ramping rule, 73, 106, 239
Directional Light, 49, 78, 94, 191, 212
DirectX, 12
Dockable Dialogs, 22
dogfooding, 2
dots, 258–259
dots per inch (DPI) monitor, 9
DPI monitor. *See* dots per inch monitor
"Dried mud", 24, 25
`Dying`, 142
Dynamic Friction, 38

E

80s arcade video games, 3
Els, Ernie, 43
ending rule, 238–239
experts rule, 166–167, 239

F

featuritis, 43
field of view, 50
finite state machines (FSM), 138, 140
First-Person Shooter (FPS), 274
flip tool, 115
floating point numbers, 278
flying rockets, 204–211
flying saucers, 217–223
`for` statements, 93
FPS. *See* First-Person Shooter
fractions, 278
Frogger®, 3
FSM. *See* finite state machines
fudge factors, 152–153
function(s), 282–283
`AddForce`, 58, 262
`GetKey`, 262
`GetKeyDown`, 262
`Instantiate`, 124–125
`MakeAliens`, 130, 135, 152
`Mathf.Abs`, 91
`OnGUI`, 20, 21, 65, 142, 145
`OnTriggerEnter`, 65, 87–88, 90, 98, 131, 137, 140, 144, 146–148, 216
`PlayClipAtPoint`, 231

PlayOneShot, 231
Random.Range, 58
Start, 20, 54, 58, 82, 102, 143, 146, 261
Update, 20, 54–55, 60, 81, 82, 121, 130, 136, 143–146, 148, 158–160, 195, 200, 211, 216, 222, 226, 228, 266–268
function calls, 282–283

G

Galaxian, 3
game design, 3, 105
game development tools
Audacity, 4–6
Blender, 4–6
GIMP, 4–6
Unity, 4–6
GameOver, 142, 232
Game panel, 14–16
GamePlay, 141, 236
GetKeyDown function, 262
GetKey function, 262
Ginner, Eric, 243
GNU Image Manipulation Program (GIMP)
animation in, 126
channels dialog in, 113
for 2D graphics, 5
making image, 22–25
open-source tool, 4–6
spaceship in, 112
graphic asset, 5
GridTest, 178, 190, 196
Gubble, 243
GUI Text Object, 13

H

hacker, 228
"Hello World" application program, 9–16, 30, 278
Hierarchy panel, 13, 14, 38
horrible hack, 228

I

Icon Theme, 23
immediate gameplay rule, 43–44
Inspector panel, 14–15, 17–18, 34
Instantiate function, 124–125
Instantiate statement, 215
integers, 278
interpreted program, 277
Iwatami, Toru, 236, 239

J

JavaScript
versus C#, 17
programming for Unity, 17
Jobs, Steve, 71, 75
Joe Cain (JoeC), 243
Joust, 3

L

Lanzinger, Franz, 45, 240, 243
level design, 223–228
levels, 156–161, 262–268
level-select, 166–167
LoadScene statement, 100
looping, 283
loop property, 39
low-poly model, 203

M

Macs, 9, 26
"magic number" code, 228
MakeAliens function, 130, 135, 152, 157
MakeBrickScript, 95–96
marathoning, 106
Mastering Pac-Man, 238
Mat Ball, 83
Mathf.Abs function, 91
Mat Playfield, 77–78
maze, 236, 245–251
maze game, 240–243
audio, 259–262
designing, 244–245

dots, 258–259
maze, 245–251
nasty enemies, 254–258
Pac-Man™, 236–237
player, 251–254
release and postmortem, 270
scoring and levels, 262–268
tuning, 268–270
Microsoft Visual Studio, 54
Missile Command, 3, 106, 238
modeling, 179–186
modern games, 273–275
monkey, 27–28, 31, 34–38
MonkeyMaterial icon, 35–36
"MonoBehaviour", 19
Monodevelop, 19
MudBackground, 33–34

N

nasty enemies, 254–258
`NextLevel`, 142
novelty rule, 273
NTSC frame, 260
"null game", 274
numbers, 278–279

O

`OnGUI` function, 20, 21, 65, 142, 145, 266
`OnTriggerEnter` function, 65, 87–88, 90, 98, 131, 140, 144, 146–148, 216, 223, 224, 226, 230, 233, 261

P

Pac-Man™
classic arcade video games, 3
cutscenes, 237
ending rule, 238–239
first maze game, 236–237
kill screen, 238
maze, 236
sequels and maze game, 240–243
Pac-Man AI, 239–240
Pac-Man Fever, 238
paddles, 52–57
PCs, 2, 9
PDP-1, 41
Physic material, 38, 58
Ping Pong, 40, 43
`PlayClipAtPoint` function, 231
players, 2, 4, 14, 80–82, 251–254
PlayerScript, 90
playfield, 31–35, 37–39, 46–52, 59–61, 77–80, 108–112, 171–179, 196, 199
`PlayOneShot` function, 231
Pluck, 29, 61–63
PNG. *See* Portable Network Graphics
Pole Position, 3
Pong
before, 40–41
classic arcade video games, 3–4
clones, 44
design elements, 42
forty years later, 45
sequels, 44
Pong, Atari, 42–44
Portable Network Graphics (PNG), 24
postmortem, 68–69, 102–103, 162–163, 233–234, 270
prefabs, 119–120, 122, 214–215
preferences window, 22
`PressStart`, 141
programming, 276
with C#, 17–22
Project panel, 14, 33, 38
proportional editing, 174–175, 197
prototype maze, 251
`public class` statement, 20

Q

Quaternion statement, 215

R

rail shooter, 168
`Random.Range` function, 58
release, 102–103, 233–234, 270
`"Result is "` string, 21

`result` variable, 21
Rigidbody, 37, 58, 87, 132, 137, 209, 213, 218, 227, 252
Robichek, Mark, 243
rocket mesh, 203
rockets, 200–204, 223–224
Russell, Steve, 41

S

SampleScene, 14
scene gizmo, 35, 47, 48, 77, 79, 178, 192
Scene panel, 13–16
score milking, 106
score rule, 105–106
scoring, 64–66, 72, 96–98, 231–233, 262–268
Scramble™
  classic arcade video games, 3–4
  experts rule, 166–167
  screen layout, 165
  scrolling shooter, 165–166
  sequels, 167–168
`screenBoundary` variable, 121
ScrollingShip, 186, 191, 194, 227
scrolling shooter, 165–166
  audio, 229–231
  designing, 169–171
  flying rockets, 204–211
  flying saucers, 217–223
  game sketch of, 169
  level 1, 196–199
  level design, 223–228
  playfield, 171–179
  postmortem, 233–234
  release, 233–234
  rockets, 200–204
  scoring, 231–233
  Scramble™, 165–166
  shots, 211–217
  spaceship control, 191–196
  spaceship modeling, 179–186
  spaceship texturing, 186–191
scroll wheel, 9, 26, 27, 33, 36, 79, 172
semicolons, 281–282
sequels, 240–243
shipscript, 137
`shipSpeed` variable, 121
shots, 211–217
simple rule, 43
skybox, 192–193, 250–251
SNES home video game systems, 44–45
sound, 153–156
Space Invaders®, 104–107
  classic arcade video games, 3–4
spaceship, 112–117
  control, 191–196
  modeling, 179–186
  texturing, 186–191
Spacewar!, 41
sphere collider, 37, 38, 58, 259, 269
sprites, 109, 112, 117–125
stamps, 112, 196
starfield, 109–111
`starfield_scroller` script, 110–111
`Start` functions, 20, 54, 58, 82, 102, 143, 146, 261
`StartingPlay`, 141–142
statements, 281–282
  `for`, 93
  `CreatePrimitive`, 93
  `LoadScene`, 100
  `public class`, 20
  `using`, 20
story-telling, 272
strings, 278–279
Subdivision Surface Modifier, 28
Super Breakout, 74

T

Taito, 107
terrain, 224–225, 227
`Testing123` class, 20
test rule, 73–74
Text object, 13
texture file, 33
texturing, 186–191
3D objects for Blender, 4, 6, 24–28
3D modeling methods, 179
3D tech *versus.* 2D tech, 170
`Time.deltaTime` variable, 84

title screen, 99–102
tuning, 268–270
2D graphics, GIMP tools for, 5

U

Unicode characters, 279
Unity, 2
  built-in functions of, 162
  demo application in, 31–39
  game development tools, 4–6
  GridTest in, 178
  "Hello World" application program, 9–16, 30
  installation, 8–9
  maze prototype in, 249
  prints error messages, 14
  programming with C#, 17–22
  scrolling playfield in, 199
  website, 8
Unity Blank Project, 11
`Update` functions, 20, 54–55, 60, 81, 82, 121, 130, 136, 143–146, 148, 158–160, 195, 200, 211, 216, 222, 226, 228, 266–268
`using` statements, 20
UV Image editor, 186–189

V

variable names, 279–281
variables, 279–281
  `aliencounter`, 157
  `BallScript.launchtimer`, 89
  `bool`, 96
  `dir`, 256
  `result`, 21
  `screenBoundary`, 121
  `shipSpeed`, 121
  `Time.deltaTime`, 84
version control, 50
vertical shooter
  alien death sequence, 149–153
  aliens, 126–133
  alien shots, 133–138
  designing, 108
  initial scoring in, 139
  levels, 156–161
  lives, 138–149
  playfield, 108–112
  postmortem, 162–163
  release, 162–163
  scoring, 138–149
  screenshot of, 162
  sketching, 108
  sound effect, 153–156
  spaceship, 112–117
  sprites, 117–125
video games
  arcade. *See* arcade video games
viewport shading, 180, 182, 201, 217
Visual Studio Community Edition, 19

W

Wahwah, 30, 153
WallTopScript, 87–88, 90, 96
`WeirdValue`, 21
white box testing, 135
whitespace, 281
Wozniak, Steve, 71, 75