FROM CONCEPT TO PLAYABLE GAME
WITH UNITY AND C#

Introduction to GAME DESIGN, PROTOTYPING, and DEVELOPMENT

Jeremy GIBSON


cover-image

Contents

Chapter 1. Thinking Like a Designer

You Are a Game Designer

Bartok: A Game Exercise

The Definition of *Game*

Summary

Chapter 2. Game Analysis Frameworks

Common Frameworks for Ludology

MDA: Mechanics, Dynamics, and Aesthetics

Formal, Dramatic, and Dynamic Elements

The Elemental Tetrad

Summary

Chapter 3. The Layered Tetrad

The Inscribed Layer

The Dynamic Layer

The Cultural Layer

The Responsibility of the Designer

Summary

Chapter 4. The Inscribed Layer

Inscribed Mechanics

Inscribed Aesthetics

The Five Aesthetic Senses

Inscribed Narrative

Inscribed Technology

Summary

Chapter 5. The Dynamic Layer

The Role of the Player

Emergence

Dynamic Mechanics

Dynamic Aesthetics

Dynamic Narrative

Dynamic Technology

Summary

Chapter 7. Acting Like a Designer

Iterative Design

Innovation

Brainstorming and Ideation

Changing Your Mind

Scoping!

Summary

Chapter 8. Design Goals

Design Goals: An Incomplete List

Designer-Centric Goals

Player-Centric Goals

Summary

Chapter 10. Game Testing

Why Playtest?

Being a Great Playtester Yourself

The Circles of Playtesters

Methods of Playtesting

Other Important Types of Testing

Summary

Chapter 11. Math and Game Balance

The Meaning of Game Balance

The Importance of Spreadsheets

Examining Dice Probability with Sheets

The Math of Probability

Randomizer Technologies in Paper Games

Weighted Distributions

Permutations

Using Sheets to Balance Weapons

Positive and Negative Feedback

Summary

Chapter 12. Guiding the Player

Direct Guidance

Indirect Guidance

Teaching New Skills and Concepts

Summary

Chapter 13. Puzzle Design

Scott Kim on Puzzle Design

Puzzle Examples in Action Games

Summary

Chapter 16. Thinking in Digital Systems

Systems Thinking in Board Games

An Exercise in Simple Instructions

Game Analysis: Apple Picker

Summary

Chapter 17. Introducing The Unity devELOPMENT environment

Downloading Unity

Introducing Our Development Environment

Launching Unity for the First Time

The Example Project

Setting Up the Unity Window Layout

Learning Your Way Around Unity

Summary

Chapter 18. Introducing Our Language: C#

Understanding the Features of C#

Reading and Understanding C# Syntax

Summary

Chapter 19. Hello World: Your First Program

Creating a New Project

Making a New C# Script

Making Things More Interesting

Summary

Chapter 20. Variables and Components

Introducing Variables

Strongly Typed Variables in C#

Important C# Variable Types

The Scope of Variables

Naming Conventions

Important Unity Variable Types

Unity GameObjects and Components

Summary

Chapter 21. Boolean Operations and Conditionals

Booleans

Comparison Operators

Conditional Statements

Summary

Chapter 22. Loops

Types of Loops

Set Up a Project

`while` Loops

`do...while` Loops

`for` Loops

`foreach` Loops

Jump Statements within Loops

Summary

Chapter 23. Collections in C#

C# Collections

Using Generic Collections

List

Dictionary

Array

Multidimensional Arrays

Jagged Arrays

Whether to Use Array or List

Summary

Chapter 24. Functions and parameters

Set Up the Function Examples Project

Definition of a Function

Function Parameters and Arguments

Returning Values

Proper Function Names

Why Use Functions?

Function Overloading

Optional Parameters

The `params` Keyword

Recursive Functions

Summary

Chapter 30. Prototype 3: Space shmup

Getting Started: Prototype 3

Setting the Scene

Making the Hero Ship

Adding Some Enemies

Spawning Enemies at Random

Setting Tags, Layers, and Physics

Making the Enemies Damage the Player

Restarting the Game

Shooting (Finally)

Summary

Chapter 31. Prototype 3.5: Space shmup Plus

Getting Started: Prototype 3.5

Programming Other Enemies

Shooting Revisited

Showing Enemy Damage

Adding Power-Ups and Boosting Weapons

Making Enemies Drop Power-Ups

Enemy_4—A More Complex Enemy

Adding Particle Effects and Background

Next Steps

Summary

Chapter 32. Prototype 4: Prospector Solitaire

Getting Started: Prototype 4

Build Settings

Importing Images as Sprites

Constructing Cards from Sprites

The Prospector Game

Implementing Prospector in Code

Implementing Game Logic

Adding Scoring to Prospector

Adding Some Art to the Game

Summary

Chapter 33. Prototype 5: Bartok

Getting Started: Prototype 5

Build Settings

Coding Bartok

Summary

Chapter 34. Protype 6: Word Game

Getting Started: Word Game Prototype

About the Word Game

Parsing the Word List

Setting Up the Game

Laying Out the Screen

Adding Interactivity

Adding Scoring

Adding Animation to Letters

Adding Color

Summary

Appendix A. Standard Project Setup Procedure

Setting Up a New Project

Getting the Scene Ready for Coding

Appendix C. Online Reference

Tutorials

Unite Archive

Programming

Searching Tips

Finding Assets

Other Tools and Educational Discounts

Chapter 1. Thinking Like a Designer

Our journey starts here. This chapter presents the basic theories of design upon which the rest of the book is built. In this chapter, you also encounter your first game design exercise and learn more about the underlying philosophy of this book.

You Are a Game Designer

As of this moment, you are a game designer, and I want you to say it out loud:[1]

[1] I thank my former professor Jesse Schell for asking me to make this statement publicly in a class full of people. He also includes this request in his book *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008).

“I am a game designer.”

It’s okay. You can say it out loud, even if other people can hear you. In fact, according to psychologist Robert Cialdini’s book *Influence: The Psychology of Persuasion*,[2] if other people hear you commit to something, you’re more likely to follow through. So, go ahead and post it to Facebook, tell your friends, tell your family:

[2] Robert B. Cialdini. *Influence: The Psychology of Persuasion.* (New York: Morrow, 1993).

“I am a game designer.”

But, what does it mean to be a game designer? This book will help you answer that question and will give you the tools to start making your own games. Let’s start with a design exercise.

Bartok: A Game Exercise

I first saw this exercise used by game designer Malcolm Ryan as part of a Game Design Workshop session at the Foundations of Digital Gaming conference. The goal of this exercise is to demonstrate how even a simple change to the rules of a game can have a massive effect on the experience of playing the game.

Bartok is a simple game played with a single deck of cards that is very similar to the commercial game Uno. In the best case scenario, you would play this game with three friends who are also interested in game design; however, I've also made a digital version of the game for you to play solo. Either the paper or digital version will work fine for our purposes.[3]

[3] The card images in this book and in the digital card games presented in the book are based on Vectorized Playing Cards 1.3, Copyright 2011, Chris Aguilar. Licensed under LGPL 3 – http://www.gnu.org/copyleft/lesser.html, http://code.google.com/p/vectorized-playing-cards/.

Getting the Digital Version of Bartok

There are two good ways to go about getting the digital version of this game. The first and simplest is to simply visit the website for this book:

http://book.prototools.net

You will find the digital version of *Bartok* in the section of the website devoted to Chapter 1.

The second way to get the game is to download the Unity build file for the game and compile it on your own machine. Although this isn't a very difficult task, you should probably only do so if you already know your way around Unity. If you are unfamiliar with Unity at this

point, you can wait to download the build files until you've read the section of the book that covers digital prototyping.

The build files for the *Bartok* game are available at the same location:

http://book.prototools.net

Later in this book, you learn the steps to build a simple digital prototype for *Bartok* (in Chapter 32???, "Prototype 5: Bartok").

You can, of course, also just grab a standard deck of playing cards and two to four friends and play the game in person, which will allow you to talk with your friends about the feel of the game and the changes you want to make to it.

Objective

Be the first player to get rid of all the cards in your hand.

Getting Started

Here are the basic rules for *Bartok*:

1. Start with a regular deck of playing cards. Remove the Jokers, leaving you with 52 cards (13 of each suit ranked Ace–King).

2. Shuffle the deck and deal seven cards to each player.

3. Place the rest of the cards face-down in a draw pile.

4. Pick the top card from the draw pile and place it on the table face-up to start the discard pile.

5. Starting with the player to the left of the dealer and proceeding clockwise, each player must play a card onto the discard pile if

possible, and if she cannot play a card, the player must draw a single card from the draw pile (see Figure 1.1).

Figure 1.1. The initial layout of *Bartok*. In the situation shown, the player can choose to play any one of the cards highlighted with blue borders (7C, JC, 2H, 2S).



6. A player may play a card onto the discard pile if the card is either:

a. The same suit as the top card of the discard pile. (For example, if the top card of the discard pile is a 2 of Clubs (2C), any other Club may be played onto the discard pile.)

b. The same rank as the top card of the discard pile. (For example, if the top card of the discard pile is a 2C, any other 2 may be played onto the discard pile.)

7. The first player to successfully get rid of all her cards wins.

Playtesting

Try playing the game a couple of times to get a feel for it. Be sure to shuffle the cards thoroughly between each playthrough. Games will often result in a somewhat sorted discard pile, and without a good shuffle, subsequent games may have results weighted by the nonrandom card distribution.

Tip

DEBLOCKING Deblocking is the term for strategies used to break up blocks of cards (that is, groups of similar cards). In *Bartok*, each successful game ends with all the cards sorted into blocks of the same suit and blocks of the same rank. If you don’t deblock those groups, the subsequent game will end much faster because players are more likely to be dealt cards that match each other.

According to mathematician and magician Persi Diaconis, seven good riffle shuffles should be sufficient for nearly all games;[4] if you run into issues, though, some of these de-blocking strategies can help.

[4] Persi Diaconis, "Mathematical Developments from the Analysis of Riffle Shuffling," *Groups, Combinatorics and Geometry,* edited by Ivanov, Liebeck, and Saxl. *World Scientific* (2003): 73–97. Also available online at http://statweb.stanford.edu/~cgates/PERSI/papers/Riffle.pdf.

Here are some standard strategies for deblocking a deck of cards if standard shuffling doesn't work:

• Deal the cards into several different piles. Then shuffle these piles together.

• Deal the cards out face-down into a large, spread-out pool. Then use both hands to move the cards around almost like mixing water. This is how dominoes are usually shuffled, and it can help break up your card blocks. Then gather all the cards into a single stack.

• Play *52 Pickup*: Throw all the cards on the floor and pick them up.

Analysis: Asking the Right Questions

After each playtest, it's important to ask the right questions. Of course, each game will require slightly different questions, though many of them will be based on these general guidelines:

• Is the game of the appropriate difficulty for the intended audience? Is it too difficult, too easy, or just right?

• Is the outcome of the game based more on strategy or chance? Does randomness play too strong a role in the game, or,

alternatively, is the game too deterministic so that once one player is in the lead, the other players don't have any chance to catch up?

• Does the game have meaningful, interesting decisions? When it's your turn, do you have several choices, and is the decision between those choices an interesting one?

• Is the game interesting when it's not your turn? Do you have any effect on the other players' turns, or do their turns have any immediate effect on you?

We could ask many other questions, but these are some of the most common.

Take a moment to think about your answers to these questions relative to the games of *Bartok* you just played and write them down. If you're playing the paper version of this game with other human players, it's worthwhile to ask them to write down their own answers to the questions individually and then discuss them after they're written; this keeps the responses from being influenced by other players.

Modifying the Rules

As you'll see throughout this book, game design is primarily a process:

1. Incrementally modify the rules, changing very few things between each playtest.

2. Playtest the game with the new rules.

3. Analyze how the feel of the game is altered by the new rules.

4. Design new rules that you think may move the feel of the game in the direction you want.

5. Repeat this process until you're happy with the game.

Iterative design is the term for this repetitive process of deciding on a small change to the game design, implementing that change, playtesting the game, analyzing how the change affected the gameplay, and then starting the process over again by deciding on another small change. Chapter 7, "Acting Like a Designer," covers iterative design in detail.

For the *Bartok* example, why don't you start by picking one of the following three rule changes and playtesting it:

• **Rule 1:** If a player plays a 2, the person to her left must draw two cards instead of playing.

• **Rule 2:** If any player has a card that matches the number and color (red or black) of the top card, she may announce "Match card!" and play it out of turn. Play then continues with the player to the left of the one who just played the out-of-turn card. This can lead to players having their turns skipped.

For example: The first player plays a 3C (three of Clubs). The third player has the 3S, so she calls "Match card!" and plays the 3S on top of the 3C out-of-turn, skipping the second player's turn. Play then continues with the fourth player.

• **Rule 3:** A player must announce "Last card" when she has only one card left. If someone else calls it first, she must draw two cards (bringing her total number of cards to three).

Choose only one of the rule changes from the previous listing and play the game through a couple times with the new rule. Then have each player write their answers to the four playtest questions. You should also try playing with another one of the rules (although I would recommend still only using one of them at a time when trying a new rule for the first time).

If you're playing the digital version of the game, you can use the check boxes on the menu screen to choose various game options.

Warning

WATCH OUT FOR PLAYTESTING FLUKES A weird shuffle or other external factor can sometimes cause a single play through the game to feel really different from the others. This is known as a fluke, and you want to be careful not to make game design decisions based on flukes. If something you do seems to affect the game feel in a very unexpected way, be sure to play through the game multiple times with that rule change to make sure you're not experiencing a fluke.

Analysis: Comparing the Rounds

Now that you've played through the game with some different rule options, it's time to analyze the results from the different rounds. Look back over your notes and see how each different rule set felt to play. As you experienced, even a simple rule change can greatly change the feel of the game. Here are some common reactions to the previously-listed rules:

• **The original rules**

Many players find the original version of the game to be pretty boring. There are no interesting choices to make, and as the players remove cards from their hands, the number of possible choices dwindles, as well, often leaving the player with only one valid choice for most of the later turns of the game. The game is largely based on chance, and players have no real reason to pay attention to other players' turns because they don't really have any way of affecting each other.

• **Rule 1:** *If a player plays a 2, the person to her left must draw two cards instead of playing.*

This rule allows players to directly affect each other, which generally increases interest in the game. However, whether a player has 2s is based entirely on luck, and each player only really has the ability to

affect the player on her left, which often seems unfair. However, this does make other players' turns a bit more interesting because other players (or at least the player to your right) have the ability to affect you.

• **Rule 2:** *If any player has a card that matches the number and color (red or black) of the top card, she may announce "Match card!" and play it out of turn. Play then continues with the player to the left of the one who just played the out-of-turn card.*

This rule often has the greatest effect on player attention. Because any player has the opportunity to interrupt another player's turn, all players tend to pay a lot more attention to each other's turns. Games played with this rule often feel more dramatic and exciting than those played with the other rules.

• **Rule 3:** *A player must announce "Last card!" when she has only one card left. If someone else calls it first, she must draw two cards.*

This rule only comes into play near the end of the game, so it doesn't have any effect on the majority of gameplay, however, it does change how players behave at the end. This can lead to some interesting tension as players try to jump in and say "last card" before the player who is down to only one card. This is a common rule in both domino and card games where the players are trying to empty everything from their hands because it gives other players a chance to catch up to the lead player if the leader forgets about the rule.

Designing for the Game Feel That You Want

Now that you've seen the effects of a few different rules on *Bartok*, it's time to do your job as a designer and make the game better. First, decide on the feel that you want the game to have: do you want it to be exciting and cutthroat, do you want it to be leisurely and slow, or do you want it to be based more on strategy than chance?

Once you have a general idea of how you want the game to feel, think about the rules that we tried out and try to come up with additional rules that can push the feel of the game in the direction that you wish. Here are some tips to keep in mind as you design new rules for the game:

• Change only one thing in between each playtest. If you change (or even tweak) a number of rules between each play through the game, it can be difficult to determine which rule is affecting the game in what way. Keep your changes incremental, and you'll be better able to understand the effect that each is having.

• The bigger change you make, the more playtests it will take to understand how it changes the game feel. If you only make a subtle change to the game, one or two plays can tell you a lot about how that change affects the feel. However, if it's a major rule change, you will need to test it more times to avoid being tricked by a fluke game.

• Change a number and you change the experience. Even a seemingly small change can have a huge effect on gameplay. For instance, think about how much faster this game would be if there were two discard piles to choose from or if the players started with five cards instead of seven.

Of course, adding new rules is a lot easier to do when playing the card game in person with friends than when working with the digital prototype. That's one of the reasons that paper prototypes can be so important, even when you're designing digital games. The first part of this book discusses both paper and digital design, but most of the examples and design exercises are done with paper games because they can be so much faster to develop and test than digital games.

The Definition of *Game*

Before moving too much further into design and iteration, we should probably clarify what we're talking about when we use terms such as *game* and *game design.* Many very smart people have tried to

accurately define the word game. Here are a few of them in chronological order:

• In his 1978 book *The Grasshopper,* Bernard Suits (who was a professor of philosophy at the University of Waterloo) declares that "a game is the voluntary attempt to overcome unnecessary obstacles."[5]

[5] Bernard Suits, *The Grasshopper* (Toronto: Toronto University Press, 1978), 56.

• Game design legend Sid Meier says that "a game is a series of interesting decisions."

• In *Game Design Workshop,* Tracy Fullerton defines a game as "a closed, formal system that engages players in a structured conflict and resolves its uncertainty in an unequal outcome."[6]

[6] Tracy Fullerton, Christopher Swain, and Steven Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, 2nd ed. (Boca Raton, FL: Elsevier Morgan Kaufmann, 2008), 43.

• In *The Art of Game Design,* Jesse Schell playfully examines several definitions for game and eventually decides on "a game is a problem-solving activity, approached with a playful attitude."[7]

[7] Jesse Schell, *Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 37.

• In the book *Game Design Theory,* Keith Burgun presents a much more limited definition of game: "a system of rules in which agents compete by making ambiguous, endogenously meaningful decisions."[8], [9]

[8] Keith Burgun. *Game Design Theory: A New Philosophy for Understanding Games* (Boca Raton, FL: A K Peters/CRC Press,

2013), 10, 19.

[9] *Endogenous* means inherent to or arising from the internal systems of a thing, so “endogenously meaningful decisions” are those decisions that actually affect the game state and change the outcome. Choosing the color of your avatar’s clothing in *Farmville* is not endogenously meaningful, whereas choosing the color of your clothing in *Metal Gear Solid 4* is, because the color of your clothing affects whether your avatar is visible to enemies.

As you can see, all of these are compelling and correct in their own way. Perhaps even more important than the individual definition is the insight that it gives us into each author’s intent when crafting that definition.

Bernard Suits’ Definition

In addition to the short definition “a game is the voluntary attempt to overcome unnecessary obstacles,” Suits also offers a longer, more robust version:

To play a game is to attempt to achieve a specific state of affairs, using only means permitted by rules, where the rules prohibit use of more efficient in favor of less efficient means, and where the rules are accepted just because they make possible such activity.

Throughout his book, Suits proposes and refutes various attacks on this definition; and having read the book, I am certainly willing to say that he has found the definition of “game” that most accurately matches the way that the word is used in day-to-day life.

However, it’s also important to realize that this definition was crafted in 1978, and even though digital games and roleplaying games existed at this time, Suits was either unaware of them or intentionally ignored them. In fact, in Chapter 9 of *The Grasshopper,* Suits laments that there is no kind of game with rules for dramatic play through which players could burn off dramatic energy (much like

children can burn off excess athletic energy via play of any number of different sports), exactly the kind of play that was enabled by games like *Dungeons & Dragons*.[10]

[10] Suits, *Grasshopper,* 95.

Although this is a small point, it gets at exactly what is missing from this definition: Whereas Suits' definition of game is an accurate definition of the word, it offers nothing to designers seeking to craft good games for others.

For an example of what I mean, take a moment to play Jason Rohrer's fantastic game Passage: http://hcsoftware.sourceforge.net/passage/ (see Figure 1.2). The game only takes 5 minutes to play, and it does a fantastic job of demonstrating the power that even short games can have. Try playing through it a couple times.

Figure 1.2. Passage by Jason Rohrer (released December 13, 2007)



Suits' definition will tell you that yes, this is a game. In fact, it is specifically an "open game," which he defines as a game that has as its sole goal the continuance of the game.[11] In Passage, the goal is to continue to play for as long as possible...or is it? Passage has several potential goals, and it's up to the player to choose which of these she wants to achieve. These goals could include the following:

[11] Suits contrasts these with closed games, which have a specific goal (for example, crossing a finish line in a race or ridding yourself of all your cards in *Bartok*). Suits' example of an open game is the games of make-believe that children play.

• Moving as far to the right as possible before dying (exploration)

• Earning as many points as possible by finding treasure chests (achievement)

• Finding a wife (socialization)

The point of Passage as an artistic statement is that each of these can be a goal in life, and to some extent, these goals are all mutually exclusive. If you find a wife early in the game, it becomes more difficult to get treasure chests because the two of you are unable to enter areas that could be entered singly. If you choose to seek treasure, you will spend your time exploring the vertical space of the world and won't be able to see the different scenery to the right. If you choose to move as far to the right as possible, you won't rack up nearly as much treasure.

In this incredibly simple game, Rohrer exposes a few of the fundamental decisions that every one of us must make in life and demonstrates how even early decisions can have a major effect on the rest of our lives. The important thing here is that he is giving players choice and demonstrating to them that their choices matter.

This is an example of the first of a number of designer's goals that I will introduce in this book: *experiential understanding.* Whereas a linear story like a book can encourage empathy with a character by exposing the reader to the character's life and the decisions that she has made, games can allow players to understand not only the outcome of decisions but also to be complicit in that outcome by giving the player the power and the responsibility of decision and then showing her the outcome wrought by her decisions. Chapter 8, "Design Goals," explores these in much greater depth.

Sid Meier's Definition

By stating that "a game is a series of interesting decisions," Meier is saying very little about the definition of the word *game* (there are many, many things that could be categorized as a series of interesting decisions and yet are not games) and quite a bit about what he personally believes makes for a good game. As the designer of games such as *Pirates*, *Civilization*, *Alpha Centauri*, and many more, Sid Meier is one of the most successful game designers

alive, and he has consistently produced games that present players with interesting decisions. This, of course, raises the question of what makes a decision *interesting.* An interesting decision is generally one where

• The player has multiple valid options from which to choose.

• Each option has both positive and negative potential consequences.

• The outcome of each option is predictable but not guaranteed.

This brings up the second of our designer's goals: to create *interesting decisions.* If a player is presented with a number of choices, but one choice is obviously superior to the others, the experience of deciding which to choose doesn't actually exist. If a game is designed well, players will often have multiple choices from which to choose, and the decision will often be a tricky one.

Tracy Fullerton's Definition

As she states in her book, Tracy is much more concerned with giving designers tools to make better games than she is with the philosophical definition of *game.* Accordingly, her definition of a game as "a closed, formal system that engages players in a structured conflict and resolves its uncertainty in an unequal outcome" is not only a good definition of game but also a list of elements that designers can modify in their games:

• **Formal elements:** The elements that differentiate a game from other types of media: rules, procedures, players, resources, objectives, boundaries, conflict, and outcome.

• **(Dynamic) systems:** Methods of interaction that evolve as the game is played.

• **Conflict structure:** The ways in which players interact with each other.

• **Uncertainty:** The interaction between randomness, determinism, and player strategy.

• **Unequal outcome:** How does the game end? Do players win, lose, or something else?

Another critical element in Fullerton's book is her continual insistence on *actually making games.* The only way to become a better game designer is to make games. Some of the games you'll design will probably be pretty awful—some of mine certainly have been—but even designing a terrible game is a learning process, and every game you create will improve your design skills and help you better understand how to make great games.

Jesse Schell's Definition

Schell defines a game as "a problem-solving activity, approached with a playful attitude." This is similar in many ways to Suits's definition, and like that definition, it approaches the definition of game from the point of view of the player. According to both, it is the playful attitude of the player that makes something a game. In fact, Suits argues in his book that two people could both be involved in the same activity, and to one, it would be a game, whereas to the other, it would not be. Suits example is a foot race where one runner is just running because he wants to take part in the race, but the other runner knows that at the finish line there is a bomb she must defuse before it explodes. According to Suits, although the two runners would both be running in the same foot race, the one who is simply racing would follow the rules of the race because of what Suits calls his *lusory attitude*. On the other hand, the bomb-defusing runner would break the rules of the game the first chance she got because she has a serious attitude (as is required to defuse a bomb) and is not engaged in the game. *Ludus* is the Latin word for play, so Suits proposes the term *lusory attitude* to describe the attitude of one who willingly takes part in playing a game. It is because of their lusory attitude that players will happily follow the rules of a game even though there may be an easier way to achieve the stated goal

of the game (what Suits would call the *pre-lusory goal*). For example, the pre-lusory goal of golf is to get the golf ball into the cup, but there are many easier ways to do so than to stand hundreds of yards away and hit the ball with a bent stick. When people have a lusory attitude, they set challenges for themselves just for the joy of overcoming them.

So, another design goal is to encourage a lusory attitude. Your game should be designed to encourage players to enjoy the limitations placed on them by the rules. Think about why each rule is there and how it changes the player experience. If a game is balanced well and has the proper rules, players will enjoy the limitations of the rules rather than feel exasperated by them.

Keith Burgun's Definition

Burgun's definition of a game as "a system of rules in which agents compete by making ambiguous, endogenously meaningful decisions" is his attempt to push the discourse on games forward from a rut that he feels it has fallen into, by narrowing the meaning of game down to something that can be better examined and understood. The core of this definition is that the player is making choices and that those choices are both ambiguous (the player doesn't know exactly what the outcome of the choice will be) and endogenously meaningful (the choice is meaningful because it has a noticeable effect upon the game system).

Burgun's definition is intentionally limited and purposefully excludes several of the things that many people think of as games (including foot races and other competitions based on physical skill) as well as reflective games like *The Graveyard*, by Tale of Tales, in which the player experiences wandering through a graveyard as an old woman. Both of these are excluded because the decisions in them lack ambiguity and endogenous meaning.

Burgun chooses such a limited definition because he wants to get down to the essence of games and what makes them unique. In

doing so, he makes several good points, including his statement that whether an experience is fun has little to do with the question of whether it is a game. Even a terribly boring game is still a game; it's just a bad game.

In my discussions with other designers, I have found that there can be a lot of contention about this question of what types of things should fall under the term *game.* Games are a medium that has experienced a tremendous amount of growth, expansion, and maturation over the last couple of decades, and the current explosion of independent game development has only hastened the pace. Today, more people with disparate voices and backgrounds are contributing work to the field of games than ever before, and as a result, the definition of the medium is expanding, which is understandably bothersome to some people because it can be seen as blurring the lines of what is considered a game. Burgun's response to this is his concern that it is difficult to rigorously advance a medium if we lack a good definition of what the medium is.

Why Care About the Definition of Game?

In his 1953 book *Philisophical Investigations,* Ludwig Wittgenstein proposed that the term *game*, as it is used colloquially, had come at that time to refer to several very different things that shared some traits (which he likened to a family resemblance) but couldn't be encapsulated in a single definition. In 1978, Bernard Suits attacked this idea by using his book, *The Grasshopper,* to argue very stringently for the specific definition of game that you read earlier in this chapter. However, as Chris Bateman points out in his book *Imaginary Games,* though Wittgenstein used the word *game* as his example, he was really trying to make a larger point: the point that words are created to define things rather than things being created to meet the definition of words.

In 1974 (between the publications of *Philosophical Investigations* and *The Grasshopper*), the philosopher Mary Midgley published a paper titled "The Game Game" in which she explored and refuted the

“family resemblance” claim by Wittgenstein not by arguing for a specific definition of game herself but instead by exploring why the word *game* existed. In her paper, she agrees with Wittgenstein that the word game came into being long after games existed, but she makes the statement that words like game are not defined by the *things* that they encompass but instead by the *needs* that they meet. As she states:

Something can be accepted as a chair provided it is properly made for sitting on, whether it consists of a plastic balloon, a large blob of foam, or a basket slung from the ceiling. Provided you understand the need you can see whether it has the right characteristics, and aptness for that need *is* what chairs have in common.[12]

[12] Mary Midgley. “The Game Game,” *Philosophy 49,* no. 189 (1974): 231–53.

In her paper, Midgley seeks to understand some of the needs that games fulfill. She completely rejects the idea that games are closed systems by both citing many examples of game outcomes that have effects beyond the game and pointing out that games cannot be closed because humans have a reason for entering into them. To her, that reason is paramount. The following are just a few reasons for playing games:

• **Humans desire structured conflict:** As Midgley points out, “The Chess Player’s desire is not for general abstract intellectual activity, curbed and frustrated by a particular set of rules. It is a desire for a particular kind of intellectual activity, whose channel is the rules of chess.” As Suits pointed out in his definition, the rules that limit behavior are there precisely because the challenge of those limitations is appealing to players.

• **Humans desire the experience of being someone else:** We are all acutely aware that we have but one life to live (or at least one at a time), and play can allow us to experience another life. Just as a game of *Call of Duty* allows a player to pretend to experience the life

of a soldier, so too does *The Graveyard* allow the player to pretend to experience the life of an old woman, and playing the role of Hamlet allows an actor to pretend to experience the life of a troubled Danish prince.

• **Humans desire excitement:** Much popular media is devoted to this desire for excitement, be it action films, courtroom dramas, or romance novels. The thing that makes games different in this regard is that the player is actively taking part in the excitement rather than vicariously absorbing it, as is the case for the majority of linear media. As a player, you aren't watching someone else be chased by zombies, you're being chased yourself.

Midgley found it critical to consider the needs that are fulfilled by games in order to understand both their importance in society and the positive and negative effects that games can have on the people who play them. Both Suits and Midgley spoke about the potentially addictive qualities of games in the 1970s, long before video games became ubiquitous and public concern emerged about players becoming addicted. As game designers, it is useful for us to understand these needs and respect their power.

The Nebulous Nature of Definitions

As Midgley pointed out, it is useful to think of the word *game* as being defined by the need that it fills. However, she also stated that a chess player doesn't want to play just any kind of game; he specifically wants to play chess. Not only is it difficult to come up with an all-encompassing definition for game, it's also true that the same word will mean different things to different people at different times. When I say that I'm going to play a game, I usually mean a console or video game; when my wife says the same thing, though, she usually means *Scrabble* or another word game. When my parents say they want to play a game, it means something like Alan R. Moon's *Ticket to Ride* (a board game that is interesting but doesn't require players to be overly competitive with each other), and my in-

laws usually mean a game of cards or dominoes when they use the word. Even within our family, the word has great breadth.

The meaning of the word *game* is also constantly evolving. When the first computer games were created, no one could have possibly imagined the multi-billion-dollar industry that we now have or the rise of the fantastic indie renaissance that we've seen over the past several years. All that they knew was that these things people were doing on computers were kind of like tabletop war board games (I'm thinking of *Space War* here), and they were called "computer games" to differentiate them from the preexisting meanings of game.

The evolution of digital games was a gradual process with each new genre building in some way on the ones which had come before, and along the way, the term *game* expanded further and further to encompass all of them.

Now, as the art form matures, many designers are entering the field from various other disciplines and bringing with them their own concepts about what can be created with the technologies and design methodologies that have been developed to make digital games. (You may even be one of them.) As these new artists and designers enter the space, some of them are making things that are very different from what we think of as a stereotypical game. That's okay; in fact, I think it's fantastic! And, this isn't just my opinion. IndieCade, the international festival of independent games, seeks every year to find games that push the envelope of what is meant by *game.* According to Festival Chair Celia Pearce and Festival Director Sam Roberts, if an independent developer wants to call the interactive piece that she has created a game, IndieCade will accept it as one.[13]

[13] This was stated during the Festival Submission Workshop given by Celia Pearce and Sam Roberts at IndieCade East 2014 and is paraphrased on the IndieCade submissions website at: http://www.indiecade.com/submissions/faq/.

Summary

After all these interwoven and sometimes contradictory definitions, you may be wondering why this book has spent so much time exploring the definition of the word *game.* I have to admit that in my day-to-day work as an educator and game designer, I don't spend a lot of time wrestling with the definitions of words. As Shakespeare points out, were a rose to be named something else, it would still smell as sweet, still have thorns, and still be a thing of fragile beauty. However, I believe that an understanding of these definitions can be critical to you as a designer in the following three ways:

• Definitions help you understand what people expect from your games. This proves especially true if you're working in a specific genre or for a specific audience. Understanding how your audience defines the term will help you to craft better games for them.

• Definitions can lead you to understand not only the core of the defined concept but also the periphery. As you read through this chapter, you encountered several different definitions by different people, and each had both a core and a periphery (i.e., games that fit the definition perfectly (the core) and games that just barely fit the definition (the periphery)). The places where these peripheries don't mesh can be hints at some of the interesting areas to explore with a new game. For example, the area of disagreement between Fullerton and Midgley about whether a game is a closed system highlights the previously untracked ground that in the 2000s grew into alternate reality games (ARGs), a genre centered on perforating the closed magic circle of play.[14]

[14] The first large-scale ARG was *Majestic* (Electronic Arts, 2001), a game that would phone players in the middle of the night and send them faxes and emails. Smaller-scale ARGs include the game *Assassin*, which is played on many college campuses, where players can "assassinate" each other (usually with Nerf or water guns, or by snapping a photo) any time that they are outside of classes. One of

the fun aspects of these games is that they are always happening and can interfere with normal life.

• Definitions can help you speak eloquently with others in the field. This chapter has more references and footnotes than any other in the book because I want you to be able to explore the philosophical understanding of games in ways that are beyond the scope of this one book (especially since this book is really focused on the practicalities of actually making digital games). Following these footnotes and reading the source material can help improve the critical thinking that you do about games.

The Core Lessons of This Book

This book will actually teach you how to design a lot more than just games. In fact, it will teach you how to craft any kind of interactive experience. As I define it:

An interactive experience is any experience created by a designer, inscribed into rules, media, or technology and decoded by people through play.

That makes *interactive experience* a pretty expansive term. In fact, any time that you attempt to craft an experience for people—whether you're designing a game, planning a surprise birthday party, or even planning a wedding—you're using the same tools that you will learn as a game designer. The processes that you will learn in this book are more than just the proper way to approach game design. They are a meaningful way to approach any design problem, and the iterative process of design that is introduced in Chapter 7, "Acting Like a Designer," is *the* essential method for improving the quality of any design.

No one bursts forth from the womb as a brilliant game designer. My friend Chris Swain[15] is fond of saying that "Game design is 1% inspiration and 99% iteration," a play on the famous quote by Thomas Edison. He is absolutely correct, and one of the great things

about game design (unlike the previously mentioned examples of the surprise party and the wedding) is that you get the chance to iterate on your designs, to playtest the game, make subtle tweaks, and play it again. With each prototype you make—and with each iteration of your prototypes—your skills as a designer will improve. Similarly, once you reach the parts of this book that teach digital development, be sure to keep experimenting and iterating. The code samples and tutorials are designed to show you how to make a playable game prototype, but every tutorial in this book will end where your work as a designer should begin. Each one of these prototypes could be built into a larger, more robust, better balanced game, and I encourage you to do so.

[15] Chris Swain co-wrote the first edition of *Game Design Workshop* with Tracy Fullerton and taught the class of the same name at the University of Southern California for many years. He is now an entrepreneur and independent game designer.

Moving Forward

Now that you've experienced a bit of game design and explored various definitions of *game,* it's time to move on to a more in-depth exploration of a few different analytical frameworks that game designers use to understand games and game design. The next chapter explores various frameworks that have been used over the past several years, and the chapter that follows synthesizes those into the framework used throughout the remainder of this book.

Chapter 2. Game Analysis Frameworks

Ludology is the fancy name for the study of games and game design. Over the past decade, ludologists have proposed various analytical frameworks for games to help them understand and discuss the structure and fundamental elements of games and the impact of games on players and society.

This chapter presents a few of the most commonly used frameworks that you should know as a designer.

The next chapter, Chapter 3, "The Layered Tetrad Framwork," synthesizes ideas from these common frameworks into the Layered Tetrad framework used throughout this book.

Common Frameworks for Ludology

The frameworks presented in this chapter include the following:

• **MDA:** First presented by Robin Hunicke, Marc LeBlanc, and Robert Zubek, MDA stands for mechanics, dynamics, and aesthetics. It is the framework that is most familiar to designers working in the field and has important things to say about the difference between how designers and players approach games.

• **Formal, Dramatic, and Dynamic elements:** Presented by Tracy Fullerton and Chris Swain in the book *Game Design Workshop,* the FDD framework focuses on concrete analytical tools to help designers make better games and push their ideas further. It owes a lot to the history of film studies.

• **Elemental tetrad:** Presented by Jesse Schell in his book *The Art of Game Design,* the elemental tetrad splits games into four core elements: mechanics, aesthetics, story, and technology.

Each of these frameworks has benefits and drawbacks, and each has contributed to the Layered Tetrad presented in the following chapter of this book. They are covered here in the order that they were published.

MDA: Mechanics, Dynamics, and Aesthetics

First proposed at the Game Developers Conference in 2001 and formalized in the 2004 paper "MDA: A Formal Approach to Game Design and Game Research,"[1] MDA is the most commonly referenced analytical framework for ludology. The key elements of MDA are its definitions of mechanics, dynamics, and aesthetics; its understanding of the different perspectives from which the designer and player view a game; and its proposal that designers should first approach a game through the lens of aesthetics and then work back toward the dynamics and mechanics that will generate those aesthetics.

[1] Robin Hunicke, Marc LeBlanc, and Robert Zubek, "MDA: A Formal Approach to Game Design and Game Research," in *Proceedings of the AAAI workshop on Challenges in Game AI Workshop* (San Jose, CA: AAAI Press, 2004), http://www.cs.northwestern.edu/~hunicke/MDA.pdf.

Definitions of Mechanics, Dynamics, and Aesthetics

One of the things that can be confusing about the three frameworks presented in this chapter is that they each reuse some of the same words, and each framework defines them slightly differently. MDA defines these terms as follows:[2]

[2] Ibid. p. 2.

• **Mechanics:** The particular components of the game at the level of data representation and algorithms

• **Dynamics:** The runtime behavior of the mechanics acting on player inputs and each other's outputs over time

• **Aesthetics:** The desirable emotional responses evoked in the player when she interacts with the game system[3]

[3] Note that this is a very singular definition of *aesthetics*. No other field or framework defines aesthetics this way. Aesthetics usually refers to the branch of philosophy having to do with notions of beauty, ugliness, etc. And, more colloquially, a design aesthetic is the cohesive intent of a design.

Designer and Player Views of a Game

According to MDA, designers should consider games first in terms of the aesthetics, the emotions that the designer wants players to feel while playing the game. Once a designer has decided on the aesthetics, she will work backwards to the kinds of dynamic play that would prompt those feelings, and finally to the gameplay mechanics that will create those dynamics. Players tend to view the game in the opposite way: first experiencing the mechanics (often by reading the written rules for the game), then experiencing the dynamics by playing the game, and finally (hopefully) experiencing the aesthetics that were initially envisioned by the designer (see Figure 2.1).

Figure 2.1. According to MDA, designers and players view a game from different directions.[4]

[4] Adapted from: Hunicke, LeBlanc, and Zubek, "MDA: A Formal Approach to Game Design and Game Research," 2.



Design from Aesthetics to Dynamics to Mechanics

Based on these differing views, MDA proposes that designers should first approach a game by deciding on the emotional response (aesthetics) that they want to engender in the player and then work backward from that to create dynamics and mechanics that fit this chosen aesthetic.

For example, children's games are often designed to make each player feel like they're doing well and have a chance to win up until the very end. To have this feeling, players must feel that the end of the game is not inevitable and must be able to hope for good luck throughout the game. Keep this in mind when looking at the layout of a *Snakes and Ladders* game.

Snakes and Ladders

Snakes and Ladders is a board game for children that originated in ancient India where it was known as *Moksha Patamu*.[5] The game requires no skill and is entirely based on chance. Each turn, a player rolls one die and moves her counter the number of spaces shown. Counters are not placed on the board initially, so if a player rolls a 1 on her first turn, she lands on the first space of the board. The goal is to be the first player to reach the end of the board (space 100). If a player lands on a space at the start of a green arrow (a ladder), she can move to the space at the end of the arrow (for example, a player landing on the 1 space can move her piece to the 38). If a player lands on the start of a red arrow (a snake), she must move her piece to the space at the end of the arrow (for example, a player landing on space 87 must move her piece all the way down to 24).

[5] Jack Botermans, *The Book of Games: Strategy, Tactics, & History* (New York / London: Sterling, 2008), 19.

In the board layout depicted in Figure 2.2, the positioning of the snakes and ladders is very important. Here are just a few examples of how:

• There is a ladder from 1 to 38. This way, if a player rolls a 1 on her first turn (which would normally feel unlucky), the player can move immediately to 38 and gain a strong lead.

• There are three snakes in the final row of the game (93 to 73, 95 to 75, and 98 to 79). These serve to slow a player who is ahead of the others.

• The snake 87 to 24 and the ladder 28 to 84 form an interesting pair. If a player lands on 28 and moves to 84, her opponents can hope that she will subsequently land on 87 and be forced back to 24. Contrastingly, if a player lands on 87 and moves to 24, she can then hope to land on 28 and be moved back up to 84.

Figure 2.2. A layout for the classic game *Snakes and Ladders*



Each of these examples of snake and ladder placement are based on building hope in players and helping them to believe that dramatic changes in position are possible in the game. If the snakes and ladders were absent from the board, a player who was significantly behind the others would have little hope of catching up.

In this original version of the game, the desired aesthetic is for the players to experience hope, reversal of fortune, and excitement in a game in which the players never make any choices. The mechanic is the inclusion of the snakes and the ladders, and the dynamic is the intersection of the two where the act of the players encountering the mechanics leads to the aesthetic feelings of hope and excitement.

Modifying Snakes and Ladders for More Strategic Play

Young-adult and adult players are often looking for more challenge in games and want to feel that they have won a game not by chance but by making strategic choices along the way. Given that we as designers want the game to feel more strategic and intentional, it is possible to modify the rules (an element of the mechanics) without

changing the board to achieve this aesthetic change. One example of this would be accomplished by adding the following rules:

1. Players each control two pieces instead of one.

2. On her turn, each player rolls two dice.

3. She may either use both dice for one piece or one die for each piece.

4. She may alternatively sacrifice one die and use the other to move one opponent's piece backward the number of spaces shown on the die.

5. If a player's piece lands on the same space as any opponent's piece, the opponent's piece is knocked down one row. (For example, a piece knocked off of 48 would fall to 33, and a piece knocked off 33 would fall to 28 and then take the ladder up to 84!)

6. If a player's piece lands on the same space as her own other piece, the other piece is knocked up one row. (For example, a piece knocked off of 61 could be knocked up to 80 and then follow the ladder to 100!)

These changes allow for a lot more strategic decision making on the part of the players (a change to the dynamic play of the game). With rules 4 and 5 in particular, it is possible to directly hurt or help other players,[6] which can lead to players forming alliances or grudges. Rules 1 through 3 also allow for more strategic decisions and make the game much less susceptible to chance. With the choice of which die to use for either piece, and the option for a player to choose to not move her own pieces, a smart player will never be forced to move her own piece onto a snake.

[6] An example of how this could be used to help another player would be a situation in which knocking another player's piece down a row would land the piece on the beginning of a ladder.

This is but one of many demonstrations of how designers can modify mechanics to change dynamic play and achieve aesthetic goals.

Formal, Dramatic, and Dynamic Elements

Where MDA seeks to help both designers and game critics better understand and discuss games, the framework of formal, dynamic, and dramatic elements[7] was created by Tracy Fullerton and Chris Swain to help students in their Game Design Workshop class at the University of Southern California more effectively design games.

[7] Tracy Fullerton, Christopher Swain, and Steven Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games, 2nd ed.* (Boca Raton, FL: Elsevier Morgan Kaufmann, 2008).

This framework breaks games down into three types of elements:

• **Formal elements:** The elements that make games different from other forms of media or interaction and provide the structure of a game. Formal elements include things like rules, resources, and boundaries.

• **Dramatic elements:** The story and narrative of the game, including the premise. Dramatic elements tie the game together, help players understand the rules, and encourage the player to become emotionally invested in the outcome of the game.

• **Dynamic elements:** The game in motion. Once players turn the rules into actual gameplay, the game has moved into dynamic elements. Dynamic elements include things like strategy, behavior, and relationships between game entities. It's important to note that this is related to the use of the term *dynamics* in MDA but is broader because it includes more than just the runtime behavior of the mechanics.

Formal Elements

Game Design Workshop proposes seven formal elements of games that differentiate them from other forms of media:

• **Player interaction pattern:** How do the players interact? Is the game single-player, one-on-one, team versus team, multilateral (multiple players versus each other, as is the case in most board games), unilateral (one player versus all the other players like some *Mario Party* minigames or the board game *Scotland Yard*), cooperative play, or even multiple individual players each working against the same system?

• **Objective:** What are the players trying to achieve in the game? When has someone won the game?

• **Rules:** Rules limit the players' actions by telling them what they may and may not do in the game. Many rules are explicitly written and included in the game, but others are implicitly understood by all players. (For example, no rule says so, but it's implicitly understood that you can't steal money from the bank in *Monopoly*.)

• **Procedures:** The types of actions taken by the players in the game. A rule in *Snakes and Ladders* tells you to roll the die and move the number of spaces shown. The procedure dictated by that rule is the actual action of rolling the die and moving the piece. Procedures are often defined by the interaction of a number of rules. Some are also outside of the rules; though it is not explicitly defined by the rules of poker, bluffing is an important procedure in the game.

• **Resources:** Resources are elements that have value in the game. These include things like money, health, items, and property.

• **Boundaries:** Where does the game end and reality begin? In his book *Homo Ludens,* John Huizina defines the "magic circle" as a temporary world where the rules of the game apply rather than the rules of the ordinary world. In a sport like football or ice hockey, the magic circle is defined by the boundaries of the playing field; but in an alternative reality game like *I Love Bees* (the ARG for *Halo 2*), the boundaries are more vague.

• **Outcome:** How did the game end? There are both final and incremental outcomes in games. In a game of chess, the final outcome is that one player will win, and the other will lose. In a pen and paper roleplaying game like *Dungeons & Dragons*, there are incremental outcomes when a player defeats an enemy or gains a level, and even death is often not a final outcome since there are ways to resurrect players.

According to Fullerton, another way to look at formal elements is that the game ceases to exist when they are removed. If one removes the rules, outcome, and so on from a game, it ceases to be a game.

Dramatic Elements

Dramatic elements help make the rules and resources more understandable to players and can give players greater emotional investment in the game.

Fullerton presents three types of dramatic elements:

• **Premise:** The basic story of the game world. In *Monopoly*, the premise is that each of the players is a real-estate developer trying to get a monopoly on corporate real estate in Atlantic City, New Jersey. In *Donkey Kong*, the player is trying to single-handedly save his girlfriend from a gorilla that has kidnapped her. The premise forms the basis around which the rest of the game's narrative is built.

• **Character:** Characters are the individuals around whom the story revolves, be it the nameless and largely undefined silent first-person protagonist of games like *Quake* or a character like Nathan Drake, from the *Uncharted* series of games, who is as deep and multidimensional as the lead characters in most movies. Unlike movies, where the goal of the director is to encourage the audience to have empathy for the film's protagonist, in games, the player actually *is* the protagonist character, and designers must choose whether the protagonist will act as an avatar for the player (conveying the emotions, desires, and intentions of the player into

the world of the game and following the wishes of the player) or as a role that the player must take on (so that instead the player acts out the wishes of the game character). The latter is the most common of the two and is much simpler to implement.

• **Story:** The plot of the game. Story encompasses the actual narrative that takes place through the course of the game. The premise sets the stage on which the story takes place.

One of the central purposes of dramatic elements that is not specifically covered in Fullerton's three types is that of helping the player to better understand the rules. In the board game *Snakes and Ladders*, the fact that the green arrows in our diagram are called "ladders" in the game implies that players are meant to move up them. In 1943, when Milton Bradley began publishing the game in the United States, they changed the name to *Chutes and Ladders*.[8] Presumably, this helped American children to better grasp the rules of the game because the chutes (which look like playground slides) were a more obvious path downward than the original snakes, just as the ladders were an obvious path upward.

[8] About.com entry on *Chutes and Ladders* versus *Snakes and Ladders*: http://boardgames.about.com/od/gamehistories/p/chutes_ladders.htm. Last accessed March 1, 2014.

In addition to this, many versions of the game have included images of a child doing a good deed at the bottom of a ladder and an image of her being rewarded for doing so at the top of the ladder. Conversely, the top of chutes depicted a child misbehaving, and the bottom of the chute showed her being punished for doing so. In this way, the narrative embedded in the board also sought to encourage the moral standards of 1940s America. Dramatic elements cover both the ability of the embedded narrative to help players remember rules (as in the case of the snakes being replaced by chutes) and the ability of the game narrative to convey meaning to the players that

persists outside of the game (as was intended by the images of good and bad deeds and their consequences).

Dynamic Elements

Dynamic elements are those that occur only when the game is being played. There are a few central things understand about dynamics in games:

• **Emergence:** Collisions of seemingly simple rules can lead to unpredictable outcomes. Even an incredibly simplistic game like *Snakes and Ladders* can lead to unexpected dynamic experiences. If one player of the game happened to exclusively land on ladders throughout the game where another exclusively landed on snakes, each would have a very different experience of the game. If you consider the six additional rules proposed earlier in this chapter, it's easy to imagine that the range of gameplay experienced by players would expand in size due to the new rules. (For example, now, instead of fate being against player A, perhaps player B would choose to attack A at every possible opportunity, leading to a very negative play experience for A.) Simple rules lead to complex and unpredictable behavior. One of a game designer's most important jobs is to attempt to understand the emergent implications of the rules in a game.

• **Emergent narrative:** In addition to the dynamic behavior of mechanics covered in the MDA model, Fullerton's model also recognizes that narrative can also be dynamic, with a fantastic breadth of narratives emerging from the gameplay itself. Games, by their nature, put players in extra-normal situations, and as a result, they can lead to interesting stories. This is one of the central appeals of pen and paper roleplaying games like *Dungeons & Dragons*, in which a single player acts as the Dungeon Master and crafts a scenario for the other players to experience and characters for them to interact with. This is different from the embedded narrative covered by Fullerton's dramatic elements and is one of the entertainment possibilities that is unique to interactive experiences.

• **Playtesting is the only way to understand dynamics:** Experienced game designers can often make better predictions about dynamic behavior and emergence than novice designers, but no one understands exactly how the dynamics of a game will play out without playtesting them. The six additional rules proposed for *Snakes and Ladders* seem like they would increase strategic play, but it is only through several rounds of playtests that one could determine the real effect the rules changes would have on the game. Repeated playtesting reveals information about the various dynamic behaviors that a game could have and helps designers understand the range of experiences that could be generated by their game.

The Elemental Tetrad

In *The Art of Game Design: a Book of Lenses,*[9] Jesse Schell describes the elemental tetrad, through which he presents his four basic elements of games:

[9] Jesse Schell, *The Art of Game Design: a Book of Lenses* (Boca Raton, FL: CRC Press, 2008).

• **Mechanics:** The rules for interaction between the player and the game. Mechanics are the elements in the tetrad that differentiate games from all noninteractive forms of media (like film or books). Mechanics contain things like rules, objectives, and the other formal elements described by Fullerton. This is different from the *mechanics* presented by MDA because Schell's use of the term differentiates between game mechanics and the underlying technology that enables them.

• **Aesthetics:** Aesthetics describe how the game is perceived by the five senses: vision, sound, smell, taste, and touch. Aesthetics cover everything from the soundtrack of the game to the character models, packaging, and cover art. This is different from MDA's use of the word *aesthetics* because MDA used the word to refer to the emotional response engendered by the game, while Schell uses the

word to refer to things that are crafted by the game developers like actual game art and sound.

• **Technology:** This element covers all the underlying technology that makes the game work. While this most obviously refers to things such as console hardware, computer software, rendering pipelines, and such, it also covers technological elements in board games. Technology in board games can include things like the type and number of dice that are chosen, whether dice or a deck of cards are used as a randomizer, and various stats and tables used to determine the outcome of actions. In fact, the Technology Award at the IndieCade game conference in 2012 went to Zac S. for *Vornheim*, a collection of tools—in the form of a printed book—to be used by game masters when running tabletop roleplaying games set in a city.[10]

[10] http://www.indiecade.com/2012/award_winners/.

• **Story:** Schell uses the term story to convey everything covered by Fullerton's dramatic elements, not just what she terms "story." Story is the narrative that occurs in your game and includes both premise and characters as well.

Schell arranges these elements into the tetrad shown in Figure 2.3.

Figure 2.3. The elemental tetrad by Jesse Schell[11]



[11] Adapted from: Schell, *The Art of Game Design,* 42.

The tetrad shows how the four elements all interrelate with each other. In addition, Schell points out that the aesthetics of the game are always very visible to the player (again, this is different from the aesthetic feelings described in MDA), and the technology of the game is the least visible with players generally having a better understanding of the game mechanics (e.g., the way that snakes and ladders affect the position of the player) than game technology

(e.g., the probability distribution of a pair of six-sided dice). Schell's tetrad does not touch on dynamic play of the game and is more about the static elements of the game as it comes in a box (in the case of a board game) or on disk. Schell's elemental tetrad is discussed and expanded considerably in the next chapter as it forms the elemental aspect of the Layered Tetrad.

Summary

Each of these frameworks for understanding games and interactive art is approaching the understanding of games from a different perspective:

• MDA seeks to demonstrate and concretize the idea that players and designers approach games from different directions and proposes that designers can be more effective by learning to see their games from the perspective of their players.

• Formal, dramatic, and dynamic elements breaks game design into specific components that can each be considered and improved. It is meant to be a designer's toolkit and to enable designers to isolate and examine all the parts of their games that could be improved. FDD also asserts the primacy of narrative in player experience.

• The elemental tetrad is more of a game developer's view on games. It separates the basic elements of a game into the sections that are generally assigned to various teams: game designers handle mechanics, artists handle aesthetics, writers handle story, and programmers handle technology.

In the following chapter, the Layered Tetrad is presented as a combination of and expansion on the ideas presented in all of these frameworks. It is important to understand these frameworks as the underlying theory that led to the Layered Tetrad, and I strongly recommend reading the original paper and books in which they were presented.

Chapter 3. The Layered Tetrad

The previous chapter presented you with various analytical frameworks for understanding games and game design. This chapter presents the Layered Tetrad, a combination and extension of many of the best aspects of those frameworks. Each layer of the Layered Tetrad is further expanded in one of the following chapters.

The Layered Tetrad is a tool to help you understand and create the various aspects of a game. It will help you to analyze games that you love and will help you to look at your game holistically, leading to an understanding of not only the game's mechanics but also their implications in terms of play, socialization, meaning, and culture.

The Layered Tetrad is an expansion and combination of the ideas expressed by the three game analysis frameworks presented in the previous chapter. The Layered Tetrad does not define what a game is. Rather, it is a tool to help you understand all the different elements that need to be designed to create a game and what happens to those elements both during play and beyond as the game becomes part of culture.

The Layered Tetrad is composed of four elements—as was Schell's elemental tetrad—but those four elements are experienced through three layers. The first two, inscribed and dynamic, are based on the division between Fullerton's formal and dynamic elements. In addition, a third cultural layer is added that covers the game's life and effects outside of play, providing a link between game and culture that is critical to understand for us to be responsible game designers and creators of meaningful art.

Each of the layers is described briefly in this chapter, and each layer has a chapter devoted to it later in the book.

The Inscribed Layer

The *inscribed* layer of the tetrad (see Figure 3.1) is very similar to Schell's elemental tetrad. The definitions of the four elements are similar to Schell's, but they are limited to the aspects of the game that exist even when it is not being played.

Figure 3.1. The inscribed layer of the Layered Tetrad[1]



[1] Adapted from: Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 42.

• **Mechanics:** The systems that define how the player and the game will interact. This includes the rules of the game and the following additional formal elements from Fullerton's book: player interaction patterns, objectives, resources, and boundaries.

• **Aesthetics:** Aesthetics describe how the game looks, sounds, smells, tastes, and feels. Aesthetics cover everything from the soundtrack of the game to the character models, packaging, and cover art. This definition differs from the use of the word *aesthetics* in the MDA (Mechanics, Dynamics, Aesthetics) framework because the MDA used the word to refer to the emotional response engendered by the game, whereas Schell and I use the word to refer to game elements that are sensed by the player like art and sound assets.

• **Technology:** Just as with Schell's technology element, this element covers all the underlying technology that makes the game work for both paper and electronic games. For digital games, the technology element is primarily developed by programmers, but it is vital for designers to understand this element because the technology written by programmers forms the possibility space of decisions that can be made by game designers. This understanding is also critical because a seemingly simple design decision (for example, let's move this level from solid ground onto a rocking ship

in a massive storm) can require thousands of hours of development time to implement.

• **Narrative:** Schell uses the term *story* in his elemental tetrad, but I've chosen to use the broader term *narrative* to encompass the premise and characters in addition to the plot and to be more in-line with Fullerton's use of these terms. The inscribed narrative includes all pre-scripted story and pre-generated characters that are in the game.

The Dynamic Layer

As in Fullerton's book *Game Design Workshop,* the *dynamic layer* (see Figure 3.2) emerges when the game is played.

Figure 3.2. The dynamic layer positioned relative to the inscribed layer



As you can see in the figure, it is players who take the static inscribed layer of the game and from it construct the dynamic layer. Everything in the dynamic layer arises from the game during play, and the dynamic layer is composed of both elements in the players' direct control and of the results of their interaction with the inscribed elements. The dynamic layer is the realm of *emergence*, the phenomenon of complex behavior arising from seemingly simple rules. The emergent behavior of a game is often difficult to predict, but one of the great skills of game design that you will build over time is the ability to do so, or at least make pretty good guesses. The four dynamic elements are:

• **Mechanics:** Whereas inscribed mechanics covered rules, objectives, and so on, the dynamic mechanics cover how the players interact with those inscribed elements. Dynamic mechanics include procedures, strategies, emergent game behavior, and eventually the outcome of the game.

• **Aesthetics:** Dynamic aesthetics cover the way that aesthetic elements are generated for the player during play. This includes everything from procedural art (digital game art or music generated on the fly by computer code) to the physical strain that can result from having to mash a button repeatedly over a long period of time.

• **Technology:** Dynamic technology describes the behavior of the technological components of a game during play. This covers how a pair of dice never actually seems to generate the smooth bell curve of results predicted by mathematical probability. It also covers nearly everything that is done by computer code in digital games. One specific example of this could be the performance of the game's artificial intelligence code for enemies, but in its broadest sense, dynamic technology covers absolutely everything that a digital game's code does once the game is launched.

• **Narrative:** Dynamic narrative refers to stories that emerge procedurally out of game systems. This can mean an individual player's path through a branching scripted narrative such as *L.A. Noire* or *Heavy Rain*, the family story created by a play through *The Sims*, or the stories generated by team play with other human players. In 2013, the Boston Red Sox baseball team went from worst to first in a story that mirrored the city of Boston's recovery from the bombing at the 2013 Boston Marathon. That kind of story, enabled by the rules of professional baseball, also fits under dynamic narrative.

The Cultural Layer

The third and final layer of the Layered Tetrad is cultural, and it describes the game beyond play (see Figure 3.3). The cultural layer covers both the impact of culture upon the game and the impact of the game upon culture. It is the community of players around the game that moves it into the cultural layer, and it is at this point that the players actually have more control and ownership over the game than the designers, but it is also through this layer that our societal responsibility as designers becomes clear.

Figure 3.3. The cultural layer exists at the collision of the game and society.



The delineations between the four elements are much blurrier in the cultural layer, but it's still worthwhile to approach this layer through the lens of the four elements:

• **Mechanics:** The simplest form of cultural mechanics is represented by things like *game mods* (modifications to a game that are created by players to affect the inscribed mechanics of the game). This also covers things as complex as the impact that the emergent play of a game can have on society. For instance, the much maligned ability for the player character in *Grand Theft Auto 3* to sleep with a prostitute and then kill her to get his money back was a result of emergent dynamic mechanics in the game, but it had a massive impact on the perception of the game by the general public (which is part of the cultural layer).

• **Aesthetics:** As with the mechanics, cultural aesthetics can cover things like fan art, remixes of the music for the game or other aesthetic fan activities like cosplay (short for costume play, when fans of the game dress in costume to resemble game characters). One key point here is that *authorized* transmedia properties (i.e., a conversion of the game's intellectual property to another medium by the owners of that IP, such as the movie version of *Tomb Raider*, a *Pokémon* lunchbox, etc.) are not part of the cultural layer. This is because authorized transmedia properties are controlled by the original owners of the game's intellectual property, while cultural aesthetics are controlled and created by the community of the game's players.

• **Technology:** Cultural technology covers both the use of game technologies for non-game purposes (e.g., flocking algorithms for game characters could also be used in robotics) and the ability of technology to affect the game experience. Back in the days of the NES (Nintendo Entertainment System), having an Advantage or Max

controller gave the player the ability to press turbo buttons (which was an automated method of pressing the regular A or B controller buttons very rapidly). This was a massive advantage in some games and had an effect on the game experience. Cultural technology also covers the expansion of possibilities of what *game* can mean by continually expanding the possibility space of gaming and the technological aspects of mods made by players to alter the inscribed elements of a game.

• **Narrative:** Cultural narrative encompasses the narrative aspects of fan-made transmedia properties created from the game (e.g., fan fiction, the narratives of fan-made tribute movies, and the fan-made characters and premises that are part of some game mods). It also covers the stories told about the game in culture and society, including both the stories that vilify games like *Grand Theft Auto* and the stories that extol the virtues and artistic merit of games like *Journey* and *Ico*.

The Responsibility of the Designer

All designers are aware of their responsibility for the formal layer of the game. It's obvious that the developers of the game must include clear rules, interesting art, and so on to enable and encourage the game to be played.

At the dynamic layer, some designers get a little muddier about their responsibility. Some designers are surprised by the behavior that emerges out of their games and want to pass responsibility for that behavior on to the players. For example, a few years ago, Valve decided to give hats to players of their game *Team Fortress 2*. The mechanic they chose was to randomly reward hats to players that were logged in. Because the distribution of hats was based exclusively on whether the player was logged in to a game at the right time, servers sprouted up that had players camping in them, not actually playing the game, just waiting for hat drops. Valve saw this behavior and chose to punish the players for it by taking hats back

from any player that they suspected of having camped on a server rather than actually playing the game.

One way of interpreting this is to see the players as trying to cheat the game. However, another is to realize that the players were just engaging in the most efficient method for obtaining hats as defined by the rules for hat drops that Valve had created. Because the system was designed to give players hats any time they were online regardless of whether they were actually doing anything, the players settled on the easiest path to get the hats. The players may not have honored the intent of the designers of the hat drop system, but they didn't cheat the system itself. Their dynamic behavior was exactly what was implied by the rules of the system that Valve set in place. As you can see from this example, the designer is also responsible for the experience at the dynamic layer through the implications of the systems she designs. In fact, one of the most important aspects of game design is the anticipation and crafting of the dynamic player experience. Of course, doing so is a very difficult task, but that's part of what makes it interesting.

So, what is the designer's responsibility at a cultural level? As a result of most game designers rarely if ever considering the cultural layer, video games are generally regarded in society as puerile and vulgar, selling violence and misogyny to teenage boys. You and I know that this doesn't have to be the case and that it isn't actually true of many or even most games, but this is the ubiquitous perception among the general public. Games can teach, games can empower, and games can heal. Games can promote pro-social behavior and help players learn new skills. A ludic attitude and some quickly devised rules can make even the most dull task enjoyable. As a designer, you are responsible for what your game says to society about gaming and for the impact that it has on players. We have become so good at making games compelling that some players are addicted to them to their detriment. Some designers have even made games that attempted to scam children into spending hundreds or thousands of dollars (eventually leading to a massive class-action lawsuit in at least one case). This kind of

behavior by designers damages the reputation of games in society and prevents many people from considering games worthy of either their time or of being regarded as art, and that's truly sad. I believe that it is our responsibility as designers to promote pro-social, thoughtful behavior through our games and to respect our players and the time that they dedicate to experiencing what we create.

Summary

As demonstrated in this chapter, it's important to explicitly realize that the three layers of the Layered Tetrad represent a transition of ownership from the developers of the game to the players of the game. Everything in the inscribed layer is owned, developed, and implemented by the game designers and developers. The inscribed layer is completely within the developers' control.

The dynamic layer is the point at which the game is actually experienced, so game designers require that players take action and make decisions for the games inscribed by the designers to actually be experienced. Through players' decisions and their effect on game systems, players take some ownership of the experience, yet that experience is still subject to the inscribed decisions of the developers. Thus, the ownership over the dynamic layer is shared between the developers and the players.

At the cultural layer, the game is no longer under the developers' control. This is why things like game mods fit in the cultural layer; through a game mod, a player is taking control of and changing inscribed aspects of the game. Of course, most of the original inscribed game still remains, but it is the player (as mod developer) who determines which inscribed elements she chooses to leave and which she chooses to replace; the player is in control. This is also why I have excluded authorized transmedia from the cultural layer. The developers and owners of the inscribed game maintain ownership over the authorized transmedia, and the cultural layer is defined by the shift of ownership to the players and the communities that surround the game. Additionally, the aspect of the cultural layer

that covers the perception of the game by non-players in society is also largely controlled by the player community's representation of their gameplay experience. People who don't play a game have their opinion of that game shaped by the media they read which was (hopefully) written by people who did actually play the game. However, even though the cultural layer is largely controlled by players, the developers and designers of a game still have an important influence over and responsibility for the game and its effect on society.

The next three chapters each cover one layer of the Layered Tetrad and reveal it in more detail.

Chapter 4. The Inscribed Layer

This is the first of three chapters that explore the layers of the Layered Tetrad in greater depth.

As you learned in Chapter 3, "The Layered Tetrad," the inscribed layer covers all elements that are directly designed and encoded by game developers.

In this chapter on, we look at the inscribed aspects of all four elements: mechanics, aesthetics, narrative, and technology.

Inscribed Mechanics

The inscribed mechanics are most of what one would think of as the traditional job of the game designer. In board games, this includes designing the board layout, the rules, the various cards that might be used and any tables that could be consulted. Much of the inscribed mechanics are described very well in Tracy Fullerton's book *Game Design Workshop* in her chapter on formal elements of games, and for the sake of lexical solidarity (and my distaste for every game design book using different terminology), I reuse her terminology throughout this section of the chapter as much as the Layered Tetrad framework allows.

As mentioned in Chapter 2, "Game Analysis Frameworks," Tracy Fullerton lists seven formal elements of games in her book: player interaction patterns, objectives, rules, procedures, resources, boundaries, and outcomes. In the formal, dramatic, and dynamic elements framework, these seven elements constitute the aspects that make games different from other media.

Inscribed mechanics are a bit different from this, though there is a lot of overlap because mechanics is the element of the tetrad that is unique to games. However, the core of the inscribed layer is that

everything in it is intentionally designed by a game developer, and the mechanics are no exception. As a result of this, inscribed mechanics does not include procedures or outcomes (although they are part of Fullerton's formal elements) because both are controlled by the player and therefore part of the dynamic layer. We'll also add a couple of new elements to give us the following list of inscribed mechanical elements:

• **Objectives:** Objectives cover the goals of the players in the game. What are the players trying to accomplish?

• **Player relationships:** Player relationships define the ways that players combat and collaborate with each other and the game. How do the players' objectives intersect, and does this cause them to collaborate or compete?

• **Rules:** Rules specify and limit player actions. What can and can't the players do to achieve their objective?

• **Boundaries:** Boundaries define the limits of the game and relate directly to the magic circle. Where are the edges of the game? Where does the magic circle exist?

• **Resources:** Resources include assets or values that are relevant within the boundaries of the game. What does the player own in-game that enables her in-game actions?

• **Spaces:** Spaces define the shape of the game space and the possibilities for interaction therein. This is most obvious in board games, where the board itself is the space of the game.

• **Tables:** Tables define the statistical shape of the game. How do players level up as they grow in power? What moves are available to a player at a given time?

All of these inscribed mechanical elements interact with each other, and overlap certainly exists between them (e.g., the tech tree in *Civilization* is a table that is navigated like a space). The purpose of

dividing them into these seven categories for this book is to help you as a designer think about the various possibilities for design in your game. Not all games have all elements, but as with the “lenses’ in Jesse Schell’s book *The Art of Game Design: A Book of Lenses,* these inscribed mechanical elements are seven different ways to look at the various things that you can design for a game.

Objectives

While many games have an apparently simple objective—to win the game—in truth, every player will be constantly weighing several objectives every moment of your game. These can be categorized based on their immediacy and their import to the player, and some objectives may be considered very important to one player while being less important to another.

Immediacy of Objectives

As shown in the image in Figure 4.1 from the beautiful game Journey by thatgamecompany, nearly every screen of a modern game presents the player with short-, mid-, and long-term objectives.

Figure 4.1. Short-, mid-, and long-term objectives in the first level of *Journey* with objectives highlighted in green, blue, and purple respectively



In the short term, the player wishes to charge her scarf (which enables flying in *Journey*), so she’s singing (the white sphere around her), which draws the highlighted scarf pieces to her. She also is drawn to explore the nearby building. For mid-term goals, there are three additional structures near the horizon. Because the rest of the desert is largely barren, the player is attracted to the ruins on the horizon (this *indirect guidance* strategy is used several times throughout *Journey* and is analyzed in Chapter 13, “Guiding the Player”). And, for long-term goals, the player is shown the mountain

with the shaft of light (shown in the top-left corner of Figure 4.1) in the first few minutes of the game, and it is her long-term goal throughout the game to reach the top of this mountain.

Importance of Objectives

Just as objectives vary in immediacy, they also vary in importance to the player. In an open-world game like *Skyrim* by Bethesda Game Studios, there are both primary and optional objectives. Some players may choose to exclusively seek the primary objectives and can play through *Skyrim* in as little as 10 to 20 hours, whereas others who wish to explore various side-quests and optional objectives can spend more than 400 hours in the game without exhausting the content (and even without finishing the primary objectives). Optional objectives are often tied to specific types of gameplay; in *Skyrim*, there is a whole series of missions for players who wish to join the Thieves Guild and specialize in stealth and theft. There are also other series of missions for those who wish to focus on archery or melee[1] combat. This ensures that the game can adapt to the varying gameplay styles of different players.

[1] This is a word that is often mispronounced by gamers. The word *melee* is pronounced “may-lay.” The word *mealy* (pronounced “mee-lee”) means either pale or in some other way like grain meal (e.g., cornmeal).

Conflicting Objectives

As a player, the objectives that you have will often conflict with each other or compete for the same resources. In a game like *Monopoly*, the overall objective of the game is to finish the game with the most money, but you must give up money to purchase assets like property, houses, and hotels that will eventually make you more money later. Looking at the design goal of presenting the player with interesting choices, a lot of the most interesting choices that a player

can make are those that will benefit one objective while hurting another.

Approaching it from a more pragmatic perspective, each objective in the game will take time to complete, and a player may only have a certain amount of time that she is willing to devote to the game. Returning to the *Skyrim* example, many people (myself included) never finished the main quest of *Skyrim* because they spent all of their time playing the side quests and lost track of the urgency of the main story. Presumably, the goal of *Skyrim's* designers was to allow each player to form her own story as she played through the game, and it's possible that the designers wouldn't care that I hadn't finished the main quest as long as I enjoyed playing the game, but as a player, I felt that the game ended not with a bang but a whimper as the layers upon layers of quests I was given had seemingly smaller and smaller returns. If, as a designer, it's important to you that your players complete the main quest of the game, you need to make sure that the player is constantly reminded of the urgency of the task and (unlike many open world games) you may need to have consequences for the player if she does not complete the main quest in a timely manner. As an example, in the classic game *Star Control*, if the player did not save a certain alien species within a given amount of time from the start of the game, the species' planet actually disappeared from the universe.

Player Relationships

Just as an individual player has several objectives in mind at any given time, the objectives that players have also determine relationships between them.

Player Interaction Patterns

In *Game Design Workshop,* Fullerton lists seven different player interaction patterns:

• **Single player versus game:** The player has the objective of beating the game.

• **Multiple individual players versus game:** Several co-located players each have the objective of beating the game, but they have little or no interaction with each other. This can often be seen in MMORPGs (massively multiplayer online roleplaying games) like *World of Warcraft* when players are each seeking to succeed at their missions in the same game world but not interacting with each other.

• **Cooperative play:** Multiple players share the common objective of beating the game together.

• **Player versus player:** Each of two players has the objective of defeating the other.

• **Multilateral competition:** The same as player versus player, except that there are more than two players, and each player is trying to defeat all the others.

• **Unilateral competition:** One player versus a team of other players. This can be seen in the board game *Scotland Yard* (also called *Mr. X*), where one player plays a criminal trying to evade the police and the other 2 to 4 players of the game are police officers trying to collaborate to catch the criminal.

• **Team competition:** Two teams of players, each with the objective of beating the other.

Some games, like BioWare's *Mass Effect*, provide computer-controlled allies for the player. In terms of designing player interaction patterns, these computer-controlled allies can either be thought of as an element of the single player's abilities in the game or as proxies for other players that could play the game, so a single-player game with computer-controlled allies could be approached by a designer either as single player versus game or as cooperative play.

Player Relationships and Roles Are Defined by Objectives

In addition to the interaction patterns listed in the preceding section, there are also various combinations of them, and in several games, one player might be another player's ally at one point and their competitor at another. For example, when trading money for property in a game like *Monopoly*, two players make a brief alliance with each other, even though the game is primarily multilateral competition.

At any time, the relationship between each player and both the game and other players is defined by the combination of all the players' layered objectives. These relationships lead each player to play one of several different roles:

• **Protagonist:** The protagonist role is that of the player trying to conquer the game.

• **Competitor:** The player trying to conquer other players. This is usually done solely to win the game, but in rare cases can be done on behalf of the game (e.g., in the 2004 board game *Betrayal at House on the Hill*, partway through the game, one of the players is turned evil and then must try to kill the other players).

• **Collaborator:** The player working to aid other players.

• **Citizen:** The player in the same world as other players but not really collaborating or competing with them.

In many multiplayer games, all players will play each of these roles at different times, and as you'll see when we look into the dynamic layer, there are different types of players who prefer different roles.

Rules

Rules limit the players' actions. Rules are also the most direct inscription of the designer's concept of how the game should be played. In the written rules of a board game, the designer is

attempting to inscribe and encode the experience that she wants for the players to have when they play the game. Later, the players decode these rules and hopefully experience something like what the designer intended.

Unlike paper games, digital games usually have very few inscribed rules that are read directly by the player; however, the programming code written by the game developers is another way of encoding rules that will be decoded through play. Because rules are the most direct method through which the game designer communicates with the player, rules act to define many of the other elements. The money in *Monopoly* only has value because the rules declare that it can be used to buy assets and other resources.

Explicitly written rules are the most obvious form of rules, but there are also implicit rules. For example, when playing poker, there is an implicit rule that you shouldn't hide cards up your sleeve. This is not explicitly stated in the rules of poker, but every player understands that doing so would be cheating.[2]

[2] This is a good example of one of the differences between single-player and multiplayer game design. In a multiplayer poker game, concealing a card would be cheating and could ruin the game. However, in the game *Red Dead Redemption* by Rockstar Studios, the in-game poker tournaments become much more interesting and entertaining once the player acquires the suit of clothes that allows her character to conceal and swap poker cards at will, with a risk of being discovered by NPCs (Non-Player Characters).

Boundaries

Boundaries define the edges of the space and time in which the game takes place. Within the boundaries, the rules and other aspects of the game apply: poker chips are worth something, it's okay to slam into other hockey players on the ice, and it matters which car crosses a line on the ground first. Sometimes, boundaries are physical, like the wall around a hockey rink. Other times,

boundaries are less obvious. When a player is playing an ARG (Alternate Reality Game), the game is often surrounding the player during her normal day. In one of the first ARGs, *Majestic* (a 2001 game by Electronic Arts), players of the game gave EA their phone number, fax number, email address, and home address, and they would receive phone calls, faxes, and so on at all times of the day from characters in the game. The intent of the game was to blur the boundaries between gaming and everyday life.

Resources

Resources are things of value in a game. These can be either things like assets or non-material attributes. Assets in games include things such as the equipment that Link has collected in a *Legend of Zelda* game; the resource cards that players earn in the board game *Settlers of Catan*; and the houses, hotels, and property deeds that players purchase in *Monopoly*. Attributes often include things such as health, the amount of air left when swimming under water, and experience points. Because money is so versatile and ubiquitous, it is somewhere between the two. A game can have physical money assets (like the cash in *Monopoly*), or it can have a nonphysical money attribute (like the amount of money that a player has in *Grand Theft Auto*).

Spaces

Designers are often tasked with creating navigable spaces. This includes both designing the board for a board game and designing virtual levels in a digital game. In both cases, you want to think about both flow through the space and making the areas of the space unique and interesting. Things to keep in mind when designing spaces include the following:

• **The purpose of the space:** Architect Christopher Alexander spent years researching why some spaces were particularly well suited to their use and why others weren’t. He distilled this knowledge into the

concept of *design patterns* through his book *A Pattern Language,*[3] which explored various patterns for good architectural spaces. The purpose of that book was to put forward a series of patterns that others could use to make a space that correctly matched the use for which it was intended.

[3]Christopher Alexander, Sara Ishikawa, and Murray Silverstein, *A Pattern Language: Towns, Buildings, Construction* (New York: Oxford University Press, 1977).

• **Flow:** Does your space allow the player to move through it easily, or if it does restrict movement, is there a good reason? In the board game *Clue*, players roll a single die each turn to determine how far they can move. This can make it very slow to move about the game board. (The board is 24x25 spaces, so with an average roll of 3.5, it could take 7 turns to cross the board.) Realizing this, the designers added secret passages that allow players to teleport from each corner of the board to the opposite corner, which helped flow through the mansion quite a bit.

• **Landmarks:** It is more difficult for players to create a mental map of 3D virtual spaces than actual spaces through which they have walked in real life. Because of this, it is important that you have landmarks in your virtual spaces that players can use to more easily orient themselves. In Honolulu, Hawaii, people don't give directions in terms of compass directions (north, south, east, and west) because these are not terribly obvious unless it's sunrise or sunset. Instead, the people of Honolulu navigate by obvious landmarks: *mauka* (the mountains to the northeast), *makai* (the ocean to the southwest), Diamond Head (the landmark mountain to the southeast), and Ewa (the area to the northwest). On other parts of the Hawaiian Islands, *mauka* means inland and *makai* means towards the ocean, regardless of compass direction (islands being circular). Making landmarks that players can easily see will limit the number of times your players need to consult the map to figure out where they are.

• **Experiences:** The game as a whole is an experience, but the map or space of the game also needs to be sprinkled with interesting experiences for your players. In *Assassin's Creed 4: Black Flag*, the world map is a vastly shrunken version of the Caribbean Sea. Even though the actual Caribbean has many miles of empty ocean between islands that would take a sailing vessel hours or days to cross, the Caribbean of *AC4* has events sprinkled throughout it that ensure that the player will encounter a chance to have an experience several times each minute. These could be small experiences like finding a single treasure chest on a tiny atoll, or they could be large experiences like coming across a fleet of enemy ships.

• **Short-, mid-, and long-term objectives:** As demonstrated in the screen shot from *Journey* shown in Figure 4.1, your space can have multiple levels of goals. In open-world games, a player is often shown a high-level enemy early on so that she has something to aspire to defeat later in the game. Many games also clearly mark areas of the map as easy, medium, or high difficulty.

Tables

Tables are a critical part of game balance, particularly when designing modern digital games. Put simply, tables are grids of data that are often synonymous with spreadsheets, but there are many different things that tables can be used to design and illustrate:

• **Probability:** Tables can be used to determine probability in very specific situations. In the board game *Tales of the Arabian Nights*, the player selects the proper table for the individual creature she has encountered, and it gives her a list of possible reactions that she can have to that encounter and the various results of each of her possible reactions.

• **Progression:** In paper roleplaying games (RPGs) like *Dungeons & Dragons*, tables show how a player's abilities increase and change as her player character's level increases.

• **Playtest Data:** In addition to tables that players use during the game, you as a designer will also create tables to hold playtest data and information about player experiences. You can find more info on this in Chapter 10, "Game Testing."

Of course, tables are also a form of technology in games, so they cross the line between mechanics and technology. Tables as technology include the storage of information and any transformation of information that can happen in the table (e.g., formulae in spreadsheets). Tables as mechanics include the design decisions that game designers make and inscribe into the table.

Inscribed Aesthetics

Inscribed aesthetics are those aesthetic elements that are crafted by the developers of the game. These cover all the five senses, and as a designer, you should be aware that throughout the time that your player is playing the game, she will be sensing with all five of her senses.

The Five Aesthetic Senses

Designers must consider all five of the human senses when inscribing games. These five senses are:

• **Vision:** Of the five senses, vision is the one that gets the most attention from most game development teams. As a result, the fidelity of the visual experience that we can deliver to players has seen more obvious improvement over the past decades than that of any other sense. When thinking about the visible elements of your game, be sure to think beyond the 3D art in the game or the art of the board in a paper game. Realize that everything that players (or potential players) see that has anything to do with your game will affect their impression of it as well as their enjoyment of it. Some game developers in the past have put tremendous time into making

their in-game art beautiful only to have the game packaged in (and hidden behind) awful box art.

• **Hearing:** Audio in games is second only to video in the amazing level of fidelity that can be delivered to players. All modern consoles can output 5.1-channel sound, and some can do even better than that. Game audio is composed of: sound effects, music, and dialogue. Each will take a different amount of time to be interpreted by the player, and each has a different best use. In addition, on a medium to large development team, each of these three will usually be handled by a different artist.



Another aspect of audio to consider is background noise. For mobile games, you can almost always expect that the player is going to be in a non-optimal audio situation when playing your game. Though audio can always add to a game, it's just not prudent to make it a vital aspect of a mobile game unless it's the core feature of the game (e.g., games like *Papa Sangre* by Somethin' Else or *Freeq* by Psychic Bunny). There is also background noise in computer and console games to consider. Some cooling fans for some are very loud, and that needs to be considered when developing quiet audio for digital games.

• **Touch:** Touch is very different between board games and digital games, but in both cases, it's the most direct contact that you have with the player. In a board game, touch comes down to the feel of the playing pieces, cards, board, and so on. Do the pieces for your game feel high quality or do they feel cheap? Often you want them to be the former, but the latter isn't terrible. James Ernst, possibly the most prolific board game designer in the world for several years, ran a company called Cheap Ass Games, the mission of which was to get great games to players at as low a cost to them as possible. In order to cut costs, playing pieces were made of cheap materials, but this was fine with players because the games from his company cost less than $10 each instead of the $40–50 that many board games

cost. All design decisions are choices; just make sure that you're aware of the options and know that you're making a choice.

One of the most exciting recent technological advancements for board game prototyping is 3D printing, and many board game designers are starting to print pieces for their game prototypes. There are also several companies online now that will print your game board, cards, or pieces.

There are also aspects of touch in digital games. The way that the controller feels in a player's hands and the amount of fatigue that it causes are definitely aspects that a designer needs to consider. When the fantastic PlayStation 2 game *Okami* was ported to the Nintendo Wii, the designers chose to change the attack command from a button press (the X on the PlayStation controller) to a waggle of the Wiimote (which mimicked the attack gesture from *The Legend of Zelda: Twilight Princess* that had done very well on the Wii). However, while attacks in the heat of battle in *Twilight Princess* happen about once every couple of seconds, attacks in *Okami* happen several times per second, so the attack gesture that worked well in *Twilight Princess* instead caused player fatigue in *Okami*. With the rise of tablet and smartphone gaming, touch and gesture are elements that every digital game designer must consider carefully.

Another aspect of touch in digital games is rumble-style player feedback. As a designer, you can choose the intensity and style of rumble feedback in most modern console controllers.

• **Smell:** Smell is not often a designed aspect of inscribed aesthetics, but it is there. Just as different book printing processes have different smells, so too do different board and card game printing processes. Make sure that you get a sample from your manufacturer before committing to printing 1,000 copies of something that might smell strange.

Aesthetic Goals

Humankind has been making art and music since long before the dawn of written history. Therefore, when designing and developing the inscribed aesthetic elements of a game, we as game developers are taking advantage of hundreds of years of cultural understanding of other forms of art. Interactive experiences have the advantage of being able to pull from all of that experience and of allowing us as designers to incorporate all of the techniques and knowledge of aesthetic art into the games that we create. However, when we do so, it must be done with a reason, and it must mesh cohesively with the other elements of the game. These are some of the goals that aesthetic elements can serve well in our games:

• **Mood:** Aesthetics do a fantastic job of helping to set the emotional mood of a game. While mood can definitely be conveyed through game mechanics, both visual art and music can do a fantastic job of influencing a player's mood much faster than mechanics are able to.

• **Information:** Several informational colors are built in to our psyche as mammals. Things like the color red or alternating yellow and black are seen by nearly every species in the animal kingdom as indicators of danger. In contrast, cool colors like blue and green are usually seen as peaceful.

In addition, players can be trained to understand various aesthetics as having specific meaning. The LucasArts game *X-Wing* was the first to have a soundtrack that was procedurally generated by the in-game situation. The music would rise in intensity to warn the player that enemies were attacking. Similarly, as described in Chapter 13, "Guiding the Player," the colors bright blue and yellow are used throughout the Naughty Dog game *Uncharted 3* to help the player identify handholds and footholds for climbing.

Inscribed Narrative

As with all forms of experience, dramatics and narrative are an important part of many interactive experiences. However, game narratives face challenges that are not present in any form of linear

media, and as such, writers are still learning how to craft and present interactive narratives. This section will explore the components of inscribed dramatics, purposes for which they are used, methods for storytelling in games, and differences between game narrative and linear narrative.

Components of Inscribed Narrative

In both linear and interactive narrative, the components of the dramatics are the same: premise, setting, character, and plot.

• **Premise:** The premise is the narrative basis from which the story emerges:[4]

[4] These are the premises of *Star Wars: A New Hope*, *Half-Life*, and *Assassin's Creed 4: Black Flag*.

A long time ago in a galaxy far, far away, an intergalactic war is brought to the doorstep of a young farmer who doesn't yet realize the importance of his ancestry or himself.

Gordon Freeman has no idea about the surprises that are in store for him on his first day of work at the top secret Black Mesa research facility.

Edward Kenway must fight and pirate his way to fortune on the high seas of the Caribbean while discovering the secret of the mysterious Observatory, sought by Templars and Assassins alike.

• **Setting:** The setting expands upon the skeleton of the premise to provide a detailed world in which the narrative can take place. The setting can be something as large as a galaxy far, far away or as small as a tiny room beneath the stairs, but it's important that it is believable within the bounds of the premise and that it is internally consistent; if your characters will choose to fight with swords in a world full of guns, you need to have a good reason for it.

In *Star Wars*, when Obi Wan Kenobi gives the lightsaber to Luke, he answers all of these questions by stating that it is “not as clumsy or random as a blaster; an elegant weapon for a more civilized age.”

• **Character:** Stories are about characters, and the best stories are about characters we care about. Narratively, characters are composed of a backstory and one or more objectives. These combine to give the character a role in the story: protagonist, antagonist, companion, lackey, mentor, and so on.

• **Plot:** Plot is the sequence of events that take place in your narrative. Usually, this takes the form of the protagonist wanting something but having difficulty achieving it because of either an antagonist or an antagonistic situation getting in the way. The plot then becomes the story of how the protagonist attempts to overcome this difficulty or obstruction.

Traditional Dramatics

Though interactive narrative offers many new possibilities to writers and developers, it still generally follows traditional dramatic structures.

Five-Act Structure

German writer Gustav Freytag wrote about five-act structure in his 1863 book *Die Technik des Dramas* (The Technique of Dramas). It described the purpose of the five acts often used by Shakespeare and many of his contemporaries (as well as Roman playwrights) and proposed what has come to be known as Freytag’s pyramid (see Figure 4.2). The vertical axes in Figures 4.2 and 4.3 represent the level of audience excitement at that point in the story.

Figure 4.2. Freytag’s pyramid of five-act structure showing examples from *Romeo and Juliet* by William Shakespeare



According to Freytag, the acts work as follows:

• **Act I: Exposition:** Introduces the narrative premise, the setting, and the important characters. In Act I of William Shakespeare's *Romeo and Juliet*, we are introduced to Verona, Italy, and the feud between the powerful Montague and Capulet families. Romeo is introduced as the son of the Montague family and is infatuated with Rosaline.

• **Act II: Rising action:** Something happens that causes new tension for the important characters, and the dramatic tension rises. Romeo sneaks into the Capulet's ball and is instantly smitten with Juliet, the daughter of the Capulet family.

• **Act III: Climax:** Everything comes to a head, and the outcome of the play is decided. Romeo and Juliet are secretly married, and the local friar hopes that this may lead to peace between their families. However, the next morning, Romeo is accosted by Juliet's cousin Tybalt. Romeo refuses to fight, so his friend Mercutio fights in his stead, and Tybalt accidentally kills Mercutio (because Romeo got in the way). Romeo is furious and chases Tybalt, eventually killing him. Romeo's decision to kill Tybalt is the moment of climax of the play because before that moment, it seems like everything might work out for the two lovers, and after that moment, the audience knows that things will end horribly.

• **Act IV: Falling action:** The play continues toward its inevitable conclusion. If it's a comedy, things get better; if it's a tragedy, they may appear to be getting better, but it just gets worse. The results of the climax are played out for the audience. Romeo is banished from Verona. The friar concocts a plan to allow Romeo and Juliet to escape together. He has Juliet fake her death and sends a message to Romeo to let him know, but the messenger never makes it to Romeo.

• **Act V: Denouement (pronounced "day-new-maw"):** The play resolves. Romeo enters the tomb believing Juliet to be truly dead and kills himself. She immediately awakens to find him dead and

then kills herself as well. The families become aware of this tragedy, and everyone weeps, promising to cease the feud.

Three-Act Structure

In his books and lectures, American screenwriter Syd Field has proposed another way of understanding traditional narrative in terms of three acts.[5] Between each act, a plot point changes the direction of the story and forces the characters actions. Figure 4.3 provides an example that is further explained in the following list.

[5] Syd Field, *Screenplay: The Foundations of Screenwriting* (New York: Delta Trade Paperbacks, 2005).

Figure 4.3. Syd Field's three-act structure, with examples from *Star Wars: A New Hope*



The core elements of Field's three-act structure are:

• **Act I: Exposition:** Introduces the audience to the world of the narrative and presents the premise, setting, and main characters. In Act I of *Star Wars*, Luke is a young, idealistic kid who works on his uncle's moisture farm. There is a galactic rebellion happening against a fascist Empire, but he's just a simple farmer dreaming of flying starfighters.

• **Hook:** Gets the audience's attention quickly. According to Field, an audience decides in the first few minutes whether they're going to watch a film, so the first few minutes should be really exciting, even if the action in them has nothing to do with the rest of the film (e.g., the beginning of any James Bond film). In *Star Wars,* the opening scene of Princess Leia's ship being attacked by a Star Destroyer had some of the best special visual effects that 1977 audiences had ever seen and a fantastic score by John Williams, both of which helped make it an exciting hook.

• **Inciting Incident:** Something new enters the life of the main character, causing her to start the adventure. Luke is leading a pretty normal life until he finds Leia's secret message stored inside of R2-D2. This discovery causes him to seek out "Old Ben" Kenobi, who changes his life.

• **First Plot Point:** The first plot point ends the first act and pushes the protagonist down the path toward the second. Luke has decided to stay home and not help Obi-Wan Kenobi, but when he finds that the Empire has killed his aunt and uncle, he changes his mind and decides to join Obi-Wan and train to become a Jedi.

• **Act II: Antagonism:** The protagonist starts her journey, but a series of obstacles get in her way. Luke and Obi-Wan hire Han Solo and Chewbacca to help them deliver the secret plans carried by R2-D2 to Alderaan, however when they arrive, Alderaan has been destroyed, and their ship is captured by the Death Star.

• **Second Plot Point:** The second plot point ends the second act and pushes the protagonist into her decision of what she will attempt in the third act. After much struggle, Luke and his friends escape from the Death Star with both the princess and the plans, but his mentor, Obi-Wan Kenobi, is killed in the process. The Death Star follows them to the rebel's secret base, and Luke must choose whether to aid in the attack on the Death Star or to leave with Han Solo.

• **Act III: Resolution:** The story is concluded, and the protagonist either succeeds or fails. Either way, she emerges from the story with a new understanding of who she is. Luke chooses to help attack the Death Star and ends up saving the day.

• **Climax:** The moment when everything comes to a head and the main question of the plot is answered. Luke is alone in the Death Star trench having lost both his wingmen and R2-D2. Just as he is about to be shot down by Darth Vader, Han and Chewbacca appear to save him, allowing him a clean shot. Luke chooses to trust the Force over technology and shoots with his eyes closed, making an extremely difficult shot and destroying the Death Star.

In most modern movies and in nearly all video games, the climax is very close to the end of the narrative with almost no time for falling action or denouement. One marked example of this not being the case is the game *Red Dead Redemption* by Rockstar Games. After the big climax where the main character, John Marston, finally defeats the man the government hired him to kill, he is allowed to go home to his family, with the game playing its only sung musical track as John rides home slowly in the snow. Then, the player is subjected to a series of rather dull missions where John clears crows out of the family grain silo, teaches his petulant son to wrangle cattle, and does other chores around the house. The player feels the boredom of these missions much like John does. Then, the same government agents that initially hired John come to his farm to kill him, eventually succeeding in their task. Once John dies, the game fades to black and fades back in on the player in the role of Jack (John's son) three years after his father's death. The game returns to more action-based missions as Jack attempts to track down the agents who killed his father. This kind of falling action is rare and refreshing to see in games, and it made the narrative of *Red Dead Redemption* one of the most memorable that I've played.

Differences Between Interactive and Linear Narrative

At their core, interactive and linear narratives are quite different because of the difference in the role of the audience versus the player. Though an audience member of course brings her own background and interpretations to any media that she consumes, she is still unable to change the actual media itself, only her perception thereof. However, a player is constantly affecting the media in which she is taking part, and therefore a player has actual agency in the interactive narratives that she experiences. This means that authors of interactive narrative must be aware of some core differences in how they can craft their narratives.

Plot Versus Free Will

One of the most difficult things to give up when crafting interactive narratives is control over the plot. Both authors and readers/viewers are accustomed to plots with elements like foreshadowing, fate, irony, and other ways in which the intended outcome of the plot actually influences earlier parts of the story. In a truly interactive experience, this would be impossible because of the free will of the player. Without knowing what choices the player will make, it is very difficult to intentionally foreshadow the results of those choices. However, there are several possibilities for dealing with this dichotomy, some of which are already used often in digital games and others of which are used in situations like pen-and-paper RPGs but have not yet been implemented in many digital games:

• **Limited possibilities:** Limited possibilities are an important part of nearly all interactive narrative experiences. In fact, most games, at their inscribed level, are not actually interactive narratives. All the most popular series of games over the past decade (*Prince of Persia*, *Call of Duty*, *Halo*, *Uncharted*, and so on) have exclusively linear stories at their core. No matter what you do in the game, your choices are to either continue with the narrative or quit the game. *Spec Ops: The Line* by Yager Development explored this issue beautifully, placing the player and the main character of the story in the same position of having only two real choices: continue to perform increasingly horrific acts or just stop playing the game. In *Prince of Persia: The Sands of Time*, this is handled by having the narrator (the prince of the title and the protagonist) say “No, no, no; that’s not the way it happened. Shall I start again?” whenever the player dies and the game has to back up to the most recent check point. In the *Assassin’s Creed* series, this is handled by stating that you have become “desynchronized” from your ancestor’s story if (through lack of player skill) the ancestor is allowed to die.

There are also several examples of games that limit choices to only a few possibilities and base those on the player’s actions throughout the game. Both *Fable*, by Lionhead Studios, and *Star Wars: Knights of the Old Republic*, by Bioware, claimed to be watching the player throughout the game to determine the final game outcome, but

though each did track the player on a good versus evil scale throughout the game, in both cases (as in many other games), a single choice made at the end of the game could override an entire game of good or evil behavior.

Other games like the Japanese RPGs *Final Fantasy VII* and *Chrono Trigger* have more subtle and varied possibilities. In *Final Fantasy VII*, there is a point where the main character, Cloud, goes on a date with someone at the Golden Saucer amusement park. The default is for Cloud to go out with Aeris; however, if the player has ignored Aeris throughout the game and kept her out of their battle party, Cloud will instead go out with Tifa. The possibilities for the date are the characters Aeris, Tifa, Yuffie, and Barrett, although it takes resolute effort to have the date with Barrett. The game never explains that this math is happening in the background but it is always there, and the Final Fantasy team used a similar strategy in *Final Fantasy X* to determine who the protagonist, Tidus, would ride on a snowmobile with in a romantic scene. *Chrono Trigger* uses several metrics to determine which of the game's thirteen endings to choose (and some of those endings have multiple possibilities within them). Again, the calculations for this are largely invisible to the player.

• **Allow the player to choose from several linear side quests:** Many of Bethesda Softwork's open-world games use this strategy, including the recent games *Fallout 3* and *Skyrim*. While the main quest is generally pretty linear for these games, it is only a small fraction of the game's total content. In *Skyrim*, for instance, the main quest takes about 12 to 16 hours to complete, but the game has more than 400 hours of additional side quests. A player's reputation and history in the game lead to some side quests being unlocked and exclude her from playing others. This means that each individual who plays the game has the potential to have a different combination of linear experiences that add up to a different overall game experience from other players.

• **Foreshadowing multiple things:** If you foreshadow several different things that might happen, some of them probably will happen. Players will generally ignore the foreshadowing that is not paid off while noticing that which does. This happens often in serial television shows where several possibilities for future plots are put in place but only a few are ever actually executed (e.g., the Nebari plot to take over the universe that is revealed in the *Farscape* episode "A Clockwork Nebari" but is never mentioned again and the titular character from the *Doctor Who* episode "The Doctor's Daughter" who never returns to the show).

• **Develop minor nonplayer characters (NPCs) into major ones:** This is a tactic used often by game masters of pen-and-paper RPGs. An example of this would be if the players were attacked by a group of ten bandits, and the players defeated the bandits, but one got away. The game master (GM) could then choose to have that bandit return at some point with a vendetta against the players for killing his friends. This differs significantly from games like *Final Fantasy VI* (originally titled *Final Fantasy III* in the U.S.), where it is pretty obvious from early in the game that Kefka will be a recurring, annoying, and eventually wholly evil nemesis character. Though the characters in the player's party don't realize this, just the fact that the developers chose to give Kefka a special sound effect for his laugh makes it apparent to the player.

Tip

Pen-and-paper RPGs still offer players a unique interactive gaming experience, and I highly recommend them. In fact, when I taught at the University of Southern California, I required all of my students to run an RPG and play in a couple run by their peers. Roughly 40% of the students each semester listed it as their favorite assignment.

Because pen-and-paper RPGs are run by a person, that game master (GM) can craft the narrative in real time for the players in a way that computers have yet to match. All of the strategies listed earlier are used by GMs to guide their players and make their

experiences seem fated, foreshadowed, or ironic in ways that are usually reserved for linear narrative.

The perennial RPG *Dungeons & Dragons*, by Wizards of the Coast, is a good place to get started, and there are a tremendous number of source books for it. However, I have found that *D&D* campaigns tend to be rather combat-focused, and the combat can take a very long time. For an experience that allows you to most easily create and experience interactive stories, I recommended the *FATE* system by Evil Hat Productions.

Empathetic Character Versus Avatar

In linear narratives, the protagonist is often a character with whom the audience is expected to empathize. When the audience watches Romeo and Juliet make stupid decisions, they remember being young themselves and empathize with the feelings that lead the two lovers down their fatal path. In contrast, the protagonist in an interactive narrative is not a character separate from the player but instead the player's *avatar* in the world. (Avatar is a word from Sanskrit that refers to the physical embodiment of a god on Earth; in games, it is the virtual embodiment of the player in the game world.) This can lead to a dissonance between the actions and personality that the player would like to have in the world and the personality of the player-character (PC). For me, this was driven home by my experience with Cloud Strife as the protagonist of *Final Fantasy VII*. Throughout the game, Cloud was a little more petulant than I would have liked, but in general, his silence allowed me to project my own character on to him. However, after a pivotal scene where Cloud loses someone close to him, he chose to sit, unresponsive in a wheelchair instead of fighting to save the world from Sephiroth, as I wanted to. This dichotomy between the PC's choice and the choice that I as the player wanted to make was extremely frustrating for me.

A fantastic example of this dichotomy being used to great effect happens in the fantastic Clover Studio game *Okami*. In *Okami*, the

player character is Amaterasu, a reincarnation of the female god of the sun in the form of a white wolf. However, Amaterasu's powers have diminished over the past 100 years, and the player must work to reclaim them. About a quarter of the way through the narrative, the main antagonist, the demon Orochi, chooses a maiden to be sacrificed to him. Both the player and Amaterasu's companion, Issun, know that Amaterasu has only regained a few of her powers at this point, and the player feels wary of facing Orochi in such a weakened state. However, despite Issun's protests, Amaterasu runs directly to the fight. As the music swells in support of her decision, my feelings as a player changed from trepidation to temerity, and I, as the player, actually felt like a hero because I knew that the odds were against me, but I was still doing what needed to be done.

This character versus avatar dichotomy has been approached several ways in games and interactive narrative:

• **Role fulfillment:** By far, the most common approach in games is to have the player roleplay the game character. When playing character-driven games like the *Tomb Raider* or *Uncharted* series, the player is playing not themselves but instead Lara Croft or Nathan Drake. The player sets aside her own personality to fulfill the inscribed personality of the game's protagonist.

• **The silent protagonist:** In a tradition reaching at least as far back as the first *Legend of Zelda* game, many protagonists are largely silent. Other characters talk to them and react as if they've said things, but the player never sees the statements made by the player character. This was done with the idea that the player could then impress her own personality on the protagonist rather than being forced into a personality inscribed by the game developers. However, regardless of what Link says or doesn't say, his personality is demonstrated rather clearly by his actions, and even without Cloud saying a word, players can still experience a dissonance between their wishes and his actions as described in the preceding example.

• **Multiple dialogue choices:** Many games offer the player multiple dialogue choices for her character, which can certainly help the

player to feel more in control of the character and her personality. However, there are a couple of important requirements:

• The player must understand the implications of her statement: Sometimes, a line that may seem entirely clear to the game's writers does not seem to have the same connotations to the player. If the player chooses dialogue that seems to her to be complimentary, but the writer meant for it to be antagonistic, the NPC's reaction can seem very strange to the player.

• The choice of statement must matter: Some games offer the player a fake choice, anticipating that she will make the choice that the game desires. If, for instance, she's asked to save the world, and she just says No, it will respond with something like "oh, you can't mean that," and not actually allow her a satisfactory choice.

One fantastic example of this being done well is the dialog wheel in the *Mass Effect* series by Bioware. In these games, the player is presented with a wheel of dialog choices, and the sections of the wheel are coded with meaning. A choice on the left side of the wheel will extend the conversation, while one on the right side will shorten it. A choice on the top of the wheel will be friendly, while one on the bottom will be surly or antagonistic. By positioning the dialog options in this way, the player is granted important information about the connotations of her possible choices and is not surprised by the outcome.

Another very different but equally compelling example is *Blade Runner* by Westwood Studios (1997). The designers felt that choosing dialog options would interrupt the flow of the player experience, so instead of offering the player a choice between dialogue options at every statement, the player was able to choose a mood for her character (friendly, neutral, surly, or random). The protagonist would act and speak as dictated by his mood without any interruption in the narrative flow, and the player could change the mood at any time to alter her character's response to the situation.

• **Track player actions and react accordingly:** Some games now track the player's relationships with various factions and have the faction members react to the player accordingly. Do a favor for the Orcs, and they may let you sell goods at their trading post. Arrest a member of the Thieves Guild, and you may find yourself mugged by them in the future. This is a common feature of open world western roleplaying games like those by Bethesda Softworks and is in some ways based on the morality system of eight virtues and three principles that was introduced in *Ultima IV*, by Origin Systems, one of the first examples of complex morality systems in a digital game.

Purposes for Inscribed Dramatics

Inscribed dramatics can serve several purposes in game design:

• **Evoking emotion:** Over the past several centuries, writers have gained skill in manipulating the emotions of their audiences through dramatics. This holds true in games and interactive narrative as well, and even purely linear narrative inscribed over a game can focus and shape the player's feelings.

• **Motivation and justification:** Just as dramatics can shape emotions, they can also be used to encourage the player to take certain actions or to justify actions if those actions seem distasteful. This is very true of the fantastic retelling of Joseph Conrad's *Heart of Darkness* in the game *Spec Ops: The Line*. A more positive example comes from *The Legend of Zelda: The Wind Waker*. At the beginning of the game, Link's sister Aryll lets him borrow her telescope for one day because it's his birthday. On the same day, she is kidnapped by a giant bird, and the first part of the game is driven narratively by Link's desire to rescue her. The inscribed storytelling of her giving something to the player before being kidnapped increases the player's personal desire to rescue her.

• **Progression and reward:** Many games use cut scenes and other inscribed narrative to help the player know where she is in the story and to reward her for progression. If the narrative of a game is

largely linear, the player's understanding of traditional narrative structure can help her to understand where in the three-act structure the game narrative currently is, and thereby, she can tell how far she has progressed in the overall plot of the game. Narrative cut scenes are also often used as rewards for players to mark the end of a level or other section of the game. This is true in the single-player modes of nearly all top-selling games with linear narratives (e.g., the *Modern Warefare*, *Halo*, and *Uncharted* series).

• **Mechanics reinforcement:** One of the most critical purposes of inscribed dramatics is the reinforcement of game mechanics. The German board game *Up the River* by Ravensburger is a fantastic example of this. In the game, players are trying to move their three boats up a board that is constantly moving backward. Calling the board a "river" reinforces the backward movement game mechanic. A board space that stops forward progress is called a "sandbar" (as boats often get hung up on sandbars). Similarly, the space that pushes the player forward is called a "high tide." Because each of these elements has dramatics associated with it, it is much easier to remember than, for instance, if the player were asked to remember that space #3 stopped the boat and #7 moved the boat forward.

Inscribed Technology

Much like inscribed mechanics, inscribed technology is largely understood only through its dynamic behavior. This is true whether considering paper or digital technology. The choice of how many dice of how many sides each to be thrown by the player only really matters when those dice are in play just as the code written by a programmer is only really understood by the player when she sees the game in action. This is one of the reasons that, as was pointed out in Schell's Elemental Tetrad in Chapter 2, "Game Analysis Frameworks," technology is the least visible of the inscribed elements.

In addition, a large overlap exists between inscribed mechanics and inscribed technology. Technology enables mechanics, and

mechanical design decisions can lead to a choice of which technology to use.

Inscribed Paper Game Technology

Inscribed technologies in paper games are often used for randomization, state tracking, and progression:

• **Randomization:** Randomization is the most common form of technology in paper games. This ranges from dice, to cards, to dominoes, to spinners, and so on. As a designer, you have a lot of control over which of these you choose and how the randomization works. Randomization can also be combined with tables to do things like generate random encounters or characters for a game. In Chapter 11, "Math and Game Balance," you can read about the various types of randomizers and when you might want to use them.

• **State tracking:** State tracking can be everything from keeping track of the scores of the different players of the game (like a cribbage board) or tables like the complex character sheets used in some roleplaying games.

• **Progression:** Progression is often inscribed via charts and tables. This includes things such as player progression of abilities when the player levels up, the progression of various technologies and units in the technology tree of a game like *Civilization*, progression of resource renewal in the board game *Power Grid*, and so on.

Inscribed Digital Game Technology

The latter sections of this book extensively cover digital game technology in the form of programming games using Unity and the C# programming language. Just as with inscribed paper game technology, the art of game programming is that of encoding the experience you want the player to have into inscribed rules (in the form of programming code) that will then be decoded by the player as she plays the game.

Summary

The four elements of the inscribed layer make up everything that players receive when they purchase or download your game, and therefore the inscribed layer is the only one over which the game developers have complete control. In the next chapter, we allow players to move our games from the static form of the inscribed layer up to the emergence of the dynamic layer.

Chapter 5. The Dynamic Layer

Once players start actually playing a game, it moves from the inscribed layer into the dynamic layer of the Layered Tetrad. Play, strategy, and meaningful player choices all emerge in this layer.

This chapter explores the dynamic layer, various qualities of emergence, and how designers can anticipate the dynamic play that emerges from their inscribed design decisions.

The Role of the Player

A fellow designer once told me that a game isn't a game unless someone is playing it. Although this might sound initially like a rehash of "if a tree falls in the woods, and there's no one to hear it, does it make a sound?" it's actually much more important for interactive media than any other medium. A film can still exist and show in a theater if there's no one to watch it.[1] Television can be sent out over the airwaves and still be television, even if no one is tuned to that station. Games, however, just don't exist without players, for it is through the actions of players that games transform from a collection of inscribed elements into an experience (see Figure 5.1).

[1] Some films, like the *Rocky Horror Picture Show,* owe a lot of their cult fandom to presentations in which the audience takes part, and the audience reactions in those films do alter the viewing experience of the other audience members. However, the film itself is completely unaffected by the audience. The dynamism in games comes from the ability of the medium to react to the player.

Figure 5.1. Players move the game from the inscribed layer into the dynamic layer



There are, of course, some edge cases to this, as there are to all things. The game *Core War* is a hacking game where players each try to write a computer virus that will propagate and take over a fake computer core from the viruses of their competitors. Players submit their viruses and wait for them to fight each other for memory and survival. In the yearly *RoboCup* tournament, various teams of robots compete against each other in soccer without any interference by the programmers during the game. In the classic card game *War*, players make no decision beyond the choice of which of the two decks to take at the beginning of the game, and the game plays out entirely based on the luck of the initial shuffle. Though in each of these cases, the player has no input and makes no choices during the actual play of the game, the play is still influenced by player decisions made before the official start of the match, and the players certainly have interest in and are waiting for the outcome of the game. In all of these cases, it still takes players to set up the game and to make the choices that determine its outcome.

Though players have a tremendous effect on the game and gameplay (including influences on the tetrad elements), players sit outside of the tetrad as the engine that makes it work. Players cause games to come into being and allow them to become the experience that has been encoded into the inscribed layer of the game by the game developers. As designers, we rely on players to aid us in helping the game to become what we intend. There are several aspects that are completely beyond our control as designers, including: whether the player is actually trying to follow the rules, whether the player cares about winning or not, the physical environment in which the game is played, the emotional state of the players, and so on. Because players are so important, we as developers need to treat them with respect and take care to ensure that the inscribed elements of the game, like rules, are clear enough to the players that they can decode them into the game experience that we intend.

Emergence

The most important concept in this chapter is emergence, the core of which is that even very simple rules can beget complex dynamic behaviors. Consider the game of *Bartok* that you played and experimented with in Chapter 1, "Thinking Like a Designer." Though *Bartok* had very few rules, complex play emerged from them. And, once you started changing rules and adding your own, you were able to see that even simple, seemingly innocuous rule changes had the potential to lead to large changes in both the feel and the play of the game.

The dynamic layer of the Layered Tetrad encompasses the results of the intersection of player and game across all four elements of the tetrad: mechanics, aesthetics, dramatics, and technology.

Unexpected Mechanical Emergence

My friend Scott Rogers, author of two books on game design,[2] once told me that he "didn't believe in emergence." After discussing it with him for a while, we came to the conclusion that he did believe in emergence, but he didn't believe that it was legitimate for game designers to use emergence as an excuse for irresponsible design. Scott felt, and I believe, that as the designer of the systems within a game, you are responsible for the play that emerges from those systems. Of course, it's extremely difficult to know what possibilities will emerge from the rules that you put in place, which is why playtesting is so critically important. As you develop your games, playtest early, playtest often, and take special care to note unusual things that happen in only one playtest. Once your game gets out in the wild, the sheer number of people playing will cause those unusual flukes to happen a lot more often than you would expect. Of course, this happens to all designers—look at some of the cards that have been declared illegal in *Magic: The Gathering*—but as Scott says, it's important that designers own these issues and take care to resolve them.

[2] Scott Rogers, *Level up!: The Guide to Great Video Game Design* (Chichester, UK: Wiley, 2010) and
Scott Rogers, *Swipe this! The Guide to Great Tablet Game Design* (Hoboken, NJ: John Wiley & Sons, 2012).

Dynamic Mechanics

Dynamic mechanics are the dynamic layer of the elements that separate games and interactive media from other media; the elements that make them games. Dynamic mechanics include: procedures, meaningful play, strategy, house rules, player intent, and outcome. As with the inscribed mechanics, many of these are an expansion of elements described in Tracy Fullerton's book *Game Design Workshop.*[3]

[3] Tracy Fullerton, Christopher Swain, and Steven Hoffman, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games* (Burlington, MA: Morgan Kaufmann Publishers, 2008), chapters 3 and 5.

Procedures

Mechanics in the inscribed layer included *rules*: instructions from the designer to the players about how to play the game. *Procedures* are the dynamic actions taken by the players in response to those rules. Another way to say this is that procedures emerge from rules. In the game *Bartok*, if you added the rule about a player needing to announce when they had only one card left, there was an explicit procedure presented in the rules that the active player needed to do so (once she had only one card left). However, there was also an implicit procedure in that rule: the procedure of other players watching the hand of the active player so that they could catch her if she forgot to announce it. Prior to this rule, there was no real reason for a player to pay attention to the game during another person's turn, but this simple rule change altered the procedures of playing the game.

Meaningful Play

In *Rules of Play*, Katie Salen and Eric Zimmerman define meaningful play as play that is both *discernable* to the player and *integrated* into the larger game.[4]

[4] Katie Salen and Eric Zimmerman, *Rules of Play: Game Design Fundamentals* (Cambridge, MA: MIT Press, 2003), 34.

• **Discernable:** An action is discernable to the player if the player can tell that the action has been taken. For example, when you press the call button for an elevator, the action is discernable because the call button lights up. If you've ever tried to call an elevator when the light inside the button was burned out, you know how frustrating it can be to take an action and yet not be able to discern whether the game interpreted your action.

• **Integrated:** An action is integrated if the player can tell that it is tied to the outcome of the game. For example, when you press the call button for the elevator, that action is integrated because you know that doing so will cause the elevator to stop on your floor. In *Super Mario Bros.*, the decision of whether to stomp an individual enemy or just avoid it is generally not very meaningful because that individual action is not integrated into the overall outcome of the game. *Super Mario Bros.* never gives you a tally of the number of enemies defeated; it only requires that you finish each level before the time runs out and finish the game without running out of lives. In HAL Laboratories series of *Kirby* games, however, the player character Kirby gains special abilities by defeating enemies, so the decision of which enemy to defeat is directly integrated into the acquisition of abilities, and the decision is made more meaningful.

If a player's actions in the game are not meaningful, she can quickly lose interest. Salen and Zimmerman's concept of meaningful play reminds designers to constantly think about the mindset of the player and whether the interactions of their games are transparent or opaque from the player's perspective.

Strategy

When a game allows meaningful actions, players will usually create strategies to try to win the game. A strategy is a calculated set of actions to help the player achieve a goal. However, that goal can be anything of the player's choosing and does not necessarily need to be the goal of winning the game. For instance, when playing with a young child or with someone of a lower skill level in a game, the player's goal might be to make sure that the other person enjoys playing the game and learns something, sometimes at the expense of the player winning the game.

Optimal Strategy

When a game is very simple and has few possible actions, it is possible for players to develop an *optimal strategy* for the game. If both players of a game are playing rationally with the goal of winning, an optimal strategy is the possible strategy with the highest likelihood of winning. Most games are too complex to really have an optimal strategy, but some games like Tic-Tac-Toe are simple enough to allow one. In fact, Tic-Tac-Toe is so simple that chickens have (possibly) been trained to play it and force a draw or a win almost every time.[5]

[5] Kia Gregory, "Chinatown Fair Is Back, Without Chickens Playing Tick-Tack-Toe," *New York Times*, June 10, 2012.

An optimal strategy is more often a fuzzy idea of the kind of thing that would likely improve a player's chance of winning. For instance, in the board game *Up the River* by Manfred Ludwig, players are trying to move three boats up a river to dock at the top of the game board, and arriving at the dock is worth 12 points to the first boat to arrive, then 11 points for the second boat, and down to only 1 point for the twelfth boat. Every round (that is, every time that all players have taken one turn), the river moves backward 1 space, and any boat that falls off the end of the river (the waterfall) is lost. Each turn, the player rolls 1d6 (a single six-sided die) and chooses which boat

to move. Because the average roll of a six-sided die is 3.5, and the player must choose from among her three boats to move every turn, each boat will move an average of 3.5 spaces every three of her turns. However, the board will move backward 3 spaces every three turns, so each boat only makes an average forward progression of 0.5 spaces every three turns (or 0.1666 (or 1/6) spaces every turn).[6]

[6] There are additional rules of the game that I'm omitting for the sake of simplicity in this example.

In this game, the optimal strategy is for the player to never move one of her boats and just let it fall off the waterfall. Then each boat would move forward an average of 3.5 spaces every two turns instead of three. With the board moving backward 2 spaces in two turns, this would give each of her boats an average movement forward of 1.5 spaces every two turns (or 0.75 spaces each turn), which is much better than the 0.1666 afforded to the player if she tries to keep all of her boats. Then this player would have a better chance of getting to the dock in first and second place, giving her 23 total points (12 + 11). In a two-player game, this strategy wouldn't work because the second player would tie up at 10, 9, and 8 for 27 points, but in a three- or four-player game, it's the closest thing to an optimal strategy for *Up the River*. However, the other players' choices, randomized outcomes of the dice, and additional factors mean that it won't always ensure a win; it will just make a win more likely.

Designing for Strategy

As a designer, you can do several things to make strategy more important in your game. For now, the main thing to keep in mind is that presenting the player with multiple possible ways to win will require her to make more difficult strategic decisions during play. In addition, if some of these goals conflict with each other while others are complementary (i.e., some of the requirements for the two goals are the same), this can actually cause individual players to move into certain roles as the game progresses. Once a player can see that she is starting to fulfill one of the goals, she will pick its

complementary goals to pursue as well, and this will lead her to make tactical decisions that fulfill the role for which those goals were designed. If these goals cause her to take a specific type of action in the game, it can alter her in-game relationship with other players.

An example of this comes from the game *Settlers of Catan*, designed by Klaus Teuber. In this game, players acquire resources through random die rolls and trade, and some of the five game resources are useful in the early game while others are useful at the end. Three that are less useful at the beginning are sheep, wheat, and ore; however, together, the three can be traded for a development card. The most common development card is the soldier card, which can move the robber token onto any space, allowing the player moving it to steal from another player. Therefore, an excess of ore, wheat, and sheep at the beginning of the game can lead the player to purchase development cards, and because having the largest number of soldier cards played can earn the player victory points, the combination of that resource and that potential goal can influence the player to rob the other players more often and actually make her play the role of the bully in the game.

House Rules

House rules occur when the players themselves intentionally modify the rules. As you saw in the Bartok game example, even a simple rule change can have drastic effects on the game. For instance, most players of *Monopoly* have house rules that cut the auction of property (which would normally happen if a player landed on an unowned property and chose not to buy it) and add collection of all fines to the Free Parking space to be picked up by any player who lands on that space. The removal of the auction rule removes nearly all potential strategy from the beginning of *Monopoly* (converting it into an extremely slow random property distribution system), and the second rule removes some determinism from the game (since it could benefit any player, either the one in the lead or in last place). Though the first house rule in this example makes the game a bit worse, most house rules are intended to make games considerably

more fun.[7] In all cases, house rules are an example of the players beginning to take some ownership of the game, making it a little more theirs and a little less the designer's. The fantastic thing about house rules is that they are many people's first experimentation with game design.

[7] If you're ever playing the game *Lunch Money* by Atlas Games, try allowing players to attack another player, heal themselves, **and** discard any cards they don't want each turn (rather than having to choose one of the three). It makes the game a lot more frantic!

Player Intent: Bartle's Types, Cheaters, Spoilsports

One thing that you have little or no control over is the intent of your players. While most players will be playing your game rationally to win, you may also have to contend with cheaters and spoilsports. Even within legitimate players of games, you will find four distinct personality types as defined by Richard Bartle, one of the designers of the first MUD (Multi-User Dungeon, a text-based online ancestor of modern massively multiplayer online roleplaying games). The four types of players that he defined have existed since his early MUD and carry through all multiplayer online games today. His 1996 article "Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs"[8] contains fantastic information on how these types of players interact with each other and the game as well as information about how to grow your community of players in positive ways.

[8] Richard Bartle, "Hearts, Clubs, Diamonds, Spades: Players Who Suit Muds," http://www.mud.co.uk/richard/hcds.htm, accessed February 2, 2014.

Bartle's four types (which he identified with the four suits of a deck of cards) are as follows:

• **Achiever (Diamond):** Seeks to get the highest score in the game. Wants to dominate the game.

• **Explorer (Spade):** Seeks to find all the hidden places in the game. Wants to understand the game.

• **Socializer (Heart):** Wants to play the game with friends. Wants to understand the other players.

• **Killer (Club):** Wants to provoke other players of the game. Wants to dominate the other players.

These can be understood as belonging to a 2x2 continuum (also from Bartle's article). Figure 5.2 represents this graphically.

Figure 5.2. Richard Bartle's four players who suit MUDs[9]



[9] Adapted from: Richard Bartle, "Hearts, Clubs, Diamonds, Spades: Players Who Suit Muds," http://www.mud.co.uk/richard/hcds.htm, accessed February 2, 2014.

There are certainly other theories of player motivation and player types,[10] but Bartle's are the most widely recognized and understood in the game industry.

[10] See Nick Yee's "Motivations of Play in MMORPGs: Results from a Factor Analytic Approach," http://www.nickyee.com/Daedalus/motivations.pdf.

The other two player types that you may encounter are *cheaters* and *spoilsports*:

• **Cheaters:** Care about winning but don't care about the integrity of the game. Cheaters will bend or break the rules to win.

• **Spoilsports:** Don't care about winning or about the game. Spoilsports will often break the game to ruin other players' experiences.

Neither of these are players that you want in your game, but you need to understand their motivations. For instance, if a cheater feels that she has a chance of winning legitimately, she may not feel as driven to cheat. Spoilsports are much more difficult to deal with since they don't care about the game or winning, but you rarely have to deal with spoilsports in digital single-player games, because they would have no reason to play the game if they weren't interested in it in the first place. However, even great players can sometimes become spoilsports when they encounter terrible game mechanics...often right before they choose to turn the game off.

Outcome

Outcome is the result of playing the game. All games have an outcome. Many traditional games are *zero sum,* meaning that one player wins and the other loses. However, this is not the only kind of outcome that a game can have. In fact, every individual moment in a game has its own outcome. There are several different levels of outcome in most games:

• **Immediate outcome:** Each individual action has an outcome. When a player attacks an enemy, the outcome of that attack is either a miss or a hit and the resultant damage to the enemy. When a player purchases property in *Monopoly*, the outcome is that the player has less money available but now owns the potential to earn more money.

• **Quest outcome:** In many games, the player will be sent on missions or quests and will gain some sort of reward for completing that quest. Missions and quests often have narratives constructed around them (e.g., a little girl has lost her balloon in *Spider-Man 2*, so Spider-Man must retrieve it for her), and the outcome of the quest also marks the end of the tiny narrative surrounding it.

• **Cumulative outcome:** When the player has been working toward a goal over time and finally achieves it, that is a cumulative outcome. One of the most common examples of this is leveling up in a game

with experience points (XP). Everything that the player does accrues a few experience points, and once the total number of XP has reached a threshold, the player's in-game character gains a new level, which grants the character a boost in stats or abilities. The main difference between this and a quest outcome is that the cumulative outcome usually doesn't have a narrative wrapped around it, and the player often reaches the cumulative outcome passively while actively doing something else (e.g., a player of *Dungeons & Dragons 4th Edition* actively takes part in a game session and then, while adding up earned XP at the end of the evening, notices that she has exceeded 10,000 XP and achieved level 7.)[11]

[11] Rob Heinsoo, Andy Collins, and James Wyatt, *Dungeons & Dragons Player's Handbook: Arcane, Divine, and Martial Heroes: Roleplaying Game Core Rules* (Renton, WA: Wizards of the Coast, 2008).

• **Final outcome:** Most games have an outcome that ends the game: A player wins chess (and the other loses), a player finishes *Final Fantasy VII* and saves the world from Sephiroth, and so on. There are a few games where the final outcome doesn't end the game (e.g., in *Skyrim*, even when the player has finished the main quest, she can still continue to play in the world and experience other quests). Interestingly, the death of the player character is very rarely a final outcome in games.

In the few games where death is a final outcome (e.g., the game *Rogue*, where a single loss will cause the player to lose all progress in the game), the individual game session is usually relatively short so that the player doesn't feel a tremendous loss at the death of the player character. In most games, however, death is just a temporary setback and in-game checkpoints usually ensure that the player never loses more than five minutes of progress in the game.

Dynamic Aesthetics

Just as with dynamic mechanics, dynamic aesthetics are those that emerge when playing the game. There are two different primary categories:

• **Procedural aesthetics:** Aesthetics that are programmatically generated by digital game code (or via the application of mechanics in a paper game). These include procedural music and art that emerge directly from inscribed aesthetics and technology.

• **Environmental aesthetics:** These are the aesthetics of the environment in which the game is played, and they are largely beyond the control of the game developers.

Procedural Aesthetics

Procedural aesthetics, as we generally think of them in digital games, are created programmatically by combining technology and inscribed aesthetics.[12] These are called *procedural* because they arise from procedures (also known as functions) that have been written as programming code. If you look at the cascading waterfall of objects that is created in the first programming chapter (Chapter 18, “Hello World: Your First Program”), that could be considered procedural art because it is an interesting visual that emerges from C# programming code. In professional games, two of the most common forms of procedural aesthetics are music and visual art.

[12] There are also examples of procedural art in board games, including things like the map created by the progressive laying of tiles in *Carcassonne*, but the most common procedural game art is digital.

Procedural Music

Procedural music has become very common in modern videogames, and it is currently created through three different techniques:

• **Horizontal Re-Sequencing (HRS):** HRS rearranges the order of several precomposed sections of music according to the emotional impact that the designers wish for the current moment in the game. An example of this is LucasArts' iMUSE (Interactive MUsic Streaming Engine), which was used in the *X-Wing* game series as well as many of LucasArts adventure games. In *X-Wing*, the pre-composed music is sections of John William's score for the *Star Wars* films. Using iMUSE, designers are able to play peaceful music when the player is just flying through space, ominous music when enemy forces are about to attack, victory music whenever a player destroys an enemy craft or achieves an objective, and so on. There are also longer sections of music that are meant to loop and provide a single mood as well as very short sections of music (one or two measures in length) that are used to mask the transition from one mood to the next. This is currently the most common type of procedural music technology and harkens at least as far back as *Super Mario Bros.*, which played a transitional musical sting and then switched to a faster version of the background music when the player had less than 99 seconds left to complete the current level.

• **Vertical Re-Orchestration (VRO):** VRO includes recordings of various tracks of a single song that can be individually enabled or disabled. This is used very commonly in rhythm games like *PaRappa the Rapper* and *Frequency*. In *PaRappa*, there are four different tracks of music representing four different levels of success for the player. The player's success is ranked every few measures, and if she either drops or increases in rank, the background music switches to a worse- or better-sounding track to reflect this. In *Frequency* and its sequel *Amplitude*, the player controls a craft traveling down a tunnel, the walls of which represent various tracks in a studio recording of a song. When the player succeeds at the rhythm game on a certain wall, that track of the recording is enabled.[13] VRO like this is nearly ubiquitous in rhythm games—with the fantastic Japanese rhythm game *Osu Tatake Ouendan!* and its Western successor *Elite Beat Agents* as marked exceptions—and has also become common in other games to give the player musical

feedback on the health of their character, speed of their vehicle, and so on.

[13] *Amplitude* also includes a mode where players can choose which tracks to enable at any point in the song to use HRS to create their own remix of the tracks included with the game.

• **Procedural Composition (PCO):** PCO is the most rare form of procedural music because it takes the most time and skill to execute. In PCO, rather than rearrange various precomposed tracks of music or enable and disable precomposed tracks, the computer program actually composes music from individual notes based on programmed rules of composition, pacing, etc. One of the earliest commercial experiments in this realm was *C.P.U. Bach* by Sid Meier and Jeff Brigs, a title for the 3DO console. In *C.P.U. Bach*, the listener/player was able to select various instruments and parameters, and the game would craft a Bach-like musical composition based on procedural rules.

Another fantastic example of procedural composition is the music created by composer and game designer Vincent Diamante for the game *Flower* by thatgamecompany. For the game, Diamante created both precomposed sections of music and rules for procedural composition. During gameplay, background music is usually playing (some of which is re-arranged based on the situation using HRS) as the player flies over flowers in a field and opens them by passing near. Each flower that is opened creates a single note as it blooms, and Diamante's PCO engine chooses a note for that flower that will blend harmoniously with the precomposed music and create a melody along with other flower notes. Regardless of when the player passes over a flower, the system will choose a note that fits well with the current audio soundscape, and passing over several flowers in sequence will procedurally generate pleasing melodies.

Procedural Visual Art

Procedural visual art is created when programming code acts dynamically to create in-game visuals. There are a few forms of procedural visuals with which you are probably already familiar:

• **Particle systems:** As the most common form of procedural visuals, particle systems are seen in almost every game these days. The dust cloud that rises when Mario lands a jump in *Super Mario Galaxy*, the fire effects in *Uncharted 3*, and the sparks that appear when cars crash into each other in *Burnout* are all various versions of particle effects. Unity has a very fast and robust particle effects engine (see Figure 5.3) that you will use to create a fire spell in Chapter 35, “Prototype 8: Omega Mage.”

Figure 5.3. Various particle effects that are included with Unity



• **Procedural animation:** Procedural animation covers everything from flocking behavior for groups of creatures to the brilliant procedural animation engine in Will Wright’s *Spore* that created walk, run, attack, and other animations for any creature that a player could design. With normal animation, the animated creatures always follow the exact paths inscribed by the animator. In procedural animation, the animated creatures follow procedural rules that emerge into complex motion and behavior. You will get some experience with the flocking behavior known as “boids” in Chapter 27, “Object-Oriented Thinking” (see Figure 5.4).

Figure 5.4. Boids, an example of procedural animation from Chapter 27, “Object-Oriented Thinking“



• **Procedural environments:** The most obvious example of a procedural environment in games is the world of *Minecraft* by Mojang. Each time a player starts a new game of *Minecraft*, an entire world (billions of square kilometers in size) is created for her to explore from a single seed number (known as the *random seed*).

Because digital random number generators are never actually random, this means that anyone who starts from the same seed will get the same world.

Environmental Aesthetics

The other major kind of dynamic aesthetics are those controlled by the actual, real-life physical environment in which the game is played. While these are largely beyond the control of the game designer, it is still the designer's responsibility to understand what environmental aesthetics might arise and accommodate them as much as possible.

Visual Play Environment

Players will play games in a variety of settings and on a variety of equipment, so it's necessary as a designer to be aware of the issues that this may cause. You should accommodate two elements in particular:

• **Brightness of the environment:** Most game developers tend to work in environments where the light level is carefully controlled to make the images on their screen as clear as possible. Players don't always interact with games in environments with perfect lighting. If your player is on a computer outside, playing on a projector, or playing anywhere else with imperfect control of lighting, it can be very difficult for them to clearly see scenes in your game that have a low level of brightness (e.g., a scene taking place in a dark cave). Remember to make sure that your visual aesthetic either has a lot of contrast in it between light and dark or allows the player the ability to adjust the gamma or brightness level of the visuals. This is especially important if designing for a phone or other mobile device, since these can easily be played outside in direct sunlight.

• **Resolution of the player's screen:** If you are developing for a fixed-screen device like a specific model of iPad or portable console (like the Nintendo DS), this won't be an issue. However, if you're

designing for a computer or traditional game console, you have very little control over the resolution or quality of your player's screen, particularly if it's a console game. You cannot assume that your player will have a 1080p or even 720p screen. All modern consoles before the PS4 and Xbox One could still output the standard composite video signal that has existed for standard-definition television since the 50s. If you're dealing with a player on a standard-def television, you will need to use a much larger font size to make it at all legible. Even AAA games like the *Mass Effect* and *Assassin's Creed* series have failed to accommodate this well in the past, making it impossible to read critical text in these games on any television made more than 10 years prior to their release. You never know when someone might be trying to play your game on older equipment, but it is possible to detect whether this is the case and change the font size to help them out.

Auditory Play Environment

As with the visual play environment, you rarely have control over the audio environment in which your game is played. Though this is most essential to accommodate when making a mobile game, it's also important to keep in mind for any game. Things to consider include the following:

• **Noisy environments:** Any number of things may be happening at the same time as your game, so you need to make sure that your player can still play even if they miss or can't hear some of the audio. You also need to make sure that the game itself doesn't create an environment so noisy that the player misses critical information. In general, important dialog and spoken instructions should be the loudest sounds in your game, and the rest of the mix should be kept a little quieter. You will also want to avoid subtle, quiet audio cues for anything important in the game.

• **Player-controlled game volume:** The player might mute your game. This is especially true with mobile games where you can never count on the player to be listening. For any game, make sure

that you have alternatives to sound. If you have important dialogue, make sure to allow the player to turn on subtitles. If you have sound cues to inform players of where things are, make sure to include visual cues as well.

Player Considerations

Another critical thing to consider about the environment in which your game will be played is the player herself. Not all players have the optimal ability to sense all five aesthetics. A player who is deaf should really be able to play your game with little trouble, especially if you follow the advice in the last few paragraphs. However, there are two other considerations in particular that many designers miss:

• **Colorblindness:** About 7% to 10% of white men have some form of colorblindness. There are several different forms of deficiency in color perception, the most common of which causes a person to be unable to differentiate between similar shades of red and green. Because colorblindness is so common, you should be able to find a colorblind friend that you can ask to playtest your game and make sure that there isn't key information being transmitted by color in a way that they can't see. Another fantastic way to check for your game is to take a screen shot and bring it into Adobe Photoshop. Under the View menu in Photoshop is a submenu called Proof Setup, and in there, you can find settings for the two most common types of color blindness. This will enable you to view your screen shot as it would be viewed by your colorblind players.

• **Epilepsy and migraine:** Both migraines and epileptic seizures can be caused by rapidly flashing lights, and children with epilepsy are particularly prone to having seizures triggered by light. In 1997, an episode of the *Pokémon* television show in Japan triggered simultaneous seizures in hundreds of viewers because of flickering images in one scene.[14] Nearly all games now ship with a warning that they may cause epileptic seizures, but the occurrence of that is now very rare because developers have accepted the responsibility

to think about the effect their games might have on their players and largely removed rapidly flashing lights from their games.

[14] Sheryl WuDunn, “TV Cartoon’s Flashes Send 700 Japanese Into Seizures,” *New York Times*, December 18, 1997.

Dynamic Narrative

There are several ways of looking at narrative from a dynamic perspective. The epitome of the form is the experience of players and their game master when playing a traditional pen-and-paper roleplaying game. While there have certainly been experiments into crafting truly interactive digital narratives, after over 30 years, they still haven’t reached the level of interaction in a well-run game of *Dungeons & Dragons* (*D&D*). The reason that *D&D* can create such fantastic dynamic narratives is that the dungeon master (DM: the player running the game for the others) is constantly considering the desires, fears, and evolving skills of her players and crafting a story around them. As mentioned earlier in this book, if the players run into a low-level enemy that (due to random die rolls working in its favor) is very difficult to fight, the DM can choose to have that enemy escape at the last minute and then return as a nemesis for the players to fight later. A human DM can adapt the game and the game narrative to the players in a way that is very difficult for a computer to replicate.

Interactive Narrative Incunabula

In 1997, Janet Murray, a professor at the Georgia Institute of Technology, published the book *Hamlet on the Holodeck*[15] in which she examined the early history of interactive narrative in relation to the early history of other forms of narrative media. In her book, Murray explores the incunabular stage of other media, which is the stage when that medium was between its initial creation and its mature form. For instance, in the incunabular stage of film, directors were attempting to shoot 10-minute versions of Hamlet and King

Lear (due to the 10-minute length of a single reel of 16mm film), and incunabular television was largely just televised versions of popular radio programs. Through comparisons to many examples from various media, Murray proceeds to talk about the growth of interactive digital fiction and where it is currently in its incunabular stage. She covers early Infocom text adventure games like the *Zork* series and *Planetfall* and points out two very compelling aspects that make interactive fiction unique.

[15] Janet Horowitz Murray, *Hamlet on the Holodeck* (New York: Free Press, 1997).

Interactive Fiction Happens to the Player

Unlike nearly every other form of narrative, interactive fiction is happening directly to the player. The following happens near the beginning of the Infocom game *Zork*. (The lines preceded by a right angle bracket [e.g., `> open trap door`] are the commands entered by the player.)

...With the rug moved, the dusty cover of a closed trap door appears.

> open trap door

The door reluctantly opens to reveal a rickety staircase descending into darkness.

> down

It is pitch dark. You are likely to be eaten by a grue.

> light lamp

The lamp is now on.
You are in a dark and damp cellar with a narrow passageway leading east and a crawlway to the south. To the west is the bottom of a

steep metal ramp which is unclimbable.
The door crashes shut, and **you hear someone barring it**.[16]

[16] *Zork* was created at the Massachusetts Institute of Technology in 1977–79 by Tim Anderson, Marc Blank, Bruce Daniels, and Dave Lebling. They formed Infocom in 1979 and released *Zork* as a commercial product.



The key element here is that *you* hear someone barring it. *You* are now trapped. Interactive fiction is the only narrative medium where the player/reader is the character taking actions and suffering consequences in the narrative.

Relationships Are Developed Through Shared Experience

Another compelling aspect of interactive fiction is that it allows the player to develop a relationship with other characters through shared experience. Murray cites *Planetfall*,[17] another Infocom text adventure, as a fantastic example of this. Following the destruction of the spaceship on which she was a janitor, the player is largely alone for the first section of *Planetfall*. Eventually, she comes across a machine to make warrior robots, but when she engages it, it malfunctions and produces a child-like, mostly useless robot named Floyd. Floyd follows the player around for the remainder of the game and does little more than provide comic relief. Much later in the game, there is a device locked in a bio-lab that the player must retrieve, but the lab is full of both radiation and vicious aliens. Immediately, Floyd simply says “Floyd go get!” and enters the lab to retrieve the item. Floyd soon returns, but he is leaking oil and barely able to move. He dies in the player’s arms as she sings *The Ballad of the Starcrossed Miner* to him. Many players reported to the designer of *Planetfall*, Steven Meretzky, that they cried when Floyd died, and Murray cites this as one of the first examples of a tangible emotional connection between a player and an in-game character.

[17] *Planetfall* was designed by Steve Meretzky and published by Infocom in 1983.

Emergent Narrative

True dynamic narrative emerges when the players and game mechanics contribute to the story. Several years ago, I was playing in a *Dungeons & Dragons* 3.5 edition game with some friends. The game master had us in a pretty tight spot. We had just retrieved an artifact from some forces of evil in another dimension and were being chased by a large balrog[18] as we fled down a narrow cave on our flying carpet toward the portal back to our dimension. It was gaining on us quickly, and our weapons were having little effect. However, I remembered a little-used property of the Rod of Splendor that I possessed. Once per week, I could use the Rod of Splendor to create a "huge pavilion of silk, 60 feet across, inside of which were the furnishings and food for a party to entertain 100 people."[19] Usually, we would use this capability of the rod to throw a party when we'd finished a mission, but this time I cast the tent directly behind us in the tunnel. Because the tunnel was only 30 feet wide, the balrog crashed into the tent and became entangled, allowing us to escape without anyone dying.

[18] A balrog is the giant winged demon of fire and smoke that faced Gandalf in the "you shall not pass" scene of *The Fellowship of the Ring* by J. R. R. Tolkien.

[19] The *Dungeons & Dragons 3.5e System Reference Document* entry for the Rod of Splendor is at http://www.d20srd.org/srd/magicItems/rods.htm#splendor.

This kind of unexpected story emerges from a combination of the situation created by the game master, the game's rules, and the creativity of individual players. I have encountered many similar stories through the roleplaying campaigns that I have been part of (as both a player and game master), and you can do several things

to encourage this kind of collaborative storytelling in roleplaying campaigns that you run. For more information about roleplaying games and how to run a good campaign, see the “Roleplaying Games” section of Appendix B, “Useful Concepts.”

Dynamic Technology

As with the previous chapter, because other large sections of this book are devoted to game technology it is covered very little in this chapter. The core concept for you to know at this point is that the game code you author (your inscribed technology) will be a system that runs as the player is experiencing the game. As with all dynamic systems, emergence will occur, and this means that there is both the opportunity for wonderful, unexpected things to happen and the danger of horrible, unexpected things happening. Dynamic technology covers all of the runtime behavior of your code and the ways in which it affects the player. This could be anything from a system to simulate physics to artificial intelligence code to anything else that is implemented in your code.

To find information on the dynamic behavior of paper game technologies such as dice, spinners, cards, and other randomizers, look to Chapter 11, “Math and Game Balance.” For information on digital game technologies, you can look to the latter two parts of the book as well as Appendix B, “Useful Concepts.”

Summary

Dynamic mechanics, aesthetics, narrative, and technology all emerge from the act of players playing a game. While the elements that emerge can be challenging to predict, it is the responsibility of designers to playtest in order to understand the envelope of that emergence.

The next chapter will explore the cultural layer of the Layered Tetrad, the layer beyond gameplay. It is in the cultural layer that players gain

more control over the game than the original game developers, and the cultural layer is the only layer of the tetrad that is experienced by members of society who do not ever play the game.

Chapter 7. Acting Like a Designer

Now that you've learned something about how to take a designer's approach to thinking about and analyzing games, it's time to look at the way that game designers go about crafting interactive experiences.

As mentioned in previous chapters, game design is a practice, and therefore, the more design you do, the better you will get at it. However, you also need to make sure that you're starting with the kinds of effective practices that will yield you the greatest gains over time. That is the purpose of this chapter.

Iterative Design

"*Game design is 1% inspiration and 99% iteration*." —Chris Swain

Remember this saying from the first chapter? In this section, we explore it further.

The number one key to good design—in fact, the most important thing that you can learn from this book—is the process of iterative design shown in Figure 7.1. I have seen iterative design take some games that were initially terrible and make them great, and I've seen it at work across all forms of design from furniture to illustration to game design.

Figure 7.1. The iterative process of design[1]



[1] Based on: Tracy Fullerton, Christopher Swain, and Steven Hoffman, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games* (Burlington, MA: Morgan Kaufmann Publishers, 2008), 36.

The four phases of the iterative process of design are:

• **Analysis:** The analysis phase is all about understanding where you are and what you want to accomplish. You must clearly understand the problem that you're trying to solve (or opportunity that you're trying to take advantage of) with your design. You must also understand the resources that you can bring to bear on the project and the amount of time you have in which to implement your design.

• **Design:** Now that you have a clear idea where you are and what you're trying to accomplish with your design, create a design that will solve the problem/opportunity with the resources you have available to you. This phase starts with brainstorming and ends with a concrete plan for implementation.

• **Implementation:** You have the design in hand; now execute it. There's an old adage: "A game is not a game until people are playing it." The implementation phase is about getting from game design idea to playable prototype as quickly as possible. As you'll see in the digital game tutorials later in this book, the earliest implementations are sometimes just moving a character around the screen—with no enemies or objectives—and seeing if the movement feels responsive and natural. It's perfectly fine to implement just a small part of the game before testing; a test of just a portion of the game can often be more focused than a large-scale implementation could be. At the end of implementation, you're ready to run a playtest.

• **Testing:** Put people in front of your game and get their reactions. As your experience as a designer grows, you will get better at knowing how the various game mechanics you design will play-out once the game is being tested, but even with years of experience, you will never know for sure. Testing will tell you. You always need to test early, when it's still possible to make changes to the game and get it on the right track. Testing must also be done very frequently so that you can best understand the causes of the changes in player feedback that you witness.

Let’s look at each phase in more detail.

Analysis

Every design seeks to solve a problem or take advantage of an opportunity, and before you can start to design, you need to have a clear idea of what that problem or opportunity is. You may be saying to yourself “I just want to make a great game,” which is true of most of us, but even with that as your initial statement, you can dig deeper and analyze your problem further.

To start, try asking yourself these questions:

1. For whom am I designing the game? Knowledge of your target audience can dictate many other elements of design. If you’re creating a game for children, it is more likely that their parents would let them use a mobile device than a computer connected to the Internet. If you’re designing a game for people who like strategy games, they will most likely be used to playing on a PC. If you’re designing a game for men, you should be aware that nearly 10% of white men are colorblind.

One thing that you should always be aware of is the danger of designing a game for yourself. If you just make a game for you, there’s a legitimate possibility that *only* you will want to play it. Researching your intended audience and understanding what makes them tick can tell you a tremendous amount about where your game design should go and help you to make your game better.

It’s also important to realize that what players think they want and what they will actually enjoy are sometimes two different things. In your research, it’s important to try to differentiate between your audience’s stated desires and the things that actually motivate and engage them.

2. What are my resources? Most of us don’t have a budget of tens of millions of dollars with which to employ a studio of 200 people to

make a game over the span of two years. But you probably do have some time and talent and maybe even a group of talented friends as well. Being honest with yourself about your resources, strengths, and weaknesses can help shape your design. As an independent developer, your primary resources are talent and time. Money can help you purchase either of these through hiring contractors or purchasing assets, but especially if you're working on a small indie game team, you want to make sure that the game you're developing makes the best use of the resources on your team. When working on a game, you should treat your time and that of your team members as an precious resource; be sure not to waste it.

3. What prior art exists? This is the single question that is most often ignored by my students (often to their detriment). *Prior art* is the term used to describe existing games and other media that are related to yours in some way. No game comes from a vacuum, and as a designer, it is up to you to know not only the other games that have inspired you (which, of course, you know), but also what other games exist in the same space that came before or after your primary inspirations.

For instance, if you were to design a first-person shooter for console, of course you'd look at *Destiny*, *Titanfall*, and the *Call of Duty: Modern Warfare* series, but you would also need to be familiar with *Halo* (the first game that made first-person shooter [FPS] work on a console when conventional wisdom held that it was impossible to do so), *Marathon* (Bungie's game prior to *Halo*, which forms the basis for a lot of the design decisions and mythology in *Halo*), and the other FPSs that were precursors for *Marathon*.

Prior art research is necessary because you must understand everything you can about the ways that other people have tried to approach the design problem that you're tackling. Even if someone else had the exact same idea as you, it's almost certain that they approached it in a different way, and understanding both their successes and failures will better equip you to make your game better.

4. What is the fastest path to a playable game that demonstrates what I want to test? Though often overlooked, this question is critical for obvious reasons. You only have 24 hours available to you each day, and if you're at all like me, only a small fraction of them can be devoted to game development. Knowing this, it is critical that your time is used as efficiently as possible if you want to get your game made. Think about the *core mechanic* of the game you want to create—the thing that the player does most throughout the game (for example, in *Super Mario Bros.*, the core mechanic is jumping)—and make sure that you design and test that first. From that, you'll know whether it's worth it to make more of the game. Art, music, and all other aesthetic elements are certainly important to your final game, but at this point, you must focus on the mechanics—on gameplay—and get that working first. That is your goal as a game designer.

Of course, you'll have many more questions of your own to add to these, but regardless of the game you're making, these four are critical to keep in mind during the analysis phase.

Design

A large portion of this book is about design, but in this section, I'm going to focus on the attitude of a professional designer. (Chapter 14 "The Digital Game Industry," covers this in more detail.)

Design isn't about getting your way, it's not about being a great genius or auteur who is followed by everyone else on the team, and it's not even about doing a great job of communicating your vision to the rest of the team. Design isn't about you; it is about the project. Working as a game designer is about collaborating with the rest of the team, compromising, and above all listening.

In the first few pages of his book *The Art of Game Design,* Jesse Schell states that listening is the most important skill that a game designer can have, and I emphatically agree. Schell lists five kinds of listening that you need to develop:[2]

[2] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 4-6.

• **Listen to your audience:** Whom do you want to play your game? Whom do you want to buy your game? As mentioned above, these are questions that you must answer, and after you have answered them, you need to listen to the kinds of experiences that your audience wants to have. The whole purpose of the iterative process of design is to make something, throw it out to playtesters, and get their feedback. Make sure you're listening to that feedback when they give it, even (especially!) if it's not what you expected or what you want to hear.

• **Listen to your team:** On most game projects, you'll be working with a team of other talented people. Your job as the designer is to listen to all of their thoughts and ideas and work with them to unearth the ideas that will create the best game for your audience. If you surround yourself with people who are willing to speak up when they disagree with you, you will have a better game. Your team should not be contentious; rather, it should be a team of creative individuals who all care passionately about the game.

• **Listen to your client:** A lot of the time, as a professional game designer, you'll be working for a client (boss, committee, etc.), and you're going to need to listen to their input. They aren't usually going to be expert game designers—that's why they hired you—but they will have specific needs that you must meet. At the end of the day, it will be your job to listen to them at several levels: what they tell you they want, what they think they want but don't say out loud, and even what they really want deep down but might not even admit to themselves. With clients, you need to listen very carefully in order to leave them with not only an excellent game but also an excellent impression of working with you.

• **Listen to your game:** Sometimes certain elements of a game design fit together like a hand in a glove, and sometimes, it's more like a wolverine in a Christmas stocking (p.s.: *bad idea*). As the

designer, you'll be the team member closest to the gameplay, and it will be up to you to understand the game from a *gestalt* (i.e., holistic) perspective. Even if a certain aspect of a game is brilliant design, it might not fit well with the rest. Don't worry, if it is a great bit of design, there's a good chance you can find a place for it in another game. You'll have lots of chances across the many games you'll make in your career.

• **Listen to yourself:** There are several important aspects of listening to yourself:

• Listen to your gut: Sometimes you'll get a gut feeling about something, and sometimes these will be wrong, but other times they'll be very right. When your gut tells you something about a design, give it a try. It may be that some part of your mind figured out the answer before your conscious mind had a chance to.

• Listen to your health: Take care of yourself and stay healthy. Seriously. There is a tremendous amount of research out there showing that pulling all-nighters, being stressed, and not exercising have a real and tremendously negative effect on your ability to do creative work. To be the best designer you can be, you need to be healthy and well rested. Don't let yourself get caught in a cycle of one crisis after another that you try to solve by working crazy hours into the night.

• Listen to how you sound to other people: When you say things to your colleagues, peers, friends, family, and acquaintances, take a moment every once in a while to really listen to how you sound. I don't want you to get a complex about it or anything, but I do want you to listen to yourself and ask these questions:

Do I sound respectful?

Do I sound like I care about the other person?

Do I sound like I care about the project?

All other things being equal, the people who do best in life are those who consistently demonstrate respect and care for others. I've known some really talented people who didn't get this; they did all right initially, but without fail, their careers sputtered and failed as fewer and fewer people wanted to work with them. Game design is a community of shared respect.

There are, of course, many more aspects to acting like a professional designer than just listening, but Schell and I agree that it is one of the most important. The rest of this book covers more nuts-and-bolts aspects of being a designer, but all of it must be approached with a humble, healthy, collaborative, and creative attitude.

Implementation

The latter two-thirds of this book are about digital implementation, but it's important to realize that the key to effective implementation in the process of iterative design is to get from design to playtest in the *most efficient* way possible. If you're testing the jump of a character in a platform game like *Super Mario Bros.* or *Mega Man*, you will need to make a digital prototype. However, if you're testing a graphical user interface (GUI) menu system, you don't need to build a fully working digital version; it's perfectly fine to print out images of the various states of the menu and ask testers to navigate through them with you acting as the computer (and swapping the printed images by hand).

A paper prototype can enable you to quickly test ideas and generate feedback. They usually take drastically less time to implement than digital prototypes and can give you the unique ability to change the game rules in the middle of a play session if the initial rules aren't working. Chapter 9, "Paper Prototyping," includes in-depth information about paper prototyping techniques and both good and bad uses for paper prototypes.

Another important way that you can shorten your implementation time is to realize that you don't' have to do everything yourself. Many of my new students approach game development with a desire to learn it all: they want to design the game; write all the code; model, texture, rig, and animate game characters; build environments; write the story; create game code; and sometimes even want to write their own game engine. If you were a multimillion-dollar studio with years of time, this might be an okay idea, but as an independent designer, it's ludicrous. Even indie developers like Notch (the creator of *Minecraft*), who are often seen as solitary geniuses have stood on the shoulders of *many* giants. *Minecraft* was initially based on an open-source project created by many other people. If you wanted to make a computer game, you could start by building a computer from individual transistors, but that would be ludicrous. It's nearly as ridiculous to think that you would want to write your own game engine. I chose *Unity* as the game engine for this book because there are hundreds of people working at Unity Technologies every day to make our job as game developers easier. By trusting them to do their job well, I enable myself to focus on the interesting work of game design and development that I would much rather do than write my own game engine.[3]

[3] If you do really want to write your own game engine, my friend Jason Gregory has written a fantastic book on the subject:

Similarly, the *Unity Asset Store* is a fantastic place to trade money for time. The Asset Store enables you to purchase thousands of time-saving assets, including models, animations, and code libraries for everything from controller input to better text rendering, to gorgeous physically-based rendering libraries.[4] It also includes several free assets that you can easily use as placeholders in your prototypes. Any time you're thinking about taking the time to write a robust, reusable piece of code for one of your prototypes, I'd recommend checking on the Asset Store to see if someone else has already done it for you. Kicking that person a few bucks could save you dozens of hours of development time.

[4] For these three things, I recommend: Controller Input - *InControl* by _____, Better Text Rendering - *TextMeshPro* by _____, Physically-Based Rendering - *Alloy* by RUST LTD.

Testing

Once you've gotten the barest minimum of a prototype working, it's time to test it. The key thing to keep in mind now is that regardless of what you think about your game, you won't really know anything until a player *who is not you* has tested it and given you feedback. The more people who play your game, the more accurate that feedback will be.

In my Game Design Workshop class at the University of Southern California, each of our board game projects took place over four weeks of labs. In the first lab, the students were placed in teams and given time to brainstorm their game ideas. Every subsequent lab was devoted entirely to playtesting the latest prototypes of their games. By the end of a 4-week project, each student team had completed nearly six hours of in-class playtesting and had drastically improved their designs as a result. The best thing you can do for your designs is to have people playing them and giving you feedback as often as possible. And, for the sake of all that is good, please write down what your playtesters tell you. If you forget what they said, the playtest is a waste.

It is also important to make sure that your playtesters are giving you honest feedback. Sometimes, playtesters will give you overly positive feedback because they don't want to hurt your feelings. In *The Art of Game Design,* Jesse Schell recommends telling your testers something like "I need your help. This game has some real problems, but we're not sure what they are. Please, if there is anything at all you don't like about this game, it will be a great help to me if you let me know"[5] to encourage them to be honest with you about flaws they see in the game.

[5] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 401.

Chapter 10, "Game Testing," covers several different aspects of testing in much more detail.

Iterate, Iterate, Iterate, Iterate, Iterate, Iterate, Iterate!

After you have run your playtest, you should have a lot of feedback written down from your testers. Now it's time to analyze again. What did the players like? What didn't they like? Were there places in the game that were overly easy or difficult? Was it interesting and engaging?

From all of these questions, you will be able to determine a new problem to solve with your design. Try to take time to interpret and synthesize player feedback (there's a side bar in Chapter 10, "Game Testing," about this). After doing so, try to pick a specific, achievable design goal for your next iteration. For instance, you might decide that you need to make the second half of the first level more exciting, or you may decide to reduce the amount of randomness in the game.

Each subsequent iteration of your game should include some changes, but don't try to change too many things or solve too many problems all at the same time. The most important thing is to get to the next playtest quickly and determine whether the solutions that you *have* implemented solved the problems they were meant to solve.

Innovation

In his book *The Medici Effect,*[6] author Frans Johansson writes about two kinds of innovation: *incremental* and *intersectional*.

[6] Frans Johansson, *The Medici Effect: What Elephants and Epidemics Can Teach Us about Innovation* (Boston, MA: Harvard Business School Press, 2006).

• **Incremental innovation** is making something a little better in a predictable way. The progressive improvement of Pentium processors by Intel throughout the 1990s was incremental innovation; each year, a new Pentium processor was released that was larger and had more transistors than the previous generation. Incremental innovation is reliable and predictable, and if you're looking for investment capital, it's easy to convince investors that it will work. However, as its name would suggest, incremental innovation can never make great leaps forward precisely because it is exactly what everyone expects.

• **Intersectional innovation** occurs at the collision of two disparate ideas, and it is where a lot of the greatest ideas can come from. However, because the results of intersectional innovation are novel and often unpredictable, it is more difficult to convince others of the merit of the ideas generated through intersectional innovation.

In 1991, Richard Garfield was trying to find a publisher for his game *RoboRally*. One of the people he approached was Peter Adkison, founder and CEO of Wizards of the Coast. Though Adkison liked the game, he didn't feel that Wizards had enough resources to publish a game like *RoboRally* that had so many different pieces, but he mentioned to Richard that they had been looking for a new game that could be played with very little equipment and resolve in 15 minutes.

Richard intersected this idea of a fast-play, low-equipment card game with another idea that had been kicking around in his head for a while—that of playing a card game with cards that were collected like baseball cards—and in 1993, Wizards of the Coast released *Magic: The Gathering*, which started the entire genre of collectible card games (CCGs).

Though Garfield had been thinking about a card game that was collectible for a little while before his meeting with Adkison, it was the intersection of that idea with Adkison's specific needs for a fast-play game that gave birth to the collectible card game genre, and nearly all CCGs that have come since have the same basic formula: a basic rule set, cards that have printed rules on them which override the basic rules, deck construction, and fast play.

The brainstorming procedure described next takes advantage of both kinds of innovation to help you create better ideas.

Brainstorming and Ideation

"The best way to have a good idea is to have a lot of ideas and throw out all the bad ones." —Linus Pauling, solo winner of both the Nobel Prize in Chemistry and the Nobel Peace Prize

Just like anyone else, not all of your ideas are going to be great ones, so the best you can do is to generate a lot of ideas and then sift through them later to find the good ones. This is the whole concept behind brainstorming. This section covers a specific brainstorming process that I have seen work very well for many people, especially in groups of creative individuals.

For this process, you will need: a whiteboard, a stack of 3x5 note cards (or just a bunch of slips of paper), a notebook for jotting down ideas, and various whiteboard markers, pens, pencils, and so on. The process works best with five to ten people, but you can alter it to work for fewer people by repeating tasks, and I've modified it in the past to work for a classroom of 65 students. (For instance, if you're by yourself, and it says that each person should do something once, just do it yourself multiple times until you're satisfied.)

Step 1: Expansion Phase

Let's say that you are just starting a 48-hour game jam with a few friends. The theme of the game jam is uroboros (the snake eating its

own tail symbol that was the theme of the Global Game Jam in 2012). Not much to go on, right? So, you start with the kind of brainstorming that you learned in grade school. Draw an uroboros in the middle of a white board, draw a circle around it, and start free-associating. Don't worry about what you're writing at this point—don't censor anything—just write whatever comes to mind as you go. Figure 7.2 shows an example.

Figure 7.2. The expansion phase of brainstorming a game for uroboros



Warning

BEWARE THE TYRANNY OF THE MARKER If you have more people taking part in the brainstorm than you have whiteboard markers, you should always be careful to make sure that everyone is being heard. Creative people come in all types, and the most introverted person on your team may have some of the best ideas. If you're managing a creative team, try to make sure that the more introverted members of your team are the ones holding the whiteboard markers. They may be willing to write something on the board that they aren't willing to say out loud.

When you're done, take a picture of the whiteboard. I have hundreds of pictures of whiteboards in my phone, and I've never regretted taking one. Once you've got it captured, email it out to everyone in the group.

Step 2: Collection Phase

Collect all of the nodes of the brainstorming expansion phase and write them each down on one 3x5 note card. These are called *idea cards* (see Figure 7.3), and they'll be used in the next phase.

Figure 7.3. Uroboros idea cards



A Quick Aside and a Bad Joke or Two

Let’s start with a bad joke:

There are two lithium atoms walking along, and one says to the other, “Phil, I think I lost an electron back there.” So Phil says, “Really Jason, are you sure?” And Jason replies, “Yeah, I’m positive!”

Here’s another:

Why was six afraid of seven?

Because seven eight nine!

Sorry, I know. They’re terrible.

You may be wondering why I’m subjecting you to these bad jokes. I’m doing so because jokes like these work on the same principle as intersectional innovation. Humans are creatures that love to think and combine weird ideas. Jokes are funny because they lead our minds down one track and then throw a completely different concept into the mix. Your mind makes the link between the two disparate, seemingly unrelated concepts, and the joy that causes comes across as humor.

The same thing happens when you intersect two ideas, and this is why it’s so pleasurable for us to get the eureka moment of intersecting two common ideas into a new uncommon one.

Step 3: Collision Phase

Here's where the fun begins. Shuffle together all the idea cards and deal two to each person in the group. Each person takes their two cards up to the whiteboard and reveals them to everyone. Then the group collectively comes up with three different game ideas inspired by the collision of the two cards. (If the two cards either are too closely paired or just don't work together at all, it's okay to skip them.) Figure 7.4 presents a couple of examples.

Figure 7.4. Uroboros idea collisions



Now, the examples in Figure 7.4 are just the first ideas that came to me, as they should be for you. We're still not doing a lot of filtering in this phase. Write down all of the different ideas that you come up with in this phase.

Step 4: Rating Phase

Now that you have a lot of ideas, it's time to start culling them. Each person should write on the whiteboard the two ideas from Step 3 that she thinks have the most merit.

Once everyone has done this, then all people should simultaneously put a mark next to the three ideas written on the board that they like the most. You should end up with some ideas with lots of marks and some with very few.

Step 5: Discussion

Continue the culling process by modifying and combining several of the ideas with the highest rating. With dozens of different crazy ideas to choose from, you should be able to find a couple that sound really good and to combine them into a great starting point for your design.

Changing Your Mind

Changing your mind is a key part of the iterative design process. As you work through the different iterations of your game, you will inevitably make changes to your design.

As shown in Figure 7.5, no one ever has an idea and turns it directly into a game with no changes at all (as shown in the top half of the figure), or if anyone ever does, it's almost certain to be a terrible game. In reality, what happens is a lot more like the bottom half of the figure. You have an idea and make an initial prototype. The results of that prototype give you some ideas, and you make another prototype. Maybe that one didn't work out so well, so you backtrack and make another. You continue this process until you've forged your idea over time into a great game, and if you stick to the process and engage in listening and creative collaboration, it'll be a much better game than the original one you set out to make.

Figure 7.5. The reality of game design



As the Project Progresses, You're More Locked In

The process just described is fantastic for small projects or the preproduction phase of any project, but after you have a lot of people who have put a lot of time into something, it's much more difficult and expensive to change your mind. A standard professional game is developed in several distinct phases:

• **Preproduction:** This is the phase covered by most of this book. In the preproduction phase, you're experimenting with different prototypes, and you're trying to find something that is demonstrably enjoyable and engaging. During preproduction, it is perfectly fine to change your mind about things. On a large industry project, there would be between 4 and 16 people on the project during preproduction, and at the end of this phase, you typically would want to have created a *vertical slice*, which is a short, five-minute section of your game at the same level of quality as the final game. This is

like a demo level for the executives and other decision-makers to play and decide whether or not to move the game into production. Other sections of your game should be designed at this point, but for the most part, they won't be implemented.

• **Production:** In the industry, when you enter the production phase of a game, your team will grow considerably in size. On a large game title, there could be well over 100 people working on the game at this point, many of whom might not be in the same city or even country as you. During production, all of the *systems design* (i.e., the game mechanics) need to be locked down very early, and other design aspects (like level design, tuning character abilities, and such) will be progressively locked down throughout production as the team finalizes them. From an aesthetics side, the production phase is when all of the modeling, texturing, animation, and other implementation of aesthetic elements take place. The production phase expands the high quality of the vertical slice out across the rest of the project.

• **Alpha:** Once you've reached the alpha phase of your game, all the functionality and game mechanics should be 100% locked down. At this point, there are no more changes to the systems design of the game, and the only changes you should make to things like level design will be in response to specific problems discovered through playtesting. This is the phase where the playtesting transitions to quality assurance (QA) testing in an effort to find problems and bugs (See Chapter 10, "Game Testing," for more information). When you start alpha, there may still be some bugs (i.e., errors in programming), but you should have identified all of them and know how to reproduce them.

• **Beta:** Once you're in beta, the game should be effectively done. At beta, you should have fixed any bugs that had the potential to crash your game, and the only remaining bugs should be minor. The purpose of the beta period is to find and fix the last of the bugs in your game. From the art side, this means making sure that every texture is mapped properly, that every bit of text is spelled properly,

etc. You are not making any new changes in the beta phase, just fixing any last problems that you can find.

• **Gold:** When your project goes gold, it is ready to ship. This name is a holdover from the days of CD-ROM production when the master for all the CDs was actually a disc made of gold that the foil layer of each CD was physically pressed onto. Now that even disc-based console games have updates delivered online, the gold phase has lost some of its finality, but gold is still the name for the game being ship-ready.

• **Post-release** – With the ubiquity of the Internet today, all games that aren't on cartridges (e.g., Nintendo DS games and some 3DS games are delivered on cartridges) can be *tuned*[7] after they're released. The post-release period can also be used for development of downloadable content (DLC). Because DLC is often composed of new missions and levels, each DLC release goes through the same phases of development as the larger game (though on a much smaller scale): preproduction, production, alpha, beta, and gold.

[7] Tuning is the term for the final stages of adjustments to game mechanics where only tiny changes are made.

Even though your initial projects will usually be much smaller than the professional ones just described, it is still imperative that you lock yourself into design decisions as early as is reasonable. On a professional team, a major design change in the production phase can cost millions of dollars, but on an indie team, it can easily push the release of the game back months, years, or forever. As you move forward in your career, no one will care about your half-finished games or unimplemented game ideas, but everyone *will* care about the games you have finished and shipped. Shipping games builds a reputation for effectiveness, and that's what people are looking for in a game developer.

Scoping!

One critical concept you must understand to act like a game designer is how to *scope* your work. Scoping is the process of limiting the design to what can reasonably be accomplished with the time and resources that you have available, and overscoping is the number one killer of amateur game projects.

I'll say that again: *Overscoping is the number one killer of game projects.*

Most of the games you see and play took dozens of people months and months of full-time work to create. Some large console games cost nearly $500 million to develop. The teams on these projects are all composed of fantastic people who have been doing their jobs well for years.

I'm not trying to discourage you, but I am trying to convince you to think small. For your own sake, don't try to make the next *Titanfall* or *World of Warcraft* or any other large, famous game you can think of. Instead find a small, really cool core mechanic and explore it deeply in a small game.

If you want some fantastic inspiration, check out the games that are nominated each year at the IndieCade Game Festival. IndieCade is the premier festival for independent games of various sizes, and I think it represents the vanguard of where independent games are going.[8] If you take a look at their website (http://indiecade.com), you can see tons of fantastic games, each of which pushes the boundaries of gaming in a cool new way. Each of these was someone's passion project, and many of them took hundreds or thousands of hours of effort for a small team or an individual to create.

[8] For purposes of full disclosure, since 2013, I have served as IndieCade's Chair for Education and Advancement, and I am very proud to belong to such a great organization.

As you look at them, you might be surprised by how small in scale they are. That's okay. Even though the scope of these games is

pretty small, they are still fantastic enough to be considered for an IndieCade award.

As you progress in your career, you may go on to make massive games like *Starcraft* or *Grand Theft Auto*, but remember that everyone got their start somewhere. Before George Lucas made *Star Wars,* he was just a talented kid in the film program at the University of Southern California. In fact, even when he made *Star Wars,* he scoped it down so perfectly that he was able to make one of the highest-grossing movies of all time for only $11 million. (It went on to make over $775 million at the box office and many, many times that in toy sales, home movie sales, and so on.)

So for now, think small. Come up with something that you know you can make in a short amount of time, work on it efficiently, and above all, **finish it**. If you make something great, you can always add on to it later.

Summary

The tools and theories you've read in this chapter are the kinds of things that I teach to my students and use in my personal design. I have seen the brainstorming strategies that I listed work in both big and small groups to create interesting, off-the-wall, yet implementable ideas, and every experience that I have had in the industry and academia has led me to feel that iterative design, rapid prototyping, and proper scoping are the key processes that you can implement to improve your designs. I cannot more highly recommend them to you.

Chapter 8. Design Goals

This chapter explores several important goals that you may have for your games. We cover everything from the deceptively complex goal of fun to the goal of experiential understanding, which may be unique to interactive experiences.

As you read this chapter, think about which of these goals matter to you. The relative importance of these goals to each other will shift as you move from project to project and will often even shift as you move through the various phases of development. However, you should always be aware of all of them, and even if one is not important to you, that should be due to deliberate choice rather than unintentional omission.

Design Goals: An Incomplete List

You could have any number of goals in mind when designing a game or interactive experience, and I'm sure that each of you has one that won't be covered in this chapter. However, I am going to try to cover most of the goals that I see in my personal work as a designer and in the design work of my students and friends.

Designer-Centric Goals

These are goals that are focused on you as the designer. What do you want to get out of designing this game?

• **Fortune**: You want to make money.

• **Fame**: You want people to know who you are.

• **Community**: You want to be part of something.

• **Personal expression**: You want to communicate with others through games.

• **Greater good**: You want to make the world better in some way.

• **Becoming a better designer**: You simply want to make games and improve your craft.

Player-Centric Goals

These goals are focused on what you want for the players of your game:

• **Fun**: You want players to enjoy your game.

• **Lusory attitude**: You want players to take part in the fantasy of your game.

• **Flow**: You want players to be optimally challenged.

• **Structured conflict**: You want to give players a way to combat others or challenge your game systems.

• **Empowerment**: You want players to feel powerful both in the game and in the metagame.

• **Interest / attention / involvement**: You want the player to be engaged by your game.

• **Meaningful decisions**: You want players' choices to have meaning to them and the game.

• **Experiential understanding**: You want the player to gain understanding through play.

Now let's explore each in detail.

Designer-Centric Goals

As a game designer and developer, there are some goals for your life that you hope the games you make might help you achieve.

Fortune

My friend John “Chow” Chowanec has been in the game industry for years. The first time I met him, he gave me some advice about making money in the game industry. He said “You can literally make hundreds of... dollars in the game industry.”

As he hinted through his joke, there are a *lot* of faster, better ways to make money than the game industry. I tell my programming students that if they want to make money, they should go work for a bank; banks have lots of money and are very interested in paying someone to help them keep it. However, the game industry is just like every other entertainment industry job: There are fewer jobs available than people who want them, and people generally enjoy doing the work; so, game companies can pay less than other companies for the same kind of employees. There are certainly people in the game industry who make a lot of money, but they are few and far between.

It is absolutely possible—particularly if you’re a single person without kids—to make a decent living working in the game industry. This is especially true if you’re working for a larger game company where they tend to have good salaries and benefits. Smaller companies (or starting your own small company) are generally a lot riskier and usually pay worse, but you may have a chance to earn a percentage ownership in the company, which could have a small chance of eventually paying out very nicely.

Fame

I’ll be honest: Very, very few people become famous for game design. Becoming a game designer because you want to be famous is a little like becoming a special effects artist in film because you want to be famous. Usually with games, even if millions of people see your work, very few will know who you are.

Of course, there are some famous names like Sid Meier, Will Wright, and John Romero, but all of those people have been making games for years and have been famous for it for equally long. There are also some newer people whom you might know like Jenova Chen, Jonathan Blow, and Markus “Notch” Persson, but even then, many more people are familiar with their games (*Flow/Flower/Journey*, *Braid/The Witness*, and *Minecraft* respectively) than with them.

However, what I personally find to be far better than fame is community, and the game industry has that in spades. The game industry is smaller than anyone on the outside would ever expect, and it’s a great community. In particular, I have always been impressed by the acceptance and openness of the independent game community and the IndieCade game conference.

Community

There are, of course, many different communities within the game industry, but on the whole, I have found it to be a pretty fantastic place filled with great people. Many of my closest friends are people whom I met through working in the game industry or in games education. Though a sad number of high-budget, AAA games appear sexist and violent, in my experience, most of the people working on games are genuinely good people. There is also a large and vibrant community of developers, designers, and artists who are working to make games that are more progressive and created from more varied perspectives. Over the past several years of the IndieCade independent game conference, there have been very well attended panels on diversity in both the games we make and the development teams who are making those games. The independent game community in particular is a meritocracy; if you make great work, you will be welcomed and respected by the indie community regardless of race, gender, sexual orientation, religion, or creed. There is certainly still room to improve the openness of the game development community, and there are always some jerks in any group, but the game development community is full of people who want to make it a welcoming place for everyone.

Personal Expression and Communication

This goal is the flip side of the player-centric goal of experiential understanding. However, personal expression and communication can take many more forms than experiential understanding (which is the exclusive domain of interactive media). Designers and artists have been expressing themselves in all forms of media for hundreds of years. If you have something that you wish to express, there are two important questions to ask yourself:

What form of media could best express this concept?

What forms of media are you adept at using?

Somewhere between these two questions you'll find the answer of whether an interactive piece will be the best way for you to express yourself. The good news is that there is a very eager audience seeking new personal expressions in the interactive realm. Very personal interactive pieces like *Papo y Yo*, *Mainichi*, and *That Dragon, Cancer* have received a lot of attention and critical acclaim recently, signaling the growing maturity of interactive experiences as a conduit for personal expression.[1]

[1] *That Dragon, Cancer* (2014, by Ryan Green and Josh Larson) relates the experience of a couple learning that their young son has terminal cancer and helped Ryan deal with his own son's cancer. *Mainichi* (2013, by Mattie Brice) was designed to express to a friend of hers what it was like to be a transgender woman living in San Francisco. *Papo y Yo* (2014, by Minority Media) places the player in the dream world of a boy trying to protect himself and his sister from a sometimes-helpful, sometimes-violent monster that represents his alcoholic father.

Greater Good

A number of people make games because they want to make the world a better place. These games are often called *serious games* or

games for change and are the subject of several developer's conferences, including the Meaningful Play conference at Michigan State University. This genre of games can also be a great way for a small studio to get off the ground and do some good in the world; there are a number of government agencies, companies, and nonprofit organizations who offer grants and contracts for developers interested in making serious games.

There are many names used to describe games for the greater good. Three of the biggest are:

• **Serious games**: This is one of the oldest and most general names for games of this type. These games can of course still be fun; the "serious" moniker is just to note that there is a purpose behind the game that is more than just playful. One common example of this category is educational games.

• **Games for social change**: This category of games for good is typically used to encompass games that are meant to influence people or change their minds about a topic. Games about things like global warming, government budget deficits, or the virtues or vices of various political candidates would fall into this category.

• **Games for behavioral change**: The intent of these games is not to change the mind or opinion of the player (as in games for social change) but instead to change a player's behavior outside of the game. For example, many medicinal games have been created to discourage childhood obesity, improve attention spans, combat depression, and detect things like childhood amblyopia. There is a large and growing amount of research out there demonstrating that games and game play can have significant effects (both positive and negative) on mental and physical health.

Becoming a Better Designer

The number one thing you can do to become a great game designer is make games... or more accurately, make *a lot* of games. The

purpose of this book is to help you get started doing this, and it's one of the reasons that the tutorials at the end of the book cover several different games rather than having just one monolithic tutorial that meanders through various game development topics. Each tutorial is focused on making a prototype for a specific kind of game and covering a few specific topics, and the prototypes you make are meant to serve not only as learning tools but also as foundations upon which you can build your own games in the future.

Player-Centric Goals

As a game designer and developer, there are some goals for your game that are centered on the effects that you want the game to have on your player.

Fun

Many people regard fun as the only goal of games, although as a reader of this book, you should know by now that this is not true. As discussed later in this chapter, players are willing to play something that isn't fun as long as it grabs and holds their attention in some way. This is true with all forms of art; I am glad to have watched the movies *Schindler's List, Life is Beautiful,* and *What Dreams May Come,* but none of them were at all "fun" to watch. Even though it is not the only goal of games, the elusive concept of fun is still critically important to game designers.

In his book *Game Design Theory,* Keith Burgun proposes three aspects that make a game fun. According to him, it must be enjoyable, engaging, and fulfilling:

• **Enjoyable**: There are many ways for something to be enjoyable, and enjoyment in one form or another is what most players are seeking when they approach a game. In his 1958 book *Les Jeux et Les Hommes,*[2] Roger Caillois identified four different kinds of play:

[2] Roger Caillois, *Le Jeux et Les Hommes (Man, Play, & Games)* (Paris: Gallimard, 1958).

• **Agon**: Competitive play (e.g., chess, baseball, the *Uncharted* series)

• **Alea**: Chance-based play (e.g., gambling and rock, paper, scissors)

• **Ilinx**: Vertiginous play (e.g., roller coasters, children spinning around until they're dizzy, and other play that makes the player feel vertigo)

• **Mimicry**: Play centered on make-believe and simulation (e.g., playing house, playing with action figures).

Each of these kinds of play are enjoyable in their own way, though in all of them, that fun depends on having a *lusory attitude* (i.e., an attitude of play, as discussed in the next section). As Chris Bateman points out in his book *Imaginary Games,* a fine line exists between excitement and fear in games of ilinx, the only difference being the lusory attitude of the player.[3] The *Tower of Terror* attraction at Disney World is a fun simulation of an out of control elevator, but actually being in an out of control elevator is not fun at all.

[3] Chris Bateman, *Imaginary Games*. (Washington, USA: Zero Books, 2011), 26-28.

• **Engaging**: The game must grab and hold the player's attention. In his 2012 talk, "Attention, Not Immersion," at the Game Developers Conference in San Francisco, Richard Lemarchand, co-lead game designer of the *Uncharted* series of games, referred to this as "attention," and it's a very important aspect of game design. I discuss his talk in greater detail later in this chapter.

• **Fulfilling**: Playing the game must fill some need or desire of the player. As humans, we have many needs that can be met through play in both real and virtual ways. The need for socialization and community, for instance, can be met both through playing a board

game with friends or experiencing the day-to-day life of *Animal Crossing* with the virtual friends who live in your town. The feeling of *fiero* (the Italian word for personal triumph over adversity)[4] can be achieved by helping your team win a soccer match, defeating a friend in fighting game like *Tekken*,[5] or by eventually defeating the final level in a difficult rhythm game like *Osu! Tatake! Ouendan*. Different players have different needs, and the same player can have drastically different needs from day to day.

[4] Nicole Lazzaro discusses *fiero* often in her talks at GDC about emotions that drive players.

[5] Thanks to my good friends Donald McCaskill and Mike Wabschall for introducing me to the beautiful intricacies of Tekken 3 and for the thousands of matches we've played together.

Lusory Attitude

In *The Grasshopper,* Bernard Suits talks at length about the *lusory attitude*: the attitude one must have to take part in a game. When in the lusory attitude, players happily follow the rules of the game for the joy of eventually winning via the rules (and not by avoiding them). As Suits points out, neither cheaters nor spoilsports have a lusory attitude; cheaters want to win but not to follow the rules, and spoilsports may or may not follow the rules but have no interest in winning the game.

As a designer, you should work toward games that encourage players to maintain this lusory attitude. In large part, I believe that this means you must show respect for your players and not take advantage of them. In 2008, my colleague Bryan Cash and I gave two talks at the Game Developers Conference about what we termed *sporadic-play* games,[6] games that the player plays sporadically throughout her day. Both talks were based on our experience designing *Skyrates*[7] (pronounced like pirates), a graduate school project for which our team won some design awards

in 2008. In designing *Skyrates* we sought to make a persistent online game (like the massively multiplayer online games [MMOs] of the time; e.g., Blizzard's *World of Warcraft*) that could easily be played by busy people. *Skyrates* set players in the role of privateers of the skies, flying from skyland (floating island) to skyland trading goods and battling pirates. The sporadic aspect of the game was that each player was able to check in for a few minutes at a time throughout her day, set orders for her skyrate character, fight a few pirate battles, upgrade her ship or character, and then let her skyrate play-out the orders while the player herself went about her day. At various times during the day, she might receive a text message on her phone letting her know that her skyrate was under attack, but it was her choice whether to jump into combat or to let her skyrate handle it on his own.

[6] Cash, Bryan and Gibson, Jeremy. "Sporadic Games: The History and Future of Games for Busy People" (presented as part of the Social Games Summit at the Game Developers Conference, San Francisco, CA, 2010). Cash, Bryan and Gibson, Jeremy "Sporadic Play Update: The Latest Developments in Games for Busy People" (presented at the Game Developers Conference Online, Austin, TX, 2010).

[7] *Skyrates* was developed over the course of two semesters in 2006 while we were all graduate students at Carnegie Mellon University's Entertainment Technology Center. The developers were Howard Braham, Bryan Cash, Jeremy Gibson (Bond), Chuck Hoover, Henry Clay Reister, Seth Shain, and Sam Spiro, with character art by Chris Daniel. Our faculty advisors were Jesse Schell and Dr. Drew Davidson. After *Skyrates* was released, we added the developers Phil Light and Jason Buckner. You can still play the game at http://skyrates.net.

As designers in the industry at the time, we were witnessing the rise of social media games like *FarmVille* and the like that seemed to have little or no respect for their players' time. It was commonplace for games on social networks to demand (through their mechanics)

that players log in to the game continually throughout the day, and players were punished for not returning to the game on time. This was accomplished through a few nefarious mechanics, the chief of which were *energy* and *spoilage*.

In social network games with energy as a resource, the player's energy level built slowly over time regardless of whether she was playing or not, but there was a cap on the energy that could be earned by waiting, and that cap was often considerably less than the amount that could be accrued in a day and less than the amount needed to accomplish the optimal player actions each day. The result was that players were required to log in several times throughout the day to spend the energy that had accrued and not waste potential accrual time on capped-out energy. Of course, players were also able to purchase additional energy that was not capped and did not expire, and this drove a large amount of the sales in these games.

The spoilage mechanic is best explained through *FarmVille*, in which players could plant crops and were required to harvest them later. However, if a crop was left unharvested for too long, it would spoil, and the player would lose her investment in both the seeds and the time spent to grow and nurture the crop. For higher-value crops, the delay before spoilage was drastically less than that of low-value, beginner-level crops, so habitual players found themselves required to return to the game within increasingly small windows of time to get the most out of their investments.

Bryan and I hoped through our GDC talks to counter these trends or at least offer some alternatives. The idea of a sporadic-play game is to give the player the most agency (ability to make choices) in the least amount of time. Our professor, Jesse Schell, once commented that *Skyrates* was like a friend who reminded him to take a break from work every once in a while, but after several minutes of play also reminded him to get back to work. This kind of respect caused our game to have a conversion rate of over 90%, meaning that in

2007, over 90% of the players who initially tried the game became regular players.

The Magic Circle

As was mentioned briefly in Chapter 2, "Game Analysis Frameworks," in his 1938 book *Homo Ludens,* Johan Huizinga proposed an idea that has come to be known as the *magic circle*. The magic circle is the space in which a game takes place, and it can be mental, physical, or some combination of the two. Within the magic circle, the rules hold sway over the players, and the amount that certain actions are encouraged or discouraged is different from the world of everyday life.

For example, when two friends are playing poker against each other, they will often bluff (or lie) about the cards that they have and how certain they are that they will win the pot. However, outside of the game, these same friends would consider lying to each other to be a violation of their friendship. Similarly, on the ice in the game of hockey, players routinely shove and slam into each other (within specific rules, of course), however these players will still shake hands and sometimes be close friends outside of the boundaries of the game.

As Ian Bogost and many other game theorists have pointed out, the magic circle is a porous and temporary thing. Even children recognize this and will sometimes call "time out" during make-believe play. Time out in this sense denotes a suspension of the rules and a temporary cessation of the magic circle, which is often done so that the players can discuss how the rules should be shaped for the remainder of the game. Once the discussion is complete, "time in" is called, and both play and the magic circle continue where they left off.

Though it is possible to pause and resume the magic circle, it is sometimes difficult to maintain the integrity of the magic circle through those pauses. During long delays of football games (for

example, if the game is delayed 30 minutes for weather in the middle of the second quarter), commentators will often comment on how difficult it is for players to either maintain the game mindset through the delay or get back into the game mindset once play resumes.

Flow

As described by psychologist Mihaly Csíkszentmihályi (pronounced chick-sent-me-high), *flow* is the state of optimal challenge, and it has been discussed frequently at the Game Developers Conference because it relates so closely to what many game designers are trying to create. In a flow state, a player is focused intently on the challenge before her and very often loses awareness of things that are outside of the flow experience. You have probably felt this at times when you have played or worked so intently on something that time seems distorted, either passing faster or more slowly than normal.

Flow in this sense was the subject of Jenova Chen's MFA thesis paper at the University of Southern California as well as the subject of his thesis game, appropriately titled *Flow*.[8] Jenova also spoke about this concept in a couple of talks at GDC.

[8] The original Flash-based version of *Flow* can be played at http://interactive.usc.edu/projects/cloud/flowing/. The updated and expanded PlayStation 3 version can be downloaded from the PlayStation Store.

As you can see in Figure 8.1, the flow state exists between boredom and frustration. If the game is too challenging for the player's skill level, she will feel frustrated; conversely, if the player is too skilled for the game, she will feel bored.

Figure 8.1. Flow as described by Csíkszentmihályi



According to the 2002 article, “The Concept of Flow,” by Jeanne Nakamura and Mihaly Csíkszentmihályi, the experience of flow is the same across cultures, genders, ages, and various kinds of activity, and it relies on two conditions:[9]

[9] Jeanne Nakamura and Mihaly Csíkszentmihályi, “The Concept of Flow.” *Handbook of positive psychology* (2002): 89–105, 90.

• Perceived challenges, or opportunities for action, that stretch (neither overmatching nor underutilizing) existing skills; a sense that one is engaging challenges at a level appropriate to one’s capacities

• Clear proximal goals and immediate feedback about the progress that is being made

This is what much of the discussion of flow has centered on in the realm of game design. Both of these conditions are concrete enough for designers to understand how to implement them in their games, and through careful testing and player interviews, it’s easy to measure whether your game is doing so. However, since 1990, when Csíkszentmihályi published his book *Flow: The Psychology of Optimal Experience,* research has expanded our understanding of flow as it relates to games in one very important way: Designers now realize that flow is difficult to maintain. It turns out that while players enjoy flow—and moments of flow are some of the most memorable of your games—it is tiring to maintain flow for more than 15 or 20 minutes. In addition, if the player is always kept in a perfect state of flow, she may never have an opportunity to realize that her skill is improving. So, for many players, you actually want a flow diagram like the one shown in Figure 8.2.

Figure 8.2. Updated flow.



There is a border between flow and boredom where the player feels powerful and skillful (i.e., they feel awesome!), and players actually need that. While the flow state is powerful and successful, it’s also

important to let your players out of the flow state so that they can reflect on what they accomplished while within flow. Think about the best boss fight you've ever had in a game. When in a flow state, by definition, you lose track of everything outside of the moment because flow requires total attention. If you are like me, it wasn't until you had actually defeated the boss that you had a moment to breathe and realize how amazing the fight had been. Players need not only these moments but also moments to revel in their increased skill.

Like many other games, the original *God of War* game did this very well. It would consistently introduce the player to a single opponent of a new type, and this often felt like a mini boss fight because the player hadn't yet figured out the strategies for defeating that type of enemy. The player eventually learned the strategy for that particular enemy and over several encounters with single enemies of this type perfected her skill. Then, several minutes later, the player was required to fight more than one of this enemy type simultaneously, though because she had increased in skill, this was actually less of a challenge than the single opponent had been originally. Her ability to easily dispatch several copies of the enemy that had given her trouble singly demonstrated to her that she had increased in skill and made her feel awesome.

As you design your games, remember that it's not just about giving the player an optimal challenge, it's also about giving her the understanding that she is getting better and granting her time to just be awesome. After a difficult fight, give the player some time to just be powerful. This encourages feelings of empowerment.

Structured Conflict

As you saw in Chapter 1, "Thinking Like a Designer," structured conflict is one of the human needs that can be fulfilled by games. One of the primary differences between *play* and *game* is that game always involves struggle or conflict, which can be conflict against other players or conflict against the systems of the game (see the

section on player relationships in Chapter 4, "The Inscribed Layer"). This conflict gives players a chance to test their skill (or that of their team) against others, against systems, against chance, or against themselves.

This desire for structured conflict is also evident in the play of animals. As Chris Bateman points out in *Imaginary Games:*

When our puppy plays with other dogs, there are clear limits as to what is acceptable behavior. When play fighting with another puppy, there is much gentle biting, climbing upon one another and general rolling around in frenzied mock violence; there are rules of a kind here.[10]

[10] Christ Bateman, *Imaginary Games*. (Washington, USA: Zero Books, 2011), 24.

Even in some actual wars, there have been game-like rules. In the memoir of his life, Chief Plenty-Coups of the Native American Crow tribe relates some of the rules of counting coup in battle. Coup was counted for getting away with dangerous actions on the battlefield. Striking an armed and able enemy warrior with a coup-stick, quirt (short riding whip), or bow before otherwise harming him; stealing an enemy's weapons while he was still alive; stealing horses or weapons from an enemy camp; and striking the first enemy to fall in battle (before he was killed) all counted for coup. Doing so while avoiding injury to oneself counted more. Plenty-coups also spoke of rules regarding the two symbolic sticks of tribal fraternities.

One of these sticks in each society was straight and bore one eagle's feather on its smaller end. If in battle its carrier stuck this stick into the ground, he must not retreat or leave the stick. He must drop his robe [die] there unless relieved by a brother member of his society riding between him and the enemy. He might then move the stick with honor, but while it was sticking in the ground it represented the Crow country. The bearers of the crooked sticks, each having two feathers, might at their discretion move them to better stands

after sticking them to mark a position. But they must die in losing them to the enemy. By striking coup with any of these society coup-sticks, the bearers counted double, two for one, since their lives were in greater danger while carrying them.[11]

[11] Frank Bird Linderman, *Plenty-coups, Chief of the Crows*, New ed. (Lincoln, NE: University of Nebraska Press, 2002), 31–32.

After the battle, coup was counted, as each warrior related the tales of his exploits during the battle. For successfully performing a coup and escaping without being harmed, the warrior would receive an eagle feather that could be worn in the hair or attached to a coup-stick. If he had been injured, the feather was painted red.

The activity of counting coup among the Native American tribes of the plains lent additional meaning to the wars between nations and provided a structured way for acts of bravery on the battlefield to translate into increased respect once the battle was complete.

Many of today's most popular games provide for structured conflict between teams of players, including all traditional team sports (soccer, football, basketball, and hockey being the most popular worldwide) as well as online team competitions like *League of Legends*, *Team Fortress 2*, and *Overwatch*. But even without teams, games as a whole provide ways for players to engage in conflict and triumph over adversity.

Empowerment

The earlier section on flow covered one kind of empowerment (giving the player the feeling that she is powerful in the game world). This section covers another kind of empowerment: giving the player power over what she chooses to do in the game. I mean this in two senses: autotelic and performative.

Autotelic

The term *autotelic* comes from the Latin words for self (auto) and goal (telos). A person is autotelic when she is determining her own goals for herself. When Csíkszentmihályi initially started developing his theory of flow, he knew that autotelisis would have a major role in it. According to his research, autotelic individuals get the most pleasure out of flow situations, whereas nonautotelic individuals (that is, those who don't enjoy setting their own goals) tend to get more pleasure out of easy situations where they perceive their skill level to be much higher than the difficulty level of the challenge.[12] Csíkszentmihályi believes that it is an autotelic personality that enables a person to find happiness in life regardless of situation.[13]

[12] Nakamura and Csíkszentmihályi, "The Concept of Flow," 98.

[13] Mihaly Csíkszentmihályi , *Flow: The Psychology of Optimal Experience (*New York: Harper & Row, 1990), 69.

So, what kinds of games encourage autotelic behavior? One fantastic example is *Minecraft*. In this game, the player is dropped into a randomly generated world where her only real goal is survival. (Zombies and other monsters will attack the player at night.) However, she is also given the ability to mine the environment for resources and then use those resources to make both tools and structures. Players of *Minecraft* have not only built castles, bridges, and a full-scale model of the Star Trek Enterprise NCC-1701D but also roller coasters that run for many kilometers and even simple working computers with RAM.[14] This is the true genius of *Minecraft*: it gives players the opportunity to choose their own path as players and provides them with flexible game systems that enable that choice.

[14] http://www.escapistmagazine.com/news/view/109385-Computer-Built-in-Minecraft-Has-RAM-Performs-Division.

While most games are less flexible than *Minecraft*, it is still possible to allow the player multiple ways to approach a problem. One of the reasons for the loss in popularity of both text-based adventures (e.g.,

Zork, *Planetfall*, and *The Hitchhiker's Guide to the Galaxy* by Infocom) and the point-and-click adventure games that followed them (e.g., the *King's Quest* and *Space Quest* series by Sierra OnLine) is that they often only allowed a single (often obtuse) approach to most problems. In *Space Quest II*, if you didn't grab a jockstrap from a random locker at the very beginning of the game, you couldn't use it as a sling much later in the game, and you would have to restart the game from the beginning. In Infocom's game version of *The Hitchhiker's Guide to the Galaxy* when a bulldozer approached your house, you had to lie down in the mud in front of it and then "wait" three times. If you didn't do this *exactly,* you would die and have to restart the game.[15] Contrast this with more modern games like *Dishonored*, where nearly every problem has at least one violent and one nonviolent solution. Giving the player choice over how she will accomplish her goals builds player interest in the game and player ownership over successes.[16]

[15] One of the major reasons that this was done was because of the multiplicative explosions of content that would occur if the player were allowed to do anything in the game narrative. The closest thing that I have seen to a truly open, branching narrative is the interactive drama *Façade* by Michael Mateas and Andrew Stern.

[16] However, you must also keep development cost and time in perspective. If you're not careful, every option that you give your player could increase the cost of development, both in terms of monetary cost and in terms of time. It's a careful balance that you must maintain as a designer and developer.

Performative

The other kind of empowerment that is important to games is performative empowerment. In *Game Design Theory,* Keith Burgun states that not only are game designers creating art, they're creating the ability for players to make art. The creators of passive media can be thought of as composers; they create something to be consumed by the audience. But, as a game designer, you're actually

somewhere between a composer and an instrument maker. Instead of just creating the notes that others will play, you're also creating the instrument that they can use to make art. One excellent example of this kind of game is *Tony Hawk's Pro Skater*, where the player has a large vocabulary of moves to draw from and must choose how to string them together in harmony with the environment to get a high score. Just as the cellist Yo-yo Ma is an artist, a game player can be an artist when empowered by a game designer who crafts a game for her that she can play artistically. This can also be seen in other games with large vocabularies of moves or strategies like fighting and real-time strategy games.

Attention and Involvement

As mentioned earlier in this chapter, the fantastic game designer Richard Lemarchand spoke at GDC about attention in his 2012 talk, "Attention, Not Immersion: Making Your Games Better with Psychology and Playtesting, the Uncharted Way." The purpose of his talk was to expose confusion about the use of the word *immersion* in game design, and to demonstrate that talking about getting and holding an audience's *attention* was a much clearer way of describing what game designers usually seek to do.

Prior to Lemarchand's talk, many designers sought to increase immersion in their games. This led to things like the reduction or removal of the HUD (heads-up onscreen display) and the minimization of elements that could pull the player out of the experience of the game. But as Lemarchand pointed out in his talk, gamers never truly achieve immersion, nor would they want to. If a gamer actually believed that he was in Nathan Drake's position halfway through *Uncharted 3*, being shot at while clinging to a cargo net that was hanging out of the open door of a transport plane thousands of feet above a desert, the player would be absolutely terrified! One of the critical aspects of the magic circle is that both entry into the circle and remaining in the circle are choices made by the player, and she is always aware that the game is voluntary. (As

Suits points out, once participation is no longer voluntary, the experience is no longer a game.)

Instead of immersion, Lemarchand seeks to initially gain the player's attention and then to maintain hold over it. For the sake of clarity, I will use *attention* to describe immediate interest that can be grabbed and *involvement* to describe long-term interest that needs to be held (a distinction that Lemarchand now uses as well). Lemarchand also differentiates between reflexive attention (the involuntary response that we have to stimuli around us) and executive attention (which occurs when we choose to pay attention to something).

According to his talk, the elements of beauty, aesthetics, and contrast are great at grabbing attention. James Bond films always open with an action scene for this very reason. They begin *in medias res* (in the middle of things) because doing so creates a marked contrast between the boredom of sitting in the theater waiting for the film to start and the excitement of the beginning of the film. This kind of attention grab exploits reflexive attention, the attention shift that is evolutionarily hard-wired into you. When you see something moving out of the corner of your eye, it will grab your attention regardless of whether you wish it to or not. Then, once the Bond movie has your attention, it switches to the rather tedious exposition required to set up the rest of the film. Because the viewer is already hooked by the film, she will choose to use executive attention (that is, attention by choice) to listen to this exposition.

In *The Art of Game Design,* Jesse Schell presents his theory of the *interest curve*. The interest curve is also about grabbing attention, and according to Schell, a good interest curve looks like Figure 8.3.

Figure 8.3. Interest curve from Jesse Schell's book



According to Schell, in a good interest curve, the audience will enter with a little interest (A), and then you want to grab them with a "hook" that piques their interest (B). After you have them interested, you can

drop it back down and steadily build interest with little peaks and valleys (C E and D F respectively) that should slowly build to the highest point of interest: the climax (G). After the climax, the audience's interest is let back down to (H) in a denouement as the experience comes to a close. This is actually very similar to Syd Field's standard three-act dramatic curve diagram (described in Chapter 4, "The Inscribed Layer"), and it has been shown to work well for time spans between a few minutes and a couple of hours. Schell tells us that this interest curve can be repeated in fractal fashion to cover longer periods of time. One way this could be accomplished is by having a mission structure within a larger game and making sure that each mission has its own good interest curve within the larger interest curve of the entire game. However, it's more complex than that because the interest that Schell discusses is what I'm calling attention, and we still need to account for involvement if we want to interest the player for long periods of time.

Taking a closer look at attention and involvement, attention is directly paired with reflexive attention (the involuntary response), while involvement is almost exclusively voluntary/executive attention. Having thought about this for a while, I've created the diagram shown in Figure 8.4 as a synthesis of Lemarchand's concepts and my personal experience as both a designer and player.

Figure 8.4. The four elements in relation to attention and involvement (because technology is largely invisible to the player it doesn't register much on this graph)



As you can see in the diagram, aesthetics (in terms of the aesthetic element in the tetrad) are best at grabbing our attention, and in the case of aesthetics, that attention is largely reflexive. This is because aesthetics deal directly with our senses and call for attention.

Narrative and mechanics both require executive attention. As pointed out by Lemarchand, narrative has a greater ability to grab our attention, but I disagree with Lemarchand and Jason Rohrer

when they state that mechanics have a greater ability to sustain involvement than narrative. While a single movie tends to last only a couple of hours, that is also relatively true of the mechanics in a single session of play. And, in my personal experience, I have found that just as great mechanics can hold my involvement for over 100 hours, so can a series of narratives hold my attention through over 100 episodes of a serial television show. The major difference between mechanics and narrative here is that narrative must be ever evolving while gameplay mechanics can exist unchanged for years and still hold interest due to the different circumstances of play (for example, consider a player's lifelong devotion to chess or go).

The one thing that I have seen outlast both narrative and mechanics in terms of long-term involvement is community. When people find that a community exists around a game, movie, or activity, and they feel part of that community, they will continue to take part long after the hold of narrative or mechanics have lost their sway. Community is what kept many guilds together in *Ultima Online* long after most people had moved on to other games. And when the members of the community did eventually move on, they more often than not chose as a community which new game to play together, thus continuing the same community through multiple different online games.

Interesting Decisions

As you read in Chapter 1, Sid Meier has stated that games are (or should be) a series of interesting decisions, but we questioned at that time what exactly was meant by *interesting.* Throughout the book thus far, we have seen several concepts presented that can help illuminate this.

Katie Salen and Eric Zimmerman's concept of *meaningful play* as presented in Chapter 5, "The Dynamic Layer," gives us some insight into this. To be meaningful, a decision must be both *discernible* and *integrated*:[17]

[17] Katie Salen and Eric Zimmerman, *Rules of Play* (Cambridge, MA: MIT Press, 2003), 34.

• **Discernible**: The player must be able to tell that the game received and understood her decision (i.e., immediate feedback).

• **Integrated**: The player must believe that her decision will have some effect on the long-term outcome of the game (i.e., long-term impact).

In his definition of game, Keith Burgun points out the importance of decisions being *ambiguous*:

• **Ambiguous**: A decision is ambiguous for the player if she can guess at how it might affect the system but can never be sure. The decision to wager money in the stock market is ambiguous. As a savvy investor, you should have a pretty decent guess about whether the value of the stock will go up or down, but the market is so volatile that you can never know for sure.

Almost all interesting decisions are also *double-edged* (as in the saying “a double-edged sword”):

• **Double-edged**: A decision is double-edged when it has both an upside and a downside. In the previous stock purchase example, the upside is the longer-term potential to make money, and the downside is the immediate loss of the resource (money) used to purchase the stock as well as the potential for the stock to lose value.

Another aspect involved in making a decision interesting is the *novelty* of the decision.

• **Novel**: A decision is novel if it is sufficiently different from other decisions that the player has made recently. In the classic Japanese roleplaying game (JRPG) *Final Fantasy VII*, combat with a specific enemy changes little once the encounter has begun, meaning that there are few novel decisions for the player to make. If the enemy is weak to fire, and the player has enough mana and fire magic, she

will generally attack every round with fire magic until the enemy is defeated. In contrast, the excellent combat in the JRPG *Grandia III* makes positioning and location important for most special attacks, but the player's characters move around the field autonomously (independent of player input). Whenever the player is able to make a decision, time freezes for her, and she must reevaluate the positions of allies and enemies before making each decision. This autonomous movement of her characters and the importance of position make every combat decision novel.

The final requirement for interesting decisions is that they must be *clear*.

• **Clear**: Although it is important for the outcomes of a choice to have some ambiguity, the choice itself must be clear, and there are many ways that choices can lack clarity:

• A choice can be unclear if there are too many options to choose from at a given time; the player can have difficulty discerning the differences between them. This leads to *choice paralysis*, the inability to choose because there are too many options.

• A choice can be unclear if the player can't intuit the likely outcome of the choice. This was often a problem in the dialog trees in some older games, which for years just listed the possible statements that a player could make without any information about the implied meaning of those statements. In contrast, the dialog tree decision wheel in *Mass Effect* included information about both whether a statement would be said in a friendly or antagonistic way and whether it would extend or shorten the conversation. This allowed the player to choose an attitude rather than specific wording of a statement and removed the ambiguity the dialog tree.

• A choice can also be unclear if the player doesn't understand the significance of the choice. One of the great advances in the combat system of *Grandia III* over *Grandia II* allowed threatened characters to automatically call for help during another character's turn. If Character A is about to be hit by an attack, and Character B can

prevent it by acting on this turn, Character A will cry for help during Character B's turn. The player may still choose to have Character B do something other than prevent the attack, but the game has made it clear to her that this is her last chance to prevent the attack on A.

These six aspects can all be combined together into a decent understanding of the things that make a decision interesting. An interesting decision is one that is discernible, integrated, ambiguous, double-edged, novel, and clear. By making your decisions more interesting, you can increase the appeal of your mechanics and thereby the player's long term involvement in your game.

Experiential Understanding

The final goal for players that we'll discuss in this chapter is *experiential understanding*, a design goal that is far more accessible to game designers than designers of any other kind of media.

In 2013, game critic and theorist Mattie Brice released *Mainichi*, the first game that she had designed and developed herself.

Figure 8.5. *Mainichi* by Mattie Brice (2013)



As described by Brice, *Mainichi* is a personal letter from her to a friend to help her friend understand what her daily life is like. In her real life, Brice is a transgender woman living in the Castro district of San Francisco. In *Mainichi*, the player takes on the role of Mattie Brice and must choose what to do to prepare to go out for coffee with a friend: does she dress nicely, put on makeup, eat a bite? Each of these decisions change how some (but not all) of the people around town react to her as she walks to the coffee shop and orders her drink. Even a simple decision like whether to pay with a credit card or cash has meaning in the game. (Paying with a credit card will cause the barista to refer to you as "Ms... er... Mr. Brice" because he reads Brice's old, male name on the credit card.)

The game is very short, and as a player, you are compelled to try again and see what happens differently based on the seemingly small choices that you make throughout the game. Because the player's decisions change how the character of Mattie is perceived, you feel *complicit* in her being treated well or poorly by the people around her. Though some kind of branching chart or a story structured like the movie *Groundhog Day* (in which Bill Murray's character must relive the same day hundreds of times until he finally gets it right) could convey the same information about the large implications of the tiny choices that Brice makes every day, neither would convey a sense of responsibility to the audience. At this time, it is only through a game (be it a video game, make-believe, or roleplaying) that a person can actually walk in the shoes of another and gain insight into what it must be like to make the decisions that she makes. This experiential understanding is one of the most interesting goals that we can seek to achieve as game designers.

Summary

Everyone making games has different feelings about each of these reasons. Some people just want to make fun experiences, some people want to give players interesting puzzles, some people want to encourage players to think deeply about a specific topic, and some people want to give players an arena in which to be empowered. Regardless of what your reasons are for wanting to make a game, it is time now to start making them.

The next two chapters are about paper prototyping and playtesting. Together, prototyping and playtesting form the core of the real work of game design. In almost any game—especially a digital game—there will be hundreds of small variables that you can tweak to change the experience. However, in digital games, even seemingly small changes can take considerable development time to implement. The paper prototyping strategies presented in the next chapter can help you get from concept to playable (paper) prototype very quickly and then get you from one prototype to the next even more rapidly. For many games, this paper prototyping phase can

save you a lot of time in digital development because you will have already run several paper playtests to find the fun before writing a single line of code.

Chapter 10. Game Testing

Inherent in the concepts of prototyping and iteration is an understanding that high-quality testing is absolutely necessary to good game design. But the question becomes how exactly should this testing be performed?

In this chapter, you learn about various methods of playtesting for games, how to implement them properly, and at what stage in development each method is appropriate.

Why Playtest?

Once you've analyzed your goals, designed a solution, and implemented a prototype, it's time to test that prototype and get some feedback on your design. I understand that this can be a frightening proposition. Games are difficult to design, and it's going to take a lot of experience for you to get good at it. Even when you become a great designer, you'll still probably have some trepidation when you think about people playing your game for the first time. That's okay. The number one thing to keep in mind is that every person who plays your game is making it better; every comment you get, whether positive or negative, can help steer you in a direction that will improve player experience and hone your design.

Refining the design is what it's all about, and you absolutely must have external feedback to do so. I've served as a judge for several game design festivals over the years, and it always amazes me how easy it is to tell whether a dev team has done sufficient playtesting. For example, without enough playtesting the goals of the game are often not clearly specified, and the difficulty of the game often ramps up very quickly. These are both common indications that the game was most often played by people who already knew how to play and knew how to get through the difficult parts, so they couldn't see the

ambiguity or the rise in difficulty the way that a naïve tester would have.

This chapter will give you the knowledge and skills to run meaningful playtests and get the information from them that you need to make your games better.

Note

Investigators Versus Playtesters Oftentimes in the game industry, we refer to both the people running the playtests and the participants in those tests as “playtesters”. For clarity, in this book, I will use these terms as follows:

• **Investigator:** A person administering a playtest, usually someone on your team

• **Playtester:** A person taking part in the playtest by playing games and giving feedback

---

Being a Great Playtester Yourself

Before getting into how to run various types of playtests for your games or what to look for in playtesters, let’s examine how you can be a great playtester for other people.

• **Think out loud:** One of the best things you can do as a playtester is to describe your internal thought processes out loud while playing. Doing so will help the investigator running the test to correctly interpret the thoughts behind your actions. This can be especially helpful if it’s the first time that you’ve ever encountered the game.

• **Reveal your biases:** We are all biased by our experiences and tastes, but it’s often difficult for investigators to know where their playtesters are coming from. As you’re playing talk about other

games, films, books, experiences, etc. that the game reminds you of. This will help the investigators understand the background and biases that you bring with you to the playtest.

• **Self-analyze:** Try to help the investigators understand why you're experiencing the reactions that you're having to the game. Instead of just saying something like "I feel happy," it's better to say something more specific like "I feel happy because the jumping mechanic makes me feel powerful and joyful."

• **Separate elements:** As a playtester, once you've given overall feedback on the game experience, try to see each element separately; analyze art, game mechanics, game feel, sound, music, etc. as individual elements. This can be very helpful to investigators and is akin to saying "the cellos sound out of tune" rather than "I didn't like that symphony." As a designer, your insight into games can allow you to give more specific feedback than most players, so take advantage of it.

• **Don't worry if they don't like your ideas:** As a designer, you should tell the investigators any ideas you have to make their game better, but you also shouldn't be at all offended if they don't use them. A lot of game design is about checking your ego at the door; it turns out that playtesting has an element of that too.

The Circles of Playtesters

The game testing you do will go through several expanding circles of playtesters, starting with you and expanding outward through your friends and acquaintances to eventually encompass many people you have never met. Each circle of people can help with different aspects of your playtesting.

The First Circle—You

As a game designer, the first and last playtester of the games you design will most likely be you. You will be the first person to try out

each of your ideas, and you'll be the first person to decide whether the game mechanics and interface feel right.

A central theme of this book is that you always want to get a prototype of your game working as soon as possible. Until you have a working prototype, all you have is a jumble of ideas, but after you have a prototype, you have something concrete to react to.

Later in this book, you'll be making digital game prototypes in Unity. Every time you press the Play button in Unity to run your game, you're acting as a playtester. Even if you're working in a team and are not the primary engineer on the project, as a designer, it will be your job to determine whether the game is heading toward the kind of experience your team wants to create. Your skills as a playtester are most useful in the very early stages of prototyping when you may need a great prototype to help other team members understand the design or when you may be still trying to discover the core mechanic or core experience of the game.

However, you can never get a first impression of your own game; you know too much about it. There is a point at which you need to branch out and show your game to other people. Once you feel that your game is anything better than terrible, it's honestly time to find a few other people and show it to them.

The Second Circle—Trusted Friends

Once you've playtested your game, iterated, made improvements, and actually crafted something that approximates the experience that you want, it's time to show it to others. The first of these should be trusted friends and family members, preferably those either in your target audience or in the game development community. Members of your target audience will give you good feedback from the point of view of your future players, and game developers can help by sharing their considerable insight and experience. Game developers will also often have the ability to overlook aspects of the

game that are obviously unfinished, which can also be very useful for relatively early prototypes.

Tissue Playtesters

Tissue playtester is an industry term to describe playtesters that are brought in to play the game and give their feedback once and are then discarded. They are one-use, like facial tissues. This kind of tester is important because they can give you a naïve reaction to your game. Once anyone has played your game even a single time, they know something about it, and that knowledge biases subsequent playtest sessions. This kind of naïve perspective is critically important when testing:

- The tutorial system
- The first few levels
- The emotional impact of any plot twists or other surprises
- The emotional impact of the end of the game

Everyone Is a Tissue Playtester Only Once

Your game never gets a second chance to make a first impression. When Jenova Chen was working on his most famous game, *Journey*, he and I were housemates. However, he asked me to wait until more than a year into the development of the game before I playtested it. Later, he expressed to me that he specifically wanted my feedback on the level of polish of the game and whether it was achieving its intended emotional arc. As such, it would have ruined the experience for me to have played it in the early stages of development before any of that polish existed. Keep this in mind when playtesting with close friends. Think about the most valuable kinds of feedback that each person can give and make sure to show them the game at the best time for each individual.

That being said, never use that point as an excuse for hiding your game from everyone “until it’s ready.” Hundreds of people playtested *Journey* before I saw it. You will find that in the initial stages of playtesting, most people will tell you the same things in slightly different ways. You need that feedback, and even very early in the development process, you need tissue playtesters to tell you which of your game mechanics are confusing or need work for a variety of reasons. Just save a couple of trusted people for later when you know that their specific feedback will be most useful.

The Third Circle—Acquaintances and Others

After you’ve been iterating on your game for a while and you’ve got something that seems pretty solid, it’s time to take it out into the wild. This isn’t yet the time to post a beta to the Internet and expose your game to the rest of the world, but this is when feedback from others that you don’t normally associate with can be helpful. The people who make up your friends and family often share your background and experiences, meaning that will also often share some of your tastes and biases. If you only test with them, you will get a biased understanding of your game. A corollary to this would be someone in Austin, Texas, being surprised that the state of Texas voted for a Republican presidential candidate. Most people in Austin are liberal, while the rest of the state is primarily conservative. If you only polled people in Austin and didn’t break out of that left-leaning bubble, you’d never know the opinion of the state as a whole. Similarly, you’re going to need to get out of your normal social circles to find more playtesters for your game and to understand a larger audience’s reaction to your game.

So, where do you look for more people to playtest your game? Here are some possibilities:

• **Local universities:** Many college students love playing games. You could try setting up your game in the student center or quad and

showing it to groups of people. Of course, you'll want to check with the campus security before doing so.

You could also look into whether your local university has a game development club or a group that meets for weekly game nights and ask if they would mind you bringing a game for them to playtest.

• **Local game stores / malls:** People head to these places to buy games, so it could be a fantastic place to get some playtest feedback. Each of these places will have different corporate policies on these kinds of things, so you need to talk with them first.

• **Farmers markets / community events / parties:** These kinds of public gatherings of people can have incredibly diverse audiences. I've gotten some great feedback on games from people I met at parties.

The Fourth Circle—The Internet

The Internet can be a scary place. Anonymity ensures that there is little or no accountability for actions or statements, and some people online will be mean just for kicks. However, the Internet also contains the largest circle of playtesters that you can possibly get. If you're developing an online game, you're eventually going to have to reach out to the Internet and see what happens. However, before you do so, you will need to have considerable data and user tracking in place, which you can read about in the "Online Playtesting" section that follows.

Methods of Playtesting

There are several different methods of playtesting, each of which is most appropriate for different phases of your game. The following pages explore various methods of playtesting that I have found to be useful in my design process.

Informal Individual Testing

As an independent developer this is how I tend to do most of my testing. I've been focusing on mobile games lately, so it's easy to carry my device around with me and show my games to people. More often than not, during a break in conversation I'll ask if the person I'm speaking with would mind taking a look at my game. This is, of course, most useful in the early stages of development or when you've got a specific new feature that you want to test. Things to keep in mind during this kind of testing include the following:

• **Don't tell the player too much:** Even in the early stages, it's important to learn whether your interface is intuitive and the goals of your game are clear. Try giving your game to players and watching what they do before they've had any instruction. This can tell you a lot about what interactions your game implies on its own. Eventually, you'll learn the specific short sentences you need to say to people to help them understand your game, and these can form the basis of your in-game tutorial.

• **Don't lead the playtester:** Be sure you don't ask leading questions that may inadvertently bias your player. Even a simple question like "Did you notice the health items?" informs your playtester that health items exist and implies that it is important for her to collect them. Once your game is released, most players won't have you there to explain the game to them, and it's important to let your playtesters struggle a bit to help you learn which aspects of your game are unintuitive.

• **Don't argue or make excuses:** As with everything in design, your ego has no place in a playtest. Listen to the feedback that playtesters are giving you, even (or possibly especially) if you disagree with it. This isn't the time to defend your game; it's the time to learn what you can from the person who is taking time out of her day to help the design improve.

• **Take notes:** Keep a small notebook with you and take notes on the feedback you get, especially if it's not what you expected or wanted to hear. Later, you can collate these notes and look for statements that you heard multiple times. You shouldn't really put too much stock in what is said by a single playtester, but you should definitely pay attention if you hear the same feedback from many different people.

Taking Playtest Notes

Playtest notes can be one of your most valuable tools for understanding and improving your games, however, you need to make sure that you're taking notes in an effective way. Just writing a bunch of notes down randomly or writing notes and never reviewing them isn't going to help you much. Figure 10.1 shows the grid of information that I typically use when writing down playtest notes.

Figure 10.1. Example row of playtest notes



You previously saw the chart below in Chapter 7, "Acting Like a Designer," but now we'll explain each of the columns in more depth.

As mentioned in Chapter 7, it is critical that you gather as much information from each playtest as possible and that you get it in a usable form.

The first three columns (with black headers) are recorded during the playtest. The other three (with green headers) are filled out later as you take time to review the playtest and interpret what you heard.

Formal Group Testing

For many years, this is how playtesting was done at large studios, and when I was working at Electronic Arts, I took part in many of

these playtests for other teams. Several people are brought into a room full of individual stations at which they can play the game. They are given little or no instruction and allowed to play the game for a specific amount of time (usually about 30 minutes). After this time, the playtesters are given a written survey to fill out, and investigators sometimes interview them individually. This is a great way to get feedback from a high volume of people, and it can get you a large number of answers to some important questions.

Some example post-playtest survey questions include:

• “What were your three favorite parts of the game?”

• “What were your three least favorite parts of the game?”

• Provide the playtester with a sequential list of various points in the game (or even better a series of images) and ask them, “How would you describe the way you felt at each of these points in the game?”

• “How do you feel about the main character (or other characters) in the game? / Did your feelings about the main character change over the course of the game?”

• “How much would you pay for this game? / How much would you charge for this game?”

• “What were the three most confusing things about the game?”

Formal group testing is often administered by investigators outside of the core development team, and there are even companies that provide testing services like this.

All Formal Testing Requires a Script

Any time you are doing formal testing, either with investigators from inside or outside the team, you will want to have a script. The script should include the following information:

• What should investigators say to the playtesters to set up the game? What instructions should they give?

• How should investigators react during the playtest? Should they ask questions if they see a playtester do something interesting or unusual? Should they provide any hints to playtesters during the test?

• What should the environment be like for the playtest? How long should the playtester be allowed to play?

• What specific survey questions should be asked of the playtester once the playtest is complete?

• What kinds of notes should the investigator take during the playtests?

Formal Individual Testing

Where formal group testing seeks to gather small bits of information from many different people and grant investigators a gestalt understanding of how playtesters react to a game, formal individual testing seeks to understand the fine details of a single playtester's experience. To accomplish this goal, investigators carefully record the details of a single individual's experience with the game and then review the recordings later to make sure that they haven't missed anything. There are several different data streams that you should record when doing formal individual testing:

• **Record the game screen:** You want to see what the player is seeing.

• **Record the playtester's actions:** You want to see the input attempted by the player. If the game is controlled with mouse and keyboard, place a camera above them. If the game is tested on a touchscreen tablet, you should have a shot of the player's hands touching the screen.

• **Record the playtester's face:** You want to see the player's face so that you can read her emotions.

• **Record audio of what the playtester says:** Even if the player doesn't vocalize her stream of consciousness, hearing utterances she makes can give you more information about her internal thought process.

• **Log game data:** Your game should also be logging time stamped data about its internal state. This can include: input from the player (e.g., button presses on the controller), the player's success or failure at various tasks, the location of the player, time spent in each area of the game, and so on. See the "Automated Data Logging" sidebar later in this chapter for more information.

All of these data streams are later synched to each other so that designers can clearly see the relationships between them. This allows you to see the elation or frustration in a player's face while simultaneously viewing exactly what the player was seeing on-screen at the time and the input their hands were attempting on the controls. Though this is a considerable amount of data, modern technology has actually made it relatively cheap to create a reasonably good lab for individual testing. See the sidebar for more information.

Setting up a lab for formal individual playtesting

You can easily spend tens of thousands of dollars setting up a lab for formal individual testing—and many game studios have—but you can also mock up a pretty decent one for not a lot of money.

For most game platforms, you should be able to capture all of the data streams listed in the chapter with a powerful gaming laptop and just one additional camera: modern graphics cards can record the screen as the game is played, the laptop's web cam can record the player's face, and the one separate camera should be set up to show the player's hands. Recording audio on all streams can help

you to synchronize them later. The game data log should also be time stamped to allow for synchronization.

Synchronizing Data

Many software packages out there enable you to sync several video streams, but often the oldest methods are the easiest, and in this case, you can use a digital version of the *slate* from the early days of sound in film. In a film, the slate is the little clap board that is shown at the beginning of a take. A member of the crew holds the slate, which shows the name of the film, the scene number, and the take number. She reads these three things out loud and then claps the slate together. This later enables the editor to match the visual film frame where the clapper closed with the moment in the audio tape that the sound was made, synching the separate video and audio tracks.

You can do the same thing by making a digital slate part of your game. At the beginning of a playtest session, the game screen can show a digital slate containing a unique ID number for the session. An investigator can read the ID number out loud and then press a button on the controller. Simultaneously, the software can show a digital clapper closing, make a clapper sound, and log game data with the time stamp according to the internal clock on your playtest machine. All of these can be used to sync the various video streams later (with the clapper sound used to sync streams which cannot see the screen), and it's even possible to sync the game data log. Most even half-decent video editing programs will allow you to put each of these videos into one quarter of the screen and fill the fourth quarter with the date, time, and unique ID of the playtest session. Then you can see all of this data synchronized in a single video.

Privacy Concerns

In modern times, many people are concerned about their personal privacy. You will need to be upfront with your playtesters and let them know that they will be recorded. However, you should also

promise them that the video will only be used for internal purposes and will never be shared with anyone outside of the company.

Running a Formal Individual Playtest

Investigators should seek to make the individual playtest as similar as possible to the experience a player who had bought the game would have at home. The player should be comfortable and at ease. You might want to provide snacks or drinks, and if the game is designed for tablet or console, you might want to give the player a couch or other comfortable seat to sit on. (For computer games, a desk and office chair are often more appropriate.)

Start the playtest by telling the playtester how much you appreciate the time she has set aside to test your game and how useful it will be for you to get her feedback. You should also request that she please speak out loud while playing. Few playtesters will actually do so, but it can't hurt to ask.

Once the playtester has finished the section of the game that you want her to play, an investigator should sit with her and discuss her experience with the game. The questions asked by the investigator should be similar to those that are asked at the end of formal group testing, but the one-on-one format will allow the investigator to frame meaningful follow-up questions and get better information. The post-playtest question and answer sessions should also be recorded, though audio recording is more important than video for the post-play interview.

As with all formal playtesting, it is best if the investigator is not part of the game development team. This helps the investigator's questions and perceptions to not be biased by personal investment in the game. However, after you have found a good investigator, it is very useful to work with the same investigator throughout the development process so that they can provide you information about the progression of playtesters' reactions to the game.

Online Playtesting

As mentioned previously, the largest circle of playtesters is composed of online playtest communities. Your game must be in the beta phase before you attempt this, so these are colloquially known as *beta tests,* and they come in a few forms:

• **Closed:** An invite-only test with a limited number of people. This is where your online tests should start. Initially, you should have only a few trusted people serve as online playtesters. This gives you a chance to find any bugs with your server architecture or any aspects of your game that are unclear before a larger audience sees it.

For *Skyrates*,[1] our first closed online beta started eight weeks into the project and was composed of the four members of the dev team and only twelve other people, all of whom had offices in the same building as the development team. After two weeks of fixing both game and server issues and adding a few more features, we expanded the playtest group to 25 people, all of whom were still in the same building. Two weeks later, we expanded to 50. Up until this point, a member of the dev team had individually sat down with each player and taught her how to play the game.

[1] *Skyrates* (Airship Studios, 2006) is a game that was introduced in Chapter 8, "Design Goals." It made extensive use of the concept of *sporadic play*, where players interacted with the game for only a few minutes at a time throughout their day. Though this is now common behavior for Facebook games, at the time we were developing it, this was an unusual concept, and it required many rounds of playtesting to refine it.

Over the next two weeks, we developed an online game tutorial document and entered the limited beta phase.

• **Limited:** A limited beta is generally open to anyone who signs up, though there are often a few specific limitations in place. The most common limitation is the number of players.

When *Skyrates* first entered the limited beta phase, we capped the number of players at 125 and told our players that they could invite one friend or family member to join the game. This was a much larger number of concurrent players than we'd had in prior rounds, and we wanted to make sure that the server could handle it. After that, we limited the next round to 250 before moving on to our first open beta.

• **Open:** Open betas will allow anyone online to play. This can be fantastic because you can watch your game gain popularity halfway around the globe, but it can also be terrifying because a spike in players can threaten to overload your server. Generally, you want to make sure that your game is near completion before you do an online, open beta.

Skyrates entered open beta at the end of the first semester of development. We didn't expect to work on the game for a second semester, so we left our game server running over the summer. To our surprise, even though *Skyrates* was initially developed as a two-week game experience, several people played the game throughout the summer, and our total numbers for the summer were somewhere between 500 and 1000 players. However, this all happened in 2006 before Facebook became a game platform and before the ubiquity of gaming on smartphones and tablets. While 99% of all games on these platforms don't gain much popularity at all, be aware that a game released on any of them has the potential to go from only a few players to millions in just a few days. Be wary of open betas on social platforms, but know that you need to open up the game eventually.

Automated data logging

You should include automated data logging (ADL) in your game as early as possible. ADL occurs when your game automatically records information about player performance and events any time someone plays your game. This is often recorded to a server online,

but can just as easily be stored as local files and then output by your game later.

At Electronic Arts in 2007, I designed and produced the game *Crazy Cakes* for Pogo.com. *Crazy Cakes* was the first Pogo game to ever use ADL, but afterward it became a standard part of production. The ADL for *Crazy Cakes* was really pretty simple. For each level of the game that was played, we recorded several pieces of data:

• Timestamp: The date and time that the round started

• Player username: This allowed us to talk to players with very high scores and ask them what strategies they employed or contact them if something unusual happened during gameplay

• Difficulty level and round number: We had a total of five difficulty levels, each of which contained four progressively more challenging rounds.

• Score

• Number and type of power-ups items used during the round

• Number of tokens earned

• Number of patrons served

• Number of desserts served to patrons: Some patrons requested multiple desserts, which this helped us track.

At the time, Pogo.com had hundreds of beta testers, so three days after releasing *Crazy Cakes*, we had recorded data from over 25,000 playtest sessions! I culled this data to a more manageable 4,000 randomly selected rows and brought it into a spreadsheet application that I used to balance the game based on real data rather than conjecture. Once I thought that the game was well-balanced relative to the data, I selected another 4,000 random rows and confirmed the balance with them.

Other Important Types of Testing

In addition to playtesting, several other important types of testing can be done on a game.

Focus Testing

Focus testing involves gathering a group of people in your game's core demographic (a *focus group*) and getting their reaction to the look, premise, music, or other aesthetic and narrative elements of your game. This is sometimes done by large studios to determine whether it is a good business decision to develop a certain game.

Interest Polling

It is now possible to use social networks like Facebook or crowdfunding sites like Kickstarter to poll the level of interest that your game could generate in the online public. On these websites, you can post a pitch video for a game and receive feedback, either in the form of likes on a social media site or pledges on a crowdfunding site. If you are an independent developer with limited resources, this may be a way to secure some funding for your game, but of course, the results are incredibly varied.

Usability Testing

Many of the techniques now used in formal individual testing grew out of the field of usability testing. At its core, usability testing is about understanding how well testers can understand and use the interface for a piece of software. Because understanding is so important to usability, data gathering of the screen, interaction, and face of the tester are common practices. In addition to the playtesting of your game, it is also important to engage in some individual usability testing that investigates how easily the playtester

can interact with and gain critical information from your game. This can include testing of various layouts for on-screen information, several different control configurations, etc.

Quality Assurance (QA) Testing

Quality assurance testing is focused specifically on finding bugs in your game and ways to reliably reproduce them. There is an entire industry devoted to this kind of testing, and it's largely outside the scope of this book, but the core elements are as follows:

1. Find a bug in the game (a place where it breaks or doesn't react properly).

2. Discover and write down the steps required to reliably reproduce the bug.

3. Prioritize the bug. Does it crash the game? How likely is it to occur for a normal player? How noticeable is it?

4. If the bug is high enough priority, tell the engineering team so that they can fix it.

QA is most often done by both the development team and a group of game testers hired for the final phase of a project. It's also possible to set up ways for players to submit bugs that they find, though most players don't have the training to generate really good bug reports that include clear steps for reproducing the bug. Many free bug tracking tools are available and can be deployed on your project website, including *Bugzilla*, *Mantis Bug Tracker*, and *Trac*.

Automated Testing

Automated testing (AT) occurs when a piece of software attempts to find bugs in your game or game server without requiring human input. For a game, AT could simulate rapid user input (like hundreds of clicks per second all over the screen). For a game server, AT

could inundate the server with thousands of requests per second to determine the level of server load that could cause the server to fail. While AT is complex to implement, it can effectively test your game in ways that are very difficult for human QA testers to accomplish. As with other forms of testing, there are several companies who make their living through automated testing.

Summary

The intent of this chapter was to give you a broad understanding of various forms of testing for your games. As a new game designer, you should find the ones that seem most useful to you and try to implement them. I have had success with several different forms of testing, and I believe that all the forms covered in this chapter can provide you with important information that can improve your game.

In the next chapter, you'll be shown some of the math that lies beneath the surface of the fun in games. You'll also learn about how to use a spreadsheet application to aid you in game balancing.

Chapter 11. Math and Game Balance

In this chapter, we explore various systems of probability and randomness and how they relate to paper game technologies. You also learn a little about the spreadsheet available in Google Docs to help us explore these possibilities.

Following the mathematical explorations (which I promise are as clear and easy to understand as possible), we cover how these systems can be used in both paper and digital games to balance and improve gameplay.

The Meaning of Game Balance

Now that you've made your initial game prototype and experimented with it a few times through playtests, you will probably need to balance it as part of your iteration process. *Balance* is a term that you will often hear when working on games, but it means different things depending on the context.

In a multiplayer game, balance most often means *fairness*: each player should have an equal chance of winning the game. This is most easily accomplished in *symmetric* games where each player has the same starting point and abilities. Balancing an *asymmetric* game is considerably more difficult because player abilities or start positions that may seem balanced could, in practice, demonstrate a bias toward one set of player abilities over the others. This is one of the many reasons why playtesting is critically important.

In a single-player game, balance usually means that the game is at an *appropriate level of difficulty* for the player and the *difficulty changes gradually*. If a game has a large jump in difficulty at any point, that point becomes a place where the game will tend to lose players. This relates to the discussion of flow as a player-centric design goal in Chapter 8, "Design Goals."

In this chapter, you learn about several disparate aspects of math that are all part of game design and balance. This includes understanding probability, an exploration of different randomizers for paper games, and the concepts of weighted distribution, permutations, and positive and negative feedback. Throughout this exploration, you use *Google Sheets*, a free online spreadsheet program, to better explore and understand the concepts presented. At the end of the chapter, you will see how Sheets was used to balance the weapons used in the paper prototype example in Chapter 9, “Paper Prototyping.”

The Importance of Spreadsheets

For some of the things that we’ll be doing in this chapter, a spreadsheet program like Sheets isn’t strictly necessary—you could get the same results with a piece of scratch paper and a calculator—however, I feel it’s important to introduce spreadsheets as an aspect of game balance for a few reasons:

• A spreadsheet can help you quickly grasp gestalt information from numerical data. In Chapter 9, I presented you with several different weapons that each had different stats. At the end of this chapter, we will recreate the process that I went through to balance those weapons to each other, contrasting the weapon stats that I created initially based on gut feeling with those that I refined through use of a spreadsheet.

• Charts and data can often be used to convince non-designers of the validity of a game design decision that you have made. To develop a game, you will often be working with many different people, some of whom will prefer to see numbers behind decisions rather than instinct. That doesn’t mean that you should always make decisions with numbers, I just want you to be able to do so if it’s necessary.

• Many professional game designers work with spreadsheets on a daily basis, but I have seen very few game design programs that

teach students anything about how to use them. In addition, the classes at universities that do cover spreadsheet use tend to be more interested in business or accounting than game balance, and therefore focus on different spreadsheet capabilities than I have found useful in my work.

As with all aspects of game development, the process of building a spreadsheet is an iterative and somewhat messy process. Rather than show you perfect examples of making spreadsheets from start to finish with every little thing planned ahead of time, the tutorials in this chapter are designed to demonstrate not only the steps to make a spreadsheet but also a realistic iterative process of both building and planning the spreadsheet.

The Choice of Google Sheets for This Book

For this book, I have chosen to use Google Sheets because it is free, cross-platform, and easily available. Most other spreadsheet programs have many of the same capabilities as Sheets (e.g., Microsoft Excel, Open Office Calc, and LibreOffice Calc Spreadsheet), but each program is subtly different from the others, so attempting to follow the directions in this chapter in an application other than Google Sheets may lead to frustration.

See the sidebar "Not All Spreadsheet Programs Are Created Equal" for more information on the various programs.

Not All Spreadsheet Programs Are Created Equal

Spreadsheet programs are most commonly used to manage and analyze large amounts of numerical data. Some popular spreadsheet programs in use today are Microsoft Excel, Apache OpenOffice Calc, LibreOffice Calc, Google Sheets, and Apple Numbers.

• **Google Sheets** (http://sheets.google.com) is part of the free, online suite of Google Drive tools. Because it is written in HTML5, it is compatible with most modern web browsers, though you should have a good Internet connection to use it effectively. Google Sheets has improved dramatically since the first edition of this book and is now my spreadsheet of choice. One other advantage that it offers is the ability to work synchronously with several team members online.

• **Microsoft Excel** (http://office.microsoft.com) was once by far the most commonly used program, though it was also the most expensive. There are also some differences between the PC and OS X platforms because they are on different release schedules. It uses the same syntax for formulae as Google Sheets and is still considered the industry standard for most businesses, though in practice, I have found it slower and less elegant than Google Sheets.

• **Apache OpenOffice Calc** (http://openoffice.org) is a free, open source program intended to offer the same functionality as Excel at no cost to the user. It is compatible with PC, OS X, and Linux platforms. There are some subtle ways in which Excel and OpenOffice spreadsheets differ from each other, including the user interface, but they largely share the same functionality. One major difference is that OpenOffice uses semicolons (;) to separate the arguments of a formula, where Excel and Google Sheets use commas (,). Calc was the spreadsheet program that I used in the first edition of the book, but Google Sheets has now improved enough to where I no longer use Calc.

• **Apple Numbers** (http://www.apple.com/numbers/) is included with new Mac computers, but it is also downloadable for about $20. Numbers works only on OS X computers and includes some nice-looking features not available in the other programs, though I find some of them get in the way. The core functionality is very comparable to the rest.

• **LibreOffice Calc** (http://libreoffice.org) is a free, open source program intended to offer the same functionality as Excel at no cost to the user. LibreOffice was originally spun off of the OpenOffice

source code, so they share many similarities. One small advantage that LibreOffice has over OpenOffice if you come from an Excel background is that commas are used to separate parameters in LibreOffice formulae (like Excel) instead of the semicolons used in OpenOffice.

Any of these programs can open and export Microsoft Excel XLS files, though each also has its own native format. Even if you already own or are familiar with one of the others, I'd like for you to give Google Sheets a chance in this chapter.

Examining Dice Probability with Sheets

A large portion of game math comes down to probability, so it's critical that you understand a little about how probability and chance work. We'll start by using Google Sheets to help us understand the probability distribution of rolling various numbers using two six-sided dice (2d6).

On a single roll of a single six-sided die (1d6), you will have any even chance of getting a 1, 2, 3, 4, 5, or 6. That's pretty obvious. However, things get much more interesting when you're talking about adding the results of two dice together. If you roll 2d6, then there are 36 different possibilities for the outcome, all of which are shown here:

Die A: 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6

Die B: 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6

It's certainly possible to write all of these by hand, but I'd like you to learn how to use Sheets to do it as an introduction to using a spreadsheet to aid in game balance. You can look ahead to Figure 11.X to see what we'll be making.

Getting Started in Google Sheets

You will need to be online to use Google Sheets. This is a bit of a negative for it, but it is quickly becoming the standard spreadsheet program that most designers I know use, so you should be familiar with it. To start our exploration into game math, please do the following:

1. Point your browser to: **http://sheets.google.com**

This will redirect you to the main Sheets page, where you can get started. I strongly recommend either Google Chrome or Mozilla Firefox as your browser for this work. Chrome has the advantage of allowing some offline editing of Google Sheets spreadsheets, though it is limited and somewhat confusing. Even when using Chrome, it's much better to be online.

2. Under the *Start a new spreadsheet* heading, click the *Blank* button as shown in Figure 11.1. You will then see an *Untitled spreadsheet* like the one shown in Figure 11.2.

Figure 11.1. Creating a new spreadsheet at http://sheets.google.com



Figure 11.2. A new Google Sheets spreadsheet showing important parts of the interface



Getting Started with Sheets

The cells in a spreadsheet are each named with a column letter and a row number. The top-left cell in the spreadsheet is A1. In Figure 11.2, cell A1 is highlighted with a blue border and a small black box in its bottom-right corner, showing that it is the Active Cell.

The directions that follow will show you how to get started using Sheets:

1. Select cell A1 in Google by clicking it.

2. Type the number `1` on your keyboard and press Return. A1 will now hold the value 1.

3. Click in cell B1 and type `=A1+1` and press return. This creates a formula in B2 that will constantly calculate its value based on A1. All formulas start with an **=**. You'll see that the value of B1 is now 2 (the number you get when you add 1 to the value in A1). If you change the value in A1, B1 will automatically update.

4. Click B1 and copy the cell (*Edit* > *Copy* or Command-C on OS X or Control+C on PC).

5. Hold Shift on the keyboard and Shift-click cell K1. This will highlight the cells from B1 to K1 (you may need to use the scroll bar at the bottom of the Sheets window to scroll to the right).

6. Paste the formula from B1 into the highlighted cells (*Edit* > *Paste* or Command-V on OS X or Ctrl+V on PC). This will paste the formula that you copied from B1 into the cells from B1 to K1 (that is, the formula `=A1+1`). Because the cell reference A1 in the formula is a *relative reference,* it will update based on the position of the new cell into which it has been pasted. In other words, the formula in K1 will be `=J1+1` because J1 is one cell to the left of K1 just as A1 was one to the left of B1.

For more information on relative and absolute references, please read the *Relative and Absolute References* sidebar.

Relative and Absolute References

As part of the formula `=A1+1` in cell B1, the `A1` is storing a *relative reference*, meaning that the actual data stores if the location of the

referenced cell *relative to* cell B1 rather than an absolute reference to cell A1. This means that `A1` in this formula refers to the cell one to the left of the cell the formula is in (B1) and will change if the formula is copied to another cell. This is critical to making spreadsheets easy to use, as you saw in step 6 of the *Getting Started with Sheets* heading.

However, there are times when you want an *absolute reference*, that is a reference to a cell that will not change if the formula is moved or copied. To make a cell reference absolute, add a `$` (dollar sign) before both the column and row. For example, to make the `A1` reference absolute, convert it to `$A$1`. It is also possible to make just the column or just the row absolute, by only adding the `$` to one or the other.

In Figure 11.3, you can see an example of partial absolute references. Here, I've written a function to subtract various numbers of days from people's birthdays so that I know when to start looking for presents for them. You can see that the formula in B5 is `=B$3-$A5`., and this same formula was copied and pasted across B5:O7. The partial absolute reference `B$3` indicates that the column should change, but not the row, and the partial absolute reference `$A5` indicates that the row should change, but not the column.

Figure 11.3. An example of partially absolute references



In Figure 11.3, the formula changed as it was copied across the other cells in as follows:

Naming the Document

To name the document, do the following:

1. Click the words *Untitled spreadsheet* in the Document Name area shown in Figure 11.2.

2. Change the name of this spreadsheet to *2d6 Dice Probability* and press Return.

Creating a Row of Numbers from 1 to 36

The preceding instructions should leave you with the numbers 1 through 11 in the cells A1:K1, in other words, the cells from A1 to K1 (the colon is used to define a range between the two listed cells). Next, we will extend the numbers to count all the way from 1 to 36 (for the 36 possible die rolls)

First, you'll need to make enough columns to hold all the cells. Right now, all of the columns are quite wide, and if you scroll to the right, you'll see that the columns stop at Z. First, we'll narrow the columns, and then we'll add enough for them to hold 36 different numbers.

Adding More Columns

1. Click directly on the column A header (i.e., the A at the top of column A).

2. Scroll to the right (using the scroll bar at the bottom of the Sheets window) and Shift-Click on the column Z header. This will select all columns A:Z.

3. When you hover your mouse cursor over the Z column header, a box with a downward-pointing arrow appears. Click that, and select *Insert 26 Right* from the pop-up menu as is shown in the left side of Figure 11.4A. This will create 26 additional columns lettered AA:AZ.

Figure 11.4. Adding 26 columns to the right of column Z



Setting Column Widths

We want to be able to see 36 columns (A:AJ) on screen at once, to make the columns narrower do the following:

1. Click the column A header.

2. Scroll to the right and shift-click the column AJ header. This will select all 36 columns A:AJ.

3. Hover the mouse over the right edge of the column AJ header, and you will see the edge thicken and turn blue (as shown in Figure 11.4B).

4. Grab the thick blue border like a handle, and make column AJ about 1/3 its original width (shown in Figure 11.4C). This will resize all columns A:AJ. If this is still too wide to fit on your screen, feel free to make them narrower.

Filling Row 1 With the Numbers 1 to 36

Fill row 1 by doing the following:

1. Click B1 to select it. Another way that you can copy the contents of a single cell over a large range is to use the blue square in the lower-right corner of a selected cell (which you can see at the lower-right corner of cell A1 in Figure 11.1).

2. Click and drag the blue square from the corner of B1 to the right until you have highlighted cells B1:AJ1. When you release the mouse button, you will now see that A1:AJ1 are filled with the series of numbers from 1 to 36.

3. If your columns are too narrow to show the numbers, select columns A:AJ again and resize them to a comfortable width. Instead of dragging the edge between columns, you can also double-click it to set all columns to their optimal width, however, if you do that here,

the columns with a single digit will be narrower than those with two digits.

Making the Row for Die A

Now, we have a series of numbers, but what we really want is two rows like those for Die A and Die B that were listed earlier in this chapter. We can achieve this with some simple formulas:

1. Click the Function Button (labeled in Figure 11.2) and choose *More Functions...* .

2. There is a text field with the words “Filter with a few keywords...” in it. Replace this text with `MOD`, which will filter the long list of functions down to about six. Look for the one of the Math type named MOD.

3. On the right side of the row for MOD, is a *Learn more* link. Click *Learn more*. This will open a page with a description of MOD. According to it, the MOD function divides one number by another and returns the remainder. For example, the formulae `=MOD(1,6)` and `=MOD(7,6)` would both return a 1 because 1/6 and 7/6 both have a remainder of 1.

4. Return to the spreadsheet by clicking the *2d6 Dice Probability* tab at the top of your browser window.

5. Click cell A2 to select it.

6. Type **`=MOD(A1,6)`**, and press Return. As you type, this will be entering text in both cell A2 and the Formula Bar (labeled in Figure 11.2). Once finished, 1 will be shown in the cell A2.

7. Click cell A2 and shift-click cell AJ2 (to select cells A2:AJ2).

8. Press Command-R on your keyboard (Control+R on PC). This will fill right, copying the contents of the leftmost cell (A2) over everything to the right (B2:AJ2).

Now you'll see that the MOD function is working properly for all 36 cells, however, we wanted the numbers 1 through 6, not 0 through 5, so we need to iterate a bit.

Iterating on the Row for Die A

We need to fix two issues: First, the lowest number should be in columns A, F, L, and so on; and second, the numbers should range from 1 to 6, not 0 to 5. Both of these can be fixed by simple adjustments:

1. Select cell A1 and change its value from 1 to 0. This will cascade through the formulae in B1:AJ1 and give you a series of numbers on row 1 from 0 to 35. Now, the formula in A2 returns 0 (the remainder when 0 is divided by 6), and the numbers in A2:AJ2 will be six series of 0 1 2 3 4 5, which fixes the first issue.

2. To fix the second issue, select A2 and change the formula in A2 to `=MOD(A1,6)+1`. This will simply add one to the result of the previous formula, which increases the formula result in A2 from 0 to 1. This may seem like we've gone in a circle, but once you complete step 3, you'll see the reason for doing so.

3. With A2 still selected, hold Shift+Command (Shift+Control on PC) and press the Right Arrow on your keyboard. This will highlight all of the filled cells to the right of A2, which should be A2:AJ2. Press Command-R on OS X (Control+R on PC) to *fill right* again.

Now, the row for Die A is complete and you have six series of the numbers 1 2 3 4 5 6. The mod values still range from 0 to 5, but now they are in the correct order, and adding one to them has generated the numbers that we wanted for Die A.

Making the Row for Die B

The row for Die B includes six repetitions of each number on the die. To accomplish this, we will use the division and floor functions in

Sheets. Divisions works as you would expect it to (for example, `=3/2` will return the result 1.5), however, floor may be a function that you have not encountered before.

1. Select cell A3.

2. Type **`=FLOOR`**, and you will see pop-up help below cell A3 that tells you "Rounds number down to the nearest integer multiple."

FLOOR is used to round decimal numbers to integers, but FLOOR always rounds down. For example, `=FLOOR(5.1)` returns 5 and `=FLOOR(5.999)` also returns 5.

3. Enter **`=FLOOR(A1/6)`** into cell A3. You will see that the Result field updates to show a result of 0.

4. As was needed with the Die A row, we must add one to the result of the formula. Change the formula to **`=FLOOR(A1/6)+1`**, and you will see that the result is now 1.

5. Copy the contents of A3. Highlight cells A3:AJ3, and paste (Command-V on OS X, Control+V on PC, or *Edit* > *Paste* on either). This will copy the formula from A3 and paste it into A3:AJ3.

Your spreadsheet should now look like top image in Figure 11.5. However, it would be much easier to understand if it were labeled as is shown in the bottom image of Figure 11.5.

Figure 11.5. Adding clarity with labels



Adding Clarity with Labels

To add the labels shown in the second image of Figure 11.5, you will need to insert a new column to the left of column A:

1. Right-click the column A header and choose *Insert 1 left*. This will insert one new column to the left of the current column A. The new column will be named A, and the old column A will now be column B. If you're on OS X and don't have a right-click button, you can Control-click. See the section "Right-Click on OS X" in Appendix B, "Useful Concepts," for a more permanent way to enable right-click on OS X.

2. Click on the new, empty cell A2 and enter `Die A`.

3. In order to make the column wide enough to see the entire label, either double-click on the edge to the right of the column A header or drag it to the right.

4. Enter `Die B` into A3.

5. Enter `Sum` into A4.

6. To make all the text in column A bold, click the column A header (to select all of column A) and then press Command-B (Ctrl+B on PC) or click the **B** formatting button in the *Text Formats* area shown in Figure 11.2.

7. To make the background of column A slightly gray, make sure that all of column A is still highlighted, and click the paint bucket above the formula bar (on the left side of *Cell Formats* in Figure 11.2). From the paint bucket pop-up menu, choose one of the lighter gray colors. This will set the background color of all highlighted cells.

8. To make the background of row 1 also gray, click the row 1 header (the 1 to the left of row 1) to select the whole row. Then choose the same gray background from the paint bucket pop-up menu.

Now your spreadsheet should look like the bottom half of Figure 11.5.

Tip

THERE'S NO NEED TO SAVE IN GOOGLE SHEETS! Throughout this book—and especially in the tutorials at the end—I will be constantly reminding you to save your files. I've lost a ton of work in many programs due to crashes and other computer errors. However, I've never lost work in Google Sheets because it constantly saves the work I'm doing online to the cloud. The one caveat to this is: if you're working offline in a Google Sheets window in Chrome on a PC and close the window before you go online again, your changes may not be saved, though in experiments I've run, they have been auto-saved even in that case.

Summing the Results of the Two Dice

Another formula will allow you to sum the results of the two dice.

1. Click B4 and enter the formula `=SUM(B2,B3)` , which will sum the values in cells B2 and B3 (the formula `=B2+B3` would also work equally well). This will put the value 2 into B4.

2. Copy B4 and paste it into B4:AK4. Now, row 4 shows the results of all 36 possible rolls of 2d6.

3. To make this stand out more, click the row 4 header and make all of row 4 bold.

Counting the Sums of Die Rolls

Row 4 now shows all the results of the 36 possible rolls of 2d6. However, although the data is there, it is still not very easy to interpret it. This is where we can really use the strength of a spreadsheet. To start the data interpretation, we'll create formulae to count the occurrences of each sum (that is, count how many times we roll a 7 on 2d6):

1. Enter `2` into A7.

2. Enter `3` into A8.

3. Select cells A7 & A8.

4. Drag the little blue box at the bottom-right of A8 down until you've highlighted cells A7:A17 and release the mouse button.

This will fill A7:A17 with the numbers from 2 to 12. Google Sheets recognizes that you're starting a series of numbers with the 2 and 3 in adjacent cells, and when you drag that series over other cells, it continues it.

5. Select cell B7 and type `=COUNTIF(` but don't press the Return or Enter key.

5. Use your mouse to click and drag from B4 to AK4. This will draw a box around B4:AK4 and enter `B4:AK4` into your in-progress formula.

6. Type `,` (a comma).

7. Click A7. This will enter A7 into the formula. At this point, the entire formula should be `=COUNTIF(B4:AK4,A7 .`

8. Type `)` and press Return (Windows, Enter). Now, the formula in B7 will be `=COUNTIF(B4:AK4,A7) .`

The COUNTIF function counts the number of times within a series of cells that certain criterion is met. The first parameter of COUNTIF is the range cells to search in (B4:AK4 in this case), and the second parameter (the entry after the comma) is the thing to search for (the value of A7). In cell B7, the COUNTIF function looks at all the cells B4:AK4 and counts the number of times that the number 2 occurs (because 2 is the number in A7).

Counting All Possible Rolls

Next, you will want to extend this from just counting the number of twos to counting the number of rolls of all numbers from 2 to 12:

1. Copy the formula from B7 and paste it into B7:B17.

You will notice that this doesn't work properly. The counts for all the numbers other than two are 0. Let's explore why this is happening.

2. Select cell B7 and then click once in the formula bar. This will highlight all of the cells that are used in the calculation of the formula in cell B7.

3. Press the Esc (escape) key. This is a critical step because it returns you from the cell-editing mode. If you were to click another cell without pressing Esc first, that cell's reference would be entered into the formula. See the following warning for more information.

Warning

EXITING FORMULA EDITING When working in Sheets, you need to press either Return, Tab, or Esc (Enter, Tab, or Esc on PC) to exit from editing a formula. Return or Tab will accept the changes that you have made, and escape will cancel them. If you don't properly exit from formula editing, any cell you click will be added to the formula (which you don't want to do accidentally). If this does happen to you, you can press Esc to exit editing without changing the actual formula.

---

4. Select cell B8 and click once in the formula bar.

Now, you should see the problem with the formula in B8. Instead of counting the occurrence of threes in B4:AK4, it is looking for threes in B5:AK5. This is a result of the automatic updating of relative references that was covered earlier in this chapter. Because B8 is one cell lower than B7, all of the references in B8 were updated to be one cell lower as well. This is correct for the second argument in the formula (i.e., B8 should be looking for the number in A8 and not A7), but needs to be fixed for the first argument.

5. Press Esc to exit from editing B8.

6. Select B7 and change the formula to `=COUNTIF(B$4:AK$4,A7)`. The $ in the formula locks it to row 4 for the first parameter of the COUNTIF function.

7. Copy the formula from B7 and paste it into B7:B17. Now you will see that the numbers update correctly, and each formula in B7:B17 properly queries the cells B$4:AK$4.

Charting the Results

Now, the cells B7:B17 show you the data we wanted. Across the 36 possible rolls of 2d6, there are six possible ways to roll a 7 but only one way to roll a 2 or a 12. This information can be read in the numbers in the cells, but this kind of thing is much easier to understand in a chart:

1. Select cells A7:B17.

2. Click the chart button shown in Figure 11.2 (if you don't see the chart button on your screen, it may be because your window is too narrow; in this case, click the *More* button at the right of the button bar, and the chart button will be in the pop-up menu that appears). This will bring up the Chart Editor shown in Figure 11.6A.

3. Select the Column chart type (highlighted in blue in Figure 11.6A).

4. Click the *Chart types* tab and check the *Use column A as labels* option. This will convert column A from data shown in the chart to labels shown at the bottom.

5. Click the *Customization* tab and set the Title of the chart to *2d6 Dice Roll Probability*.

6. Scroll down to the Axis: Horizontal section and check the *Treat labels as text* option. This will cause all the numbers for each column

to be shown (rather than treated as numbers that the chart could interpolate between as they were before).

7. When this is done, click Insert, and the chart you made will be inserted into the spreadsheet (see Figure 11.6B). You can move and resize the chart if you wish.

Figure 11.6. A probability distribution chart for 2d6



I know that this was a pretty exhausting way to get this data, but I wanted to introduce you to spreadsheets because they can be an extremely important tool in helping you to balance your games.

The Math of Probability

At this point, you are probably thinking that there must be an easier way to learn about the probability of rolling dice than just enumerating all of the possibilities. Happily, there is an entire branch of mathematics that deals with probability, and this section of the chapter will cover several of the rules that it has taught us.

First, let's try to determine how many possible different combinations there can be if you roll 2d6. Because there are two dice, and each has 6 possibilities, there are 6 x 6 = 36 different possible rolls of the 2 dice. For 3d6, there are 6 x 6 x 6 = 216, or 6^3 different combinations. For 8d6, there are 6^8 = 1,679,616 possibilities! This means that we would require a ridiculously large spreadsheet to calculate the distribution of results from 8d6 if we used the enumeration method that we employed earlier in the chapter for 2d6.

In *The Art of Game Design,* Jesse Schell presents "Ten Rules of Probability Every Game Designer Should Know," [1] which I have paraphrased here:

[1] Schell, *The Art of Game Design*, 155–163.

• **Rule 1: Fractions are decimals are percents:** Fractions, decimals, and percents are interchangeable, and you'll often find yourself switching between them when dealing with probability. For instance, the chance of rolling a 1 on 1d20 is 1/20 or 0.05/1 or 5%. To convert from one to the other, follow these guidelines:

• **Fraction to Decimal**: Type the fraction into a calculator. (Typing 1 ÷ 20 = will give you the result 0.05.) Note that decimals can not represent many numbers accurately (e.g., 2/3 is accurate, whereas 0.666666667 is just an approximation).

• **Percent to Decimal**: Divide by 100 (5% = 5 / 100 = 0.05).

• **Decimal to Percent**: Multiply by 100 (0.05 = (0.05 * 100)% = 5%).

• **Anything to Fraction**: This is pretty difficult, there's often no easy way to convert a decimal or percent to a fraction except for the few equivalencies that most people know (e.g., 0.5 = 50% = 1/2, 0.25 = 1/4).

• **Rule 2: Probabilities range from 0 to 1 (which is equivalent to 0% to 100% and 0/1 to 1/1):** There can never be less than a 0% chance or higher than a 100% chance of something happening.

• **Rule 3: Probability is "sought outcomes" divided by "possible outcomes":** If you roll 1d6 and want to get a 6, that means that there is 1 sought outcome (the 6) and 6 possible outcomes (1, 2, 3, 4, 5, or 6). The probability of rolling a 6 is 1/6 (which is roughly equal to 0.16666 or about 17%). There are 13 spades in a regular deck of 52 playing cards, so if you pick one random card, the chance of it being a spade is 13/52 (which is equal to 1/4 or 0.25 or 25%).

• **Rule 4: Enumeration can solve difficult mathematical problems:** If you have a low number of possible outcomes, enumerating all of them can work fine, as you saw in the earlier 2d6 spreadsheet example. If you have a larger number (something like 10d6, which has 60,466,176 possible rolls), you could write a computer program to enumerate them. Once you've gotten some

programming under your belt, you should check out the program to do so that is included in Appendix B, “Useful Concepts.”

• **Rule 5: When sought outcomes are mutually exclusive, add their probabilities:** Schell’s example of this is figuring the chance of drawing either a face card *or* an ace from the deck. There are 12 face cards (3 per suit) and 4 aces in the deck. Aces and face cards are mutually exclusive, meaning that there is no card that is both an ace and a face card. The question for this is “What is the probability of drawing a face card **OR** an ace from the deck?” If you have an **OR** in your probability question, then you can add the two probabilities. 12/52 + 4/52 = 16/52 (0.3077 ≈ 31%).

What is the probability of rolling a 1, 2, **OR** 3 on 1d6? 1/6 + 1/6 + 1/6 = 3/6 (0.5 = 50%). Remember, if you use an **OR** to combine mutually exclusive sought outcomes, you can *add* their probabilities.

• **Rule 6: When sought outcomes are not mutually exclusive, multiply their probabilities:** If you want to know the probability of choosing a card that is both a face card **AND** a spade, you can multiply the two probabilities together. Because probabilities are less than or equal to 1, multiplying them usually makes it less likely that the thing will happen. There are 13 spades (13/52) and 12 face cards (12/52). Multiplied together, you get the following:



We know this is correct because there are actually 3 spades that are face cards in the deck (which is 3 out of 52).

Another example would be the probability of rolling a 1 on 1d6 **AND** a 1 on another 1d6. This would be 1/6 x 1/6 = 1/36 (0.0278 ≈ 3%), and as we saw in the enumerated example in Sheets, there is exactly a 1/36 chance of getting a 1 on both dice when you roll 2d6.

Remember, if you use an **AND** to combine non-mutually exclusive sought outcomes, you can *multiply* their probabilities.

Corollary: When sought outcomes are independent, multiply their probabilities: If two actions are completely independent of each other (which is a subset of them not being mutually exclusive), the probability of them both happening is the multiplication of their individual probabilities. For instance, the probability of rolling a 6 on 1d6 is 1/6. The two dice in a 2d6 roll are completely independent of each other, so the probability of getting 6 on both dice is the multiple of the two independent probabilities (1/6 x 1/6 = 1/36) as we saw in our enumerated example in Sheets.

The probability of getting a six on 1d6 (1/6) **AND** getting heads on a coin toss (1/2) **AND** drawing an Ace from a deck of cards (4/52) is 1/156 (1/6 x 1/2 x 4/52 = 6/624 = 1/156).

• **Rule 7: One minus "does" = "doesn't":** The probability of something happening is one minus the probability of it not happening. For instance the chance of rolling a 1 on 1d6 is 1/6, as you know. This means that the chance of *not* rolling a 1 on 1d6 is 1 - 1/6 = 5/6 (0.8333 ≈ 83%). This is useful because it is sometimes easier to figure out the chance of something not happening than something happening.

For example, what if you wanted to calculate the odds of rolling a 6 on at least one die when you roll 2d6? If we enumerate, we'll find that the answer is 11/36 (the sought outcomes being 6_x, x_6, and 6_6 where the x could be any number other than six). You can also count the number of columns with at least one 6 in them in the Sheets chart we made. However, we can also use rules 5, 6, and 7 to figure this out mathematically.

The possibility of rolling a 6 on 1d6 is 1/6. The possibility of rolling a non-6 on 1d6 is 5/6, so the possibility of rolling 6 **AND** a non-6 (6_x) is 1/6 x 5/6 = 5/36. (Remember, **AND** means multiply from Rule 6.) Because this can be accomplished by either rolling 6_x **OR** x_6, we add those two possibilities together: 5/36 + 5/36 = 10/36. (Rule 5: **OR** means add.)

The possibility of rolling a 6 **AND** a 6 (6_6) is 1/6 x 1/6 = 1/36. Because this is another possible mutually exclusive case from 6_x **OR** x_6, they can all be added together: 5/36 + 5/36 + 1/36 = 11/36 (0.3055 ≈ 31%).

This got complicated pretty quickly, but we can actually use Rule 7 to simplify it. If you reverse the problem and look for the chance of *not* getting a 6 in two rolls, that can be restated "What is the chance of getting a non-6 on the first roll **AND** a non-6 on the second roll?" These two sought possibilities are not mutually exclusive, so you can multiply them! So, the chance of getting a non-6 on both rolls is just 5/6 x 5/6 or 25/36. 1 – 25/36 = 11/36, which is pretty awesome and a lot easier to figure out!

Now, what if you were to roll 4d6 and sought at least one 6? This is now simply:



There is about a 52% chance of rolling at least one 6 on 4d6.

• **Rule 8: The sum of multiple dice is not a linear distribution:** As we saw in the enumerated Sheets example of 2d6, though each of the individual dice has a linear distribution—that is, each number 1–6 has an equal chance of happening on 1d6—when you sum multiple dice together, you get a weighted distribution of probability. It gets even more complex with more than two dice, as shown in Figure 11.7.

Figure 11.7. Probability distribution for 2d6, 3d6, 4d6, 5d6, and 10d6



As you can see in the figure, the more dice you add, the more severe the bias is toward the average sum of the dice. In fact, with 10d6, you have a 1/60,466,176 chance of rolling all 6s, but a 4,395,456/60,466,176 (0.0727 ≈ 7%) chance of rolling exactly 35 or a 41,539,796/60,466,176 (0.6869922781 ≈ 69%) chance of rolling a

number from 30 to 40. There are a couple complex math papers about how to calculate these values with a formula, but I followed Rule 4 and wrote a program to do so.

As a game designer, it's not important for you to understand the exact numbers of these probability distributions. The thing that it *is* very important for you to remember is this: *The more dice you have the player roll, the more likely they are to get a number near the average*.

• **Rule 9: Theoretical versus practical probability:** In addition to the theoretical probabilities that we've been talking about, it is sometimes easier to approach probability from a more practical perspective. There are both digital and analog ways in which this can be done.

Digitally, you could write a simple computer program to do millions of trials and determine the outcome. This is often called the *Monte Carlo* method, and it is actually used by some of the best artificial intelligences that have been devised to play chess and go. Go is so complex that until recently, the best a computer could do was to calculate the results of millions of random plays by both the computer and its human opponent and determine the play that statistically led to the best outcome for it. This can also be used to determine the answers to what would be very challenging theoretical problems. Schell's example of this is a computer program that could rapidly simulate millions of rolls of the dice in *Monopoly* and let the programmer know which spaces on the board players were most likely to land on.

Another aspect of this rule is that not all dice are created equal. For instance, if you wanted to publish a board game and were looking for a manufacturer for your dice, it would be very worthwhile to get a couple of dice from each potential manufacturer and roll each of them a couple hundred times, recording the result each time. This might take an hour or more to accomplish, but it would tell you whether the dice from the manufacturer were properly weighted or if they would instead roll a certain number more often than the others.

• **Rule 10: Phone a friend:** Nearly all college students who major in computer science or math will have to take a probability class or two as part of their studies. If you run into a difficult probability problem that you can't figure out on your own, try asking one of them. In fact, according to Schell, the study of probability began in 1654 when the Chevalier de Méré couldn't figure out why he seemed to have a better than even chance of rolling at least one 6 on four rolls of 1d6 but seemed to have a less than even chance of rolling at least one 12 on 24 rolls of 2d6. The Chevalier asked his friend Blaise Pascal for help. Pascal wrote to his father's friend Pierre de Fermat, and their conversation became the basis for probability studies.[2]

[2] Schell, *The Art of Game Design*, 154.

In Appendix B, "Useful Concepts," I have included a Unity program that will calculate the distribution of rolls for any number of dice with any number of sides (as long as you have enough time to wait for it to calculate).

Randomizer Technologies in Paper Games

Some of the most common randomizers used in paper games include dice, spinners, and decks of cards.

Dice

We've already covered a lot of information about dice in this chapter. The important elements are as follows:

• A single die generates randomness with a linear probability distribution.

• The more dice you add together, the more the result is biased toward the average (and away from a linear distribution).

• Standard die sizes include: d4, d6, d8, d10, d12, and d20. Commonly available packs of dice for gaming usually include 1d4,

2d6, 1d8, 2d10, 1d12, and 1d20.

• The 2d10 in a standard dice pack are sometimes called *percentile dice* because one will be used for the 1s place (marked with the numbers from 0–9) and the other for the 10s place (marked with the multiples of 10 from 00–90), giving an even distribution of the numbers from 00 to 99 (where a roll of 0 and 00 is usually counted as 100%).

Spinners

There are a couple of different kinds of spinners, but all have a rotating element and a still element. In most board games, the spinner is composed of a cardboard base that is divided into sections with a spinning arrow mounted above it (see Figure 11.8A). Larger spinners (for example, the wheel from the television show *Wheel of Fortune*) often have the sections on the spinning wheel and the arrow on the base (Figure 11.8B). As long as players spin the spinner with enough force, a spinner is effectively the same as a die from a probability standpoint.

Spinners are often used in children's games for two major reasons:

• Young children lack the motor control to throw a die within a small area, so they will often accidentally throw dice in a way that they roll off of the gaming table.

• Spinners are a lot more difficult for young children to swallow.

Though they are less common in games for adults, spinners provide interesting possibilities that are not feasible with dice:

• Spinners can be made with any number of slots, whereas it is difficult (though not impossible) to construct a die with 3, 7, 13, or 200 sides.

• Spinners can be weighted very easily so that not all possibilities have the same chance of happening. Figure 11.8C shows a

hypothetical spinner to be used by a player when attacking. On this spinner, the player would have the following chances on a spin:

• 3/16 chance of Miss

• 5/16 chance of Hit 1

• 3/16 chance of Hit 2

• 2/16 chance of Hit 3

• 1/16 chance of Hit 4

• 1/16 chance of Hit 5

• 1/16 chance of Crit!

Figure 11.8. Various spinners. In all diagrams, the gray elements are static, and the black element rotates.



Decks of Cards

A standard deck of playing cards includes 13 cards of 4 different suits and sometimes two jokers (see Figure 11.9). This includes the ranks 1 (also called the Ace) through 10, Jack, Queen, and King in each of the four suits: Clubs, Diamonds, Hearts, and Spades.

Figure 11.9. A standard deck of playing cards with two jokers[3]



[3] Vectorized Playing Cards 1.3 (http://code.google.com/p/vectorized-playing-cards/) ©2011 - Chris Aguilar Licensed under LGPL 3 - www.gnu.org/copyleft/lesser.html

Playing cards are very popular because of both their compactness and the many different ways that they can be divided.

In a draw of a single card from a deck without Jokers, you have the following probabilities:

• Chance of drawing a particular single card: 1/52 (0.0192 ≈ 2%)

• Chance of drawing a specific suit: 13/52 = 1/4 (0.25 = 25%)

• Chance of drawing a face card (J, Q, or K): 12/52 = 3/13 (0.2308 ≈ 23%)

Custom Card Decks

A deck of cards is one of the easiest and most configurable randomizers that can be made for a paper game. You can very easily add or remove copies of a specific card to change the probability of that card appearing in a single draw from the deck. See the section on weighted distributions later in this chapter for more information.

Tips for Making Custom Card Decks

One of the difficulties in making custom cards is getting material for them that you can easily shuffle. 3x5 note cards don't work particularly well for this, but there are a couple of better options:

• Use marker or stickers to modify an existing set of cards. Sharpees work well for this and don't add thickness to the card like stickers do.

• Buy a deck of card sleeves, insert a slip of paper along with a regular card into each, as described in Chapter 9, "Paper Prototyping."

The key thing you want to avoid when making a deck (or any element of a paper prototype) is putting too much time into any one piece. After you've devoted time to making a lot of nice cards (for

instance) you may be less willing to remove any of those cards from the prototype or to scrap them and start over.

Digital Deck Construction

I have recently started using digital deck construction tools like nanDECK in my game prototyping (http://www.nand.it/nandeck/). nanDECK is Windows software that builds a deck of cards from a simple markup language (somewhat like HTML). My favorite nanDECK feature is its ability to pull card data from an online Google Sheets file and turn it into a full deck of cards. There isn't room in this book to describe how this works, but I recommend checking out nanDECK if you're interested. In addition to the pdf reference file on the website, there are also several very useful video tutorials that can be found by searching for "nanDECK" on YouTube.

When to Shuffle a Deck

If you shuffle the entire deck of cards before every draw, then you will have an equal likelihood of drawing any of the cards (just like when you roll a die or use a spinner). However, this isn't how most people use decks of cards. In general, people will draw until the deck is completely exhausted and then shuffle all the cards. This leads to very different behavior from a deck than from an equivalent die. If you had a deck of six cards numbered 1–6, and you drew every card before reshuffling, you would be guaranteed to see each of the numbers 1–6 once for every six times you drew from the deck. Rolling a die six times will not give you this same consistency. An additional difference is that players could count cards and know which have and have not been drawn thus far, giving them insight into the probability of a certain card being drawn next. For example, if the cards 1, 3, 4, and 5 have been drawn from the six-card deck, there is a 50% chance that the next card will be a 2 and a 50% chance that it will be a 6.

This difference between decks and dice came up in the board game *Settlers of Catan* where some players got so frustrated at the difference between the theoretical probability of the 2d6 rolls in the game versus the actual numbers that came up in play that the publisher of the game now sells a deck of 36 cards (marked with each of the possible outcomes of 2d6) as an option to replace the dice in play, ensuring that the practical probability experienced in the game is the same as the theoretical probability.

Weighted Distributions

A weighted distribution is one in which some options are more likely to come up than others. Most of the examples that we've looked at so far involved linear distributions of random possibilities, but it is a common desire as a designer to want to weight one option more heavily than another. For example, in the board game *Small World*, the designers wanted an attacker to get a random bonus on her final attack of each turn about half of the time, and they wanted that bonus to range from +1 to +3. To do this, they created a die with the six sides shown in Figure 11.10.

Figure 11.10. The attack bonus die from Small World with weighted bonus distribution



With this die, the chance of getting no bonus is 3/6 = 1/2 (0.5 = 50%), and the chance of getting a bonus of 2 is 1/6 (0.1666 ≈ 17%), so the chance of no bonus is weighted much more heavily than the other three choices.

What if instead, you still wanted the player to get a bonus only half of the time, but you wanted for the bonus of 1 to be three times more likely than the 3, and the 2 to be twice as likely as the 3. This would give us the weighted distribution shown in Figure 11.11.

Figure 11.11. Die with 1/2 chance of 0, 1/4 chance of 1, 1/6 chance of 2, and 1/12 chance of 3



Luckily, this adds up to 12 total possible sides for a die (a common die size). However, if it didn't add up to a common size, you could always create a spinner or a deck of cards with the same probabilities (though the card deck would need to be shuffled each time before drawing a card). It's also possible to model weighted distributions with randomized outcomes in Sheets. Doing so is very similar to how you will deal with random numbers later in Unity and C#.

Weighted Probability in Sheets

Weighted probability is commonplace in digital games. For instance, if you wanted an enemy who encounters the player to attack her 40% of the time, adopt a defensive posture 40% of the time, and run away 20% of the time, you could create an array of values [Attack, Attack, Defend, Defend, Run][4] and have the enemy's artificial intelligence code pull a random value from it when the player was first detected.

[4] Square brackets (i.e., [ ]) are used in C# to define arrays (a group of values), so I'm using them here to group the five possible action values.

Following these steps will give you a Sheets worksheet that can be used to randomly select from a series of values. Initially, it will pick a random number between 1 and 12, and once the worksheet is made, you can replace choices in column A with any that you choose.

1. Add a new Worksheet to your existing Sheets document by clicking the plus symbol to the left of the existing *Sheet 1* worksheet tab at the bottom of the window (see Figure 11.12).

Figure 11.12. Google Sheets table for weighted random number selection



2. Fill in everything shown in columns A and B of Figure 11.12 but leave column C empty for now. To right-align the text in column B, select cells B1:B4 and choose *Format* > *Alignment* > *Right* from the Sheets menu bar (inside the browser window, as shown highlighted in a blue rectangle in Figure 11.2).

3. Select cell C1 and enter the formula `=COUNTIF(A1:A100,"<>")`. This will count the number of cells in the range A1:A100 that are not empty (in Sheets, <> means "different from," and not following it with a specified value means "different from nothing"). This gives us the number of valid choices that are listed in column A (we currently have 12).

4. In cell C2 enter the formula `=RAND()`, which will generate a number from 0 to 1 (including 0, but never actually reaching 1).

5. Select cell C3 and enter the formula `=FLOOR(C2*C1)+1`. The number we're flooring is the random number between 0 and 0.9999 multiplied by the number of possible choices, which is 12 in this case. This means that we're flooring numbers between 0 and 11.9999 to give us the integers from 0 to 11. Then we add 1 to the result to give us the integers 1 to 12.

6. In cell C4, enter the formula `=INDEX(A1:A100,C3)`. INDEX() takes a range of values (e.g., A1:A100) and chooses from them based on an index (C3 in this case, which can range from 1 to 12). Now, C4 will choose a random value from the list in column A.

To get a different random value, copy cell C2 and paste it back into C2. This will force a recalculation of the RAND function. You can also make any change to the spreadsheet, and that will cause it to recalculate the RAND (for instance, you could type `1` into E1 and hit Return to force a recalculation).

You can put either numbers or text into the cells in column A as long as you don't skip any rows. Try replacing the numbers in cells A1:A12 with the weighted values from Figure 11.11 (that is, [0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 3]). If you do so and try recalculating the random value in C3 several times, you will see that zero comes up about half of the time. You can also fill the values A1:A5 with [Attack, Attack, Defend, Defend, Run] and see the weighted enemy AI choice that was used as an example at the beginning of this section.

Permutations

There is a traditional game called *Bulls and Cows* (see Figure 11.13) that served as the basis for the board game *Master Mind* (1970 by Mordecai Meirowitz). In this game, each player starts by writing down a secret four-digit code (where each of the digits are different). Players take turns trying to guess their opponent's code, and the first person to guess correctly wins. When a player guesses, her opponent responds with a number of bulls and a number of cows. The guesser gets a bull for each number she guessed that is in the correct position and a cow for each number that is part of the code but not in the right position. In the diagram below, black will represent a bull, and white will represent a cow.

Figure 11.13. An example game of Bulls and Cows



From the perspective of the guesser, the secret code is effectively a series of random choices. Mathematicians call series like these *permutations*. In *Bulls and Cows*, the secret code is a permutation of the ten digits 0–9, where four are chosen with no repeating elements. In the game *Master Mind*, there are eight possible colors, of which four are chosen with no repetition. In both cases, the code is a permutation and not a *combination* because the positions of the elements matter (9305 is not the same as 3905). A combination is a selection of choices where the position doesn't matter. For example,

1234, 2341, 3421, 2431, and so on are all the same thing in a combination. A good example of a combination is the combination of three flavors of ice cream you want in your sundae; it doesn't matter what order they're added in, as long as they're all there.

Permutations with Repeating Elements

We'll start with permutations that allow repetition because the math is a little easier. If there were four digits with repetition allowed, there would be 10,000 possible combinations (the numbers 0000 through 9999). This is easy to see with numbers, but we need a more general way of thinking about it (for cases where there are not exactly 10 choices per slot). Because each digit is independent of the others and each has 10 possible choices, according to probability Rule 6, the probability of getting any one number is 1/10 x 1/10 x 1/10 x 1/10 = 1/10,000. This also tells us that there are 10,000 possible choices for the code (if repetition is allowed).

The general calculation for permutations with repetition allowed is to multiply the number of choices for each slot by each other. With four slots and ten choices each, this is 10 x 10 x 10 x 10 = 10,000. If you were to make the code from six-sided dice instead of digits, then there would be six choices per slot, making 6 x 6 x 6 x 6 = 1296 possible choices.

Permutations with No Repeating Elements

But what about the actual case for *Bulls and Cows* where you're not allowed to repeat any digits? It's actually simpler than you might imagine. Once you've used a digit, it's no longer available. So, for the first slot, you can pick any number from 0–9, but once a 9 has been chosen for the first slot, there are only 9 choices remaining for the second slot (0–8). This continues for the rest of the slots, so the calculation of possible codes for Bulls and Cows is actually 10 x 9 x 8 x 7 = 5040. Almost half of the possible choices are eliminated by not allowing repeating digits.

Using Sheets to Balance Weapons

Another use of math and programs like Google Sheets in game design is the balance of various weapons or abilities. In this section, we'll look at the process that went into balancing the weapons for the paper prototype example in Chapter 9, "Paper Prototyping." As you saw in Chapter 9, each weapon has values for three things:

• The number of shots fired at a time

• The damage done by each shot

• The chance that each shot will hit at a given distance

As we balance these weapons, we want them to feel roughly equal to each other in power, though we want each weapon to have a distinct personality. For each weapon, this would be as follows:

• **Pistol:** A basic weapon; pretty decent in most situations but doesn't excel in any

• **Rifle:** A good choice for mid and long range

• **Shotgun:** Deadly up close but its power falls off quickly; only one shot, so a miss really matters

• **Sniper Rifle:** Terrible at close range but fantastic at long range

• **Machine Gun:** Fires many shots, so even if some miss, the others will usually hit; this should feel like the most reliable gun, though not the most powerful

Figure 11.14 shows the values for the weapons as I initially imagined they might work. The ToHit value is the minimum roll on 1d6 that would hit at that range. For example, in cell K3, you can see that the ToHit for the pistol at a range of 7 is 4, so if shooting a target 7 spaces away, any of 4 or above would be a hit. This is a 50% chance of hitting on 1d6 (because it would hit on a roll of 4, 5, or 6).

Figure 11.14. Initial values for the weapons balance spreadsheet



Create a new spreadsheet document in Sheets and enter all of the data shown in Figure 11.14. To change the background color of a cell, you can use the Cell Color button shown in Figure 11.1.

Determining the Percent Chance for Each Bullet

In the cells under the heading Percent Chance, we want to calculate the chance that each shot of a weapon will hit at a certain distance. To do so, follow these steps.

1. In cell E3, you can see that each shot from a pistol will hit at a distance of 1 if the player rolls a 2 or better on 1d6. This means that it will miss on a 1 and hit on 2, 3, 4, 5, or 6. This is a 5/6 chance (or ≈83%), and we need a formula to calculate this. Looking at probability rule #7, we know that there is a 1/6 chance of it missing (which is the same as the ToHit number minus one). Select cell P3 and enter the formula `=(E3-1)/6`. This will cause P3 to display the chance of the pistol *missing* at a range of one.

2. Using Rule 7 again, we know that 1-miss = hit, so change the formula in P3 to `=1-((E3-1)/6)`. I've added parentheses for clarity, but order of operations does work in Sheets, so the divide operation would happen before the minus operation regardless. Once you've done this, P3 will hold the value 0.8333333.

3. To convert P3 from showing decimal numbers to showing a percentage, select P3 and click the button labeled **%** in the *Number Formats* area shown in Figure 11.2. You will also probably want to click the button just to the right of **%** a couple of times (this button looks like a **.0** with a left arrow). This removes decimal places from the cell view to clean up that %83.33 and make it just show %83.

4. Copy the formula in P3 and past it into all the cells in the range P3:Y7. You'll see that everything works perfectly except for the blank

ToHit cells, which now have a percent chance of %117! The formula needs to be altered to ignore blank cells.

5. Select P3 again and change the formula to `=IF(E3="", "", 1-((E3-1)/6))`. The IF statement has three parts, which are divided by semicolons.

• **`E3=""`:** The first part is a question: is E3 equal to “”. That is, is E3 equal to an empty cell.

• **`""`:** The second part is what to put in the cell if the question evaluates to true. That is, if E3 is empty, make cell P3 empty as well.

• **`1-((E3-1)/6)`:** The third part is what to put in the cell if the question evaluates to false. That is, if E3 is not empty, then use the same formula we had before.

6. Copy the new formula from P3 and paste it into P3:Y7. You will see that the empty cells in the ToHit area now cause empty cells in the Percent Chance area. (For example, L5:N5 are empty, so W5:Y5 are empty as well.)

7. Lastly, let’s add some color to this chart. Select cells P3:Y7. From the Sheets menu bar choose *Format* > *Conditional formatting...* . This will open a new *Conditional format rules* pane on the right side of your window. Conditional formatting is applied differently based on the contents of a cell.

8. Click *Color scale* near the top of the *Conditional format rules* pane.

9. Under the heading *Preview* in the *Color scale* section, you’ll see the word *Default* over a gradient of green color swatches. Click *Default* and choose the bottom, middle option: *Green to yellow to red*.

10. Click *Done*, and your spreadsheet will look like the *Percent Chance* section in Figure 11.15.

Figure 11.15. The Percent Chance and Average Damage sections of the weapons spreadsheet. You will be making the Average Damage section next. (Note that the spreadsheet has been scrolled to the right so that Average Damage can also be shown.)



Calculating Average Damage

The next step in the balancing process is to determine how much average damage each gun will do at a certain distance. Because some guns fire multiple shots, and each shot has a certain amount of damage, the average damage will be equal to the number of shots fired * the amount of damage per shot * the chance that each shot will hit:

1. Highlight columns O:AA and copy them (Command-C, Ctrl+C on PC, or *Edit* > *Copy*).

2. Select cell Z1 and paste (Command-V, Ctrl+V on PC, or *Edit* > *Paste*). This will expand your worksheet out to column AK and fill it with a duplicate of the columns you copied.

3. Enter `Percent Chance` into cell AA1.

4. Select cell AA3 and enter the formula `=IF(P3="", "", $B3*$C3*P3)`. Just like in the formula for P3, the IF statement here ensures that only non-empty cells are calculated. The formula includes absolute column references to columns $B3 and $C3 because B3 is the number of shots, and C3 is the damage per shot. We don't want those references moving to other columns (though we do want them able to move to other rows, so only the column reference is absolute).

5. Select cell AA3 and click the rightmost button in the Number Formats area of the button bar (labeled **123** and a downward triangle). Select *Number* from the pop-up menu.

6. Copy cell AA3 and paste it into AA3:AJ7. Now the numbers are accurate, but the conditional formatting is still tied to that for the Percent Chance section.

7. Select cells AA3:AJ7. If the Conditional format rules pane is no longer showing, choose Format > Conditional formatting... from the Sheets menu bar to open it again.

8. With AA3:AJ7 selected, you should see a rule in the Conditional format rules pane that reads **Color scale** with the cell references `P3:Y7,AA3:AJ7` under it. Click this rule.

9. This will expand the rule and allow you to edit it. Change the *Apply to range* to just `P3:Y7` and click *Done*. This will cause the previous format to only apply to the Percent Chance section.

10. Select AA3:AJ7 once more. In the *Conditional format rules* pane, click *Add new rule*.

11. Choose *Color scale* and *Green to yellow to red* as you did before.

There are now two separate conditional formatting rules, one for the Percent Chance section and another for the Average Damage section. Keeping these rules separate causes the colors to be determined for each separately, which is important because the ranges of the numbers in each section are so different from each other. Now your Average Damage section should look like the one shown in Figure 11.15.

Charting the Average Damage

The next important step is to chart the average damage. While it's possible to look carefully at the numbers and interpret them, it's much easier to have Sheets do the job of charting information, allowing you to visually assess what is going on. Follow these steps to do so:

1. Click and drag to select cells A3:A7. AA3:AJ7.

2. Scroll over to where you can see the Average Damage section. With A3:A7 still selected, hold Command (or Ctrl on PC), click on AA3, and drag to select AA3:AJ7. You should now have A3:A7,AA3:AJ7 selected.

3. Click the Chart Button (see Figure 11.2) to bring up the Chart Editor.

4. Click the *Chart types* tab and choose the first chart under the *Line* heading. It has an image showing a blue and red line with sharp angles.

5. Check the box next to *Switch rows / column*s.

6. Check the box next to *Use column A as headers*.

7. Click the *Insert* button to complete the chart.

You can see the results of the chart in Figure 11.16. As you can see, there are some problems with the weapons. Some, like the sniper rifle and shotgun have personalities as we had hoped (the shotgun is deadly at close range, and the sniper rifle is better at long range), but there are a lot of other problems:

• The machine gun is ridiculously weak.

• The pistol may be too strong.

• The rifle is also overly strong compared to the other weapons.

Figure 11.16. The weapon balance at the halfway point showing the chart of initial weapon stats.



In short, the weapons are not balanced well to each other.

Duplicating the Weapon Data

To rebalance the weapons, it will be very helpful to have the original and rebalanced information next to each other:

1. Start by moving the chart further down the worksheet. It should be somewhere below row 16.

2. Double click the current name of the chart, *Chart title*, and rename it *Original*.

3. Next, you need to make a copy of the data and formulas that you've already worked out. Select the cells A1:AK8 and copy them.

4. Click cell A9 and paste. This should create a full copy of all the data you just created.

5. Change the text in A10 to `Rebalanced`.

This set of rebalanced data will be where you make the changes and try out new numbers.

6. Next, you'll want to make a chart for the new data that is identical to the one for the original weapon stats. Select A11:A15,AA11:AJ15 just as you selected A3:A7,AA3:AJ7 in step 2 of *Charting the Average Damage*. Choose the same chart options as in that section as well, and click *Insert* to create a second chart.

7. Position the new chart to the right of the original one so that you can see both charts and the data above them.

8. Change the name of the new chart to Rebalanced.

Showing Overall Damage

One final stat that you might want to track is overall damage. This sums the average damage that a weapon can do at all ranges to give you an idea of the overall power of the weapon. To do this, we

will take advantage of a trick that will allow us to make a simple bar chart within the spreadsheet (and not in a specific chart).

1. Right-click the AK column header and choose *Insert 1 right*.

2. Drag the right edge of the new AL column heading to the right to make it about as wide as column AA.

3. Right-click the AL column header and choose *Insert 1 right*. If you wish, you can set the background colors to match those I have in Figure 11.17, but it's not necessary.

Figure 11.17. The weapon balance chosen for Chapter 9



4. Enter `Overall Damage` into cell AL1.

4. Select cell AL3 and enter the formula `=SUM(AA3:AJ3)`. This will add up the average damage done by the pistol at all ranges (it should equal `45.33`).

5. For the bar chart trick, we need to be working with integers, not decimal numbers, so the SUM will need to be rounded. Change the formula in AL3 to `=ROUND( SUM(AA3:AJ3) )`. The result will now be `45.00`. To remove the extra zeroes, highlight cell AL3 and click the button to remove decimal places as you did before (the third button in the Number Formats group).

6. Select cell AM3 and enter the formula `=REPT("|", AL3)`. The REPT function repeats text a certain number of times. The text in this case is the pipe character (you type a pipe by holding Shift and pressing the backslash key, which is above the Return/Enter key on most U.S. keyboards), and it is repeated 45 times because the value in AL3 is 45. After you've done this, you will see a little bar of pipe characters extending to the right in cell AM3. Double-click the right edge of the AM column heading to expand the AM column to show this.

7. Select cells AL3:AM3 and copy them. Paste them into cells AL3:AM7 and AL11:AM15. This will give you a text-based total damage bar chart for all weapons, both original and balanced. Once you've done this, adjust the width of column AM again to make sure you can see all the repeated characters.

You can see this REPT()-based bar chart in Figure 11.17.

Rebalancing the Weapons

Now that you have two sets of data and two charts, you can try rebalancing the weapons. How will you make the machine gun more powerful? Will you increase its number of shots, its chance to hit, or its damage per shot? Keep the following things in mind as you balance:

• Units in the game from Chapter 9, "Paper Protyping," have only 6 health, so they will fall unconscious if 6 or more damage is dealt to them.

• In the paper prototype, if an enemy was not downed by an attacking soldier, the enemy could counterattack. This makes dealing 6 damage in an attack much more powerful than dealing 5 damage because it also protects the attacker from counterattack.

• Weapons with many shots (e.g., the machine gun) will have a much higher chance of dealing the average amount of damage in a single turn, whereas guns with a single shot will feel much less reliable (e.g., the shotgun and sniper rifle). Figure 11.7 shows how the probability distribution shifts toward the average when you start rolling dice for multiple shots instead of a single one.

• Even with all this information, some aspects of weapon balance will not be shown in this chart. This includes things like the point made above about multishot weapons having a much higher chance of dealing the average amount of damage as well as the benefit of the

sniper rifle to deal damage to enemies who are too far away to effectively shoot back.

Try your hand at balancing the stats for these weapons. You should only change the values in the range B11:N15. Leave the original stats alone, and don't touch the Percent Chance and Average Damage cells; both they and the Rebalanced chart will update to reflect the changes you make to the Shots, D/Shot, and ToHit cells. Once you've played with this for a while, continue reading.

The Balance Chosen for Chapter 9

In Figure 11.17, you can see the weapon stats that I chose for the prototype in Chapter 9. This is absolutely not the only way to balance these weapons or even the best way to balance them, but it does achieve many of the design goals.

• The weapons each have a personality of their own, and none is too overpowered or underpowered.

• Though the shotgun may look a little too similar to the machine gun in it's chart, the two guns will feel very different due to two factors: 1. A hit with the 6-damage shotgun is an instant knockout, and 2. The machine gun fires many bullets, so it will deal average damage much more often.

• The pistol is pretty decent at close range and is more versatile than the shotgun or machine gun with its ability to attempt to hit enemies at longer range.

• The rifle really shines at mid-range.

• The sniper rifle is terrible at close range, but it dominates long-distance. A hit with the sniper rifle is 6 points of damage like the shotgun, so it will also take down an enemy in one shot.

Even though this kind of spreadsheet-based balancing doesn't cover all possible implications of the design of the weapons, it's still a

critical tool to have in your game design arsenal because it can help you understand large amounts of data quickly. Several designers of free-to-play games spend most of their day modifying spreadsheets to make slight tweaks in game balance, so if you're interested in a job in that field, spreadsheets and data-driven design (like you just did) are very important skills.

Positive and Negative Feedback

One final critical element of game balance to discuss is the concept of positive and negative feedback. In a game with positive feedback, a player who takes an early lead gains an advantage over the other players and is more likely to win the game. In a game with negative feedback, the players who are losing have an advantage.

Poker is an excellent example of a game with positive feedback. If a player wins a big pot and has more money than the other players, individual bets matter less to her, and she has more freedom to do things like bluff (because she can afford to lose). However, a player who loses money early in the game and has little left can't afford to take risks and has less freedom when playing. *Monopoly* has a strong positive feedback mechanism where the player with the best properties consistently gets more money and is able to force other players to sell her their properties if they can't afford the rent when they land on her spaces. Positive feedback is generally frowned upon in most games, but it can be very good if you want the game to end quickly (though *Monopoly* does not take advantage of this; for info on why this may be, check out Chapter 8, "Design Goals"). Single-player games also often have positive feedback mechanisms to make the player feel increasingly powerful throughout the game.

Mario Kart is a great example of a game with negative feedback in the form of the random items that it awards to players when they drive through item boxes. The player in the lead will usually only get a banana (a largely defensive weapon), a set of three bananas, or a green shell (one of the weakest attacks). A player in last place will often get much more powerful items like the lightning bolt that will

slow down every other player in the race. Negative feedback makes games feel more fair to the players who are not in the lead and generally leads to both longer games and to all players feeling that they have still have a chance of winning even if they're pretty far behind.

Summary

There was a lot of math in this chapter, but I hope you saw that learning a little about math can be very useful for you as a game designer. Most of the topics covered in this chapter could merit their own book or course, so I encourage you to look into them further if your interest has been piqued.

In the next chapter, you learn about the specific discipline of puzzle design. Though games and puzzles are similar, there are some key differences that make puzzle design worth separate consideration.

Chapter 12. Guiding the Player

As you've read in earlier chapters, your primary job as a designer is to craft an experience for players to enjoy. However, the further you get into your project and your design, the more obvious and intuitive your game appears to you. This comes from your familiarity with the game and is entirely natural.

However, this means that you need to keep a wary eye on your game, making sure that players who have never seen your game before also intuitively understand what they need to do to experience the game as you intended. This requires careful, sometimes invisible, guidance, and it is the topic of this chapter.

This chapter covers two styles of player guidance: *direct*, where the player knows that she is being guided; and *indirect*, where the guidance is so subtle that players often don't even realize that the guidance is there. The chapter concludes with information about *sequencing*, a style of progressive instruction to teach players new concepts or introduce them to new game mechanics.

Direct Guidance

Direct guidance methods are those that the player is explicitly aware of. Direct guidance takes many forms, but in all of them, quality is determined by immediacy, scarcity, brevity, and clarity:

• **Immediacy:** The message must be given to the player when it is immediately relevant. Some games try to tell the player all the possible controls for the game at the very beginning (sometimes showing a diagram of the controller with all of the buttons labeled), but it is ridiculous to think that a player will be able to remember all of these controls when the time comes that she actually needs to use them. Direct information about controls should be provided

immediately the first time that the player needs it. In the PlayStation 2 game *Kya: Dark Lineage*, a tree falls into the path of the player character that she must jump over, and as it is falling, the game shows the player the message “Press X to jump” at exactly the time she needs to know that information.

• **Scarcity:** Many modern games have lots of controls and lots of simultaneous goals. It is important that the player not be flooded with too much information all at one time. Making instructions and other direct controls more scarce makes them more valuable to the player and more likely to be heeded. This is also the case with missions. A player can only really concentrate on a single mission at once, and some otherwise fantastic open world games like *Skyrim* inundate the player with various missions to the point that after several hours of gameplay, the player could potentially be in the middle of dozens of different missions, many of which will just be ignored.

• **Brevity:** Never use more verbiage than is necessary, and don’t give the player too much information at one time. In the tactical combat game *Valkyria Chronicles*, if you wanted to teach the player to press O to take cover behind some sandbags, the least you would need to say is “When near sandbags, press O to take cover and reduce damage from enemy attacks.”

• **Clarity:** Be very clear about what you’re trying to convey. In the previous example, you might be tempted to just tell the player “When standing near sandbags, press O to take cover,” because you might assume that players should know that cover will shield them from incoming bullets. However, in *Valkyria Chronicles*, cover not only shields you but also drastically reduces the amount of damage you take from bullets that do hit (even if the cover is not between the attacker and the target). For the player to understand everything she needs to know about cover, she must also be told about the damage reduction.

Methods of Direct Guidance

There are a number of methods of direct guidance:

• **Instructions:** The game explicitly tells the player what to do. These can take the form of text, dialogue with an authoritative non-player character (NPC), or visual diagrams and often incorporate combinations of the three. Instructions are one of the clearest forms of direct guidance, but they also have the greatest likelihood of either overwhelming the player with too much information or annoying her by pedantically telling her information she already knows.

• **Call to action:** The game explicitly gives the player an action to perform and a reason to do so. This often takes the form of missions that are given to the player by NPCs. A good strategy is to present the player with a clear long-term goal and then give her progressively smaller medium- and short-term goals that must be accomplished on the way to the long-term goal.

The *Legend of Zelda: Ocarina of Time* begins with the fairy Navi waking Link to tell him that he has been summoned by the Great Deku Tree. This is then reinforced by the first person Link encounters upon leaving his home, who tells him that it is a great honor to be summoned and that he should hurry. This gives Link a clear long-term goal of seeking the Great Deku Tree (and the Great Deku Tree's conversation with Navi before Link is awoken hints to the player that she will be assigned a much longer-term goal once she arrives). Link's path to the Great Deku Tree is blocked by Mido, who tells him that he will need a sword and shield before venturing into the forest. This gives the player two medium-term goals that are required before she can achieve her long-term goal. Along the way to obtaining both, Link must navigate a small maze, converse with several people, and earn at least 40 rupees. These are all small-term, clear goals that are directly tied to the long-term goal of reaching the Great Deku Tree.

• **Map or Guidance System:** Many games include a map or other GPS-style guidance system that directs the player toward her goals or toward the next step in her mission. For example, *Grand Theft Auto V* has a radar/mini-map in the corner of the screen with a

highlighted route for the player to follow to the next objective. The world of *GTA V* is so vast that missions will often take the player into an unfamiliar part of the map, where the player relies very heavily on the GPS. However, be aware that this kind of guidance can lead to players spending most of their time just following the directions of the virtual GPS rather than actually thinking about a destination and choosing a path of their own, which can increase the time it takes for the player to learn the layout of the game world.

• **Pop-ups:** Some games have contextual controls that change based on the objects near the player. In *Assassin's Creed IV: Black Flag*, the same button controls such diverse actions as opening doors, lighting barrels of gunpowder on fire, and taking control of mounted weapons. To help the player understand all the possibilities, pop-ups with the icon for the button and a very short description of the action appear whenever an action is possible.

Indirect Guidance

Indirect guidance is the art of influencing and guiding the player without her actually knowing that she is being controlled. Several different methods of indirect guidance can be useful to you as a designer. I was first introduced to the idea of indirect guidance by Jesse Schell, who presents it as "indirect control" in Chapter 16 of his book *The Art of Game Design: A Book of Lenses.* This list is an expansion of his six methods of indirect control.[1]

[1] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), pp. 283–298.

Constraints

If you give the player limited choices, she will choose one of them. This seems elementary, but think about the difference between a question that asks you to fill in the blank and one that gives you four choices to pick from. Without constraint, players run the risk of

choice paralysis, which occurs when a person is presented with so many choices that she can't weigh them all against each other and instead just doesn't make a choice. This is the same reason that a restaurant menu might have 100 different items on it but only feature images of 20. The restaurant owners want to make it easier for you to make a decision about dinner.

Goals

In the previous section of this chapter, we discussed ways that goals can be used for direct guidance. Goals can also be used to guide the player indirectly. As Schell points out, if the player has a goal to collect bananas and has two possible doors to go through, placing clearly visible bananas behind one of the doors will encourage the player to head toward the door with bananas.

Players are also often willing to create their own goals, to which you can guide them by giving them the materials to achieve those goals. In the game *Minecraft* (the name of which includes the two direct instructions "mine" and "craft"), the designers defined which items the player are able to craft from which materials, and these design choices in turn imply the goals that players are able to create for themselves. Because most of the simplest recipes allow the player to make building materials, simple tools, and weapons, these recipes start the player down the path toward building a defensible fort to make her home. That goal then causes her to explore for materials. In particular, the knowledge that diamond makes the best tools will lead a player to explore deeper and deeper tunnels to find diamond (which is rare and only occurs at depths of about 50-55 meters) and encourage her to expand the amount of the world that she has seen.

Physical Interface

Schell's book covers information about how the shape of a physical interface can be used to indirectly guide the player: If you give a player of *Guitar Hero* or *Rock Band* a guitar-shaped controller, she will generally expect to use it to play music. Giving a *Guitar Hero*

player a regular game controller might lead her to think that she could control her character's movement around the stage (because a standard game controller usually directs character movement), but with a guitar, her thoughts focus on making music.

Another way in which the sense of touch can be used for indirect guidance is through the rumble feature on many game controllers, which enables the controller to vibrate in the players hands at various intensities. Actual automobile racetracks include red and white rumble strips on the inside of turns. The rumble strips alternate height along with color, allowing the driver to feel rumbling in the steering wheel if her wheel goes too far to the inside of the turn and makes contact with the rumble strip. This is helpful because racers are often trying to be as close to the inside of a turn as possible to be on the perfect racing line, and it's not possible to see exactly where the wheels are touching the road from inside the car. This same method is used in many racing games, rumbling the controller when the player is at the extreme inside edge of a turn. Expanding on this, you could imagine keeping the controller still when the player is on the track but causing it to rumble erratically if the player goes off the track into some grass. The tactile sensation would help the player understand that she should return to the track.

Visual Design

Visuals are used in several different ways to indirectly guide the player:

• **Light:** Humans are naturally drawn to light. If you place a player in a dark room with a pool of light at one end, she will usually move toward that light before exploring anything else.

• **Similarity:** Once a player has seen that something in the world is good in some way (helpful, healing, valuable, etc.), she will seek out similar things.

• **Trails:** Similarity can lead to a breadcrumb-trail-like effect where the player picks up a certain item and then follows a trail of similar items to a location that the designer wishes her to explore.

• **Landmarks:** Visually interesting and distinct objects can be used as landmarks. At the beginning of *Journey* by thatgamecompany, the player starts in the middle of a desert next to a sand dune. Everything around her looks like sand and dunes except for a few dark stone markers atop the tallest nearby dune (see Figure 12.1, left). Because this group of markers is the only thing in the landscape that stands out, the player is driven to move up the dune toward it. Once she reaches the top, the camera rises above her, revealing a towering mountain with light bursting from the top (see Figure 12.1, right). The camera move causes the mountain to emerge from directly behind the stone markers, signifying to the player that the mountain is her new goal. The camera move directly transfers the goal state from the markers to the mountain.

Figure 12.1. Landmarks in Journey



When initially designing Disneyland, Walt Disney Imagineering (which at the time was named WED Enterprises) designed various landmarks to guide guests around the part and keep them from bunching up in the main hub. When guests first enter the park, they are located in Main Street USA, which looks like an idealized small American town from the early twentieth century. However, very soon into their journey down Main Street, they notice Sleeping Beauty's Castle at the end of the street and are immediately drawn to it. Upon finally reaching the castle, guests notice that it is much smaller than it initially appeared and that there's really nothing to do there. Now that they are in the main hub of Disneyland, they can see the mountain of the Matterhorn rising in front of them, the space-age statue at the entrance to Tomorrowland to their right, and the fort of Frontierland to their left. From their position in the hub, these new landmarks look much more interesting than the small castle, and

guests soon leave the castle area to disperse through the park toward these new landmarks.[2]

[2] This was first pointed out to me by Scott Rogers, who covers it in more detail in Level 9 (i.e., Chapter 9) of his book *Level Up!*: *The Guide to Great Video Game Design* (Chichester, UK: Wiley, 2010).

Landmarks are also used throughout the *Assassin's Creed* series. Whenever a player first enters a new part of the map, she will see that there are a few structures that are taller than the others in the area. In addition to the natural attraction of these landmarks, each is also a *view point* in the game from which the player can *synchronize*, which updates her in-game map with detailed information about the area. Because the designers have given the player both a landmark and a goal (filling in her map), they can guess that players will often seek a view point as their first activity in a new part of the world.

• **Arrows:** The annotated image in Figure 12.2 shows examples of subtle arrows used to direct the player in the game *Uncharted 3: Drake's Deception* by Naughty Dog. In these images, the player (as Drake) is chasing an enemy named Talbot.

Figure 12.2. Arrows created by line and contrast in Uncharted 3 direct the player where to run.



A. As the player vaults up to the roof of a building, there are numerous lines formed by physical edges and contrasting light that direct the player's attention to the left. These lines include the ledge she is vaulting, the short half-wall in front of her, the boards on the left, and even the facing of the gray plastic chair.

B. Once the player is on top of the roof, the camera angle rotates, and now the ledge, the wall, and the wooden planks all point directly at the next location where the player must jump (the roof of the building at the top of the frame). The cinderblock next to the wall in shot B even forms the head of an arrow made by the wall.

This is particularly important in this moment of the chase because the landing area will collapse when the player hits it, which could cause the player to doubt whether jumping on that roof was the correct direction for the player to have gone. The arrows in the environment minimize this doubt. .

The *Uncharted 3* dev team referred to wooden planks like those shown in this image as *diving boards*, and they were used throughout the game to guide players to make leaps in a specific direction. You can see another diving board in image A of Figure 12.3.

Figure 12.3. Camera-based guidance in Uncharted 3



C. In this part of the same chase, Talbot has run through a gate and slams it in the player's face. The blue fabric on the short wall draws the player's eye to the left, and the folds in the fabric form an arrow to the left as well.

D. The camera has now panned to the left, and from this perspective, the blue fabric forms an arrow pointing directly at the yellow window frame (the player's next goal). Bright blue and yellow colors like those seen in this image are used throughout the game to show the correct path to the player, so their presence here confirms the player's decision to head through the yellow window.

• **Camera:** Many games that involve traversal puzzles use the camera to guide the player. By showing the player the next objective or next jump, the camera guides her in areas where she might otherwise be confused. This is demonstrated in the shots from *Uncharted 3* that are shown in Figure 12.3.

In shot A, the camera is directly behind the player, however, once the player jumps to the handholds in front of her, the camera pans to the left, directing her to move left (shot B). The camera continues to face left (shot C) until the player reaches the far left ladder, at which point

the camera straightens out and pans down to reveal the yellow rungs going forward (shot D).

• **Contrast:** The shots in Figures 12.2 and 12.3 each also demonstrate the use of contrast to guide player attention. There are several forms of contrast demonstrated in Figures 12.2 and 12.3 that contribute to player guidance:

• **Brightness:** In shots A and B of Figure 12.2, the ledge and the wall that form the arrows have the highest range of brightness contrast in the image. The dark areas alongside light areas cause the lines stand out visually.

• **Texture:** In shots A and B of Figure 12.2, the wooden planks are smooth while the surrounding stone textures are rough. In shots C and D of Figure 12.2, the soft texture of the blue fabric contrasts with the hard stone on which it rests. By laying over the stone edge, the fabric also serves to soften the edge, making the player more aware that she can leap over it.

• **Color:** In shots C and D of Figure 12.2, the blue fabric, yellow window frame, and yellow bars contrast with the other colors (or lack thereof) in the scene. In shot D of Figure 12.3, the yellow rung at the bottom stands out because the rest of the scene is mostly blue and gray.

• **Directionality:** Though it is not as commonly used as the other three, contrast in directionality can also be used effectively to draw the eye. In shot A of Figure 12.3, the horizontal rungs stand out because every other line in that part of the screen is vertical.

Audio Design

Schell states that music can be used to influence the player's mood and thereby her behavior.[3] Certain types of music have become linked to various types of activity: Slow, quiet, somewhat jazzy music is often linked to activities like sneaking or searching for clues (as

seen in the *Scooby Doo* cartoon series), whereas loud, fast, powerful music (like that in an action movie) is better suited to scenes where the player is expected to brazenly fight through enemies and feel invincible.

[3] Schell, *Art of Game Design*, 292–293.

Sound effects can also be used to influence player behavior by drawing attention to possible actions that the player can take. In the *Assassin's Creed* series, a shimmering, ringing sound effect plays whenever the player is near a treasure chest. This informs the player that she could choose to take the action of looking for the chest and, because it only happens with a chest is nearby, it tells her that it wouldn't be too far out of her way to do so. With a guaranteed reward in close proximity, the player is usually guided to search for the chest unless she is already engaged in another more important activity.

Player Avatar

The model of the player's avatar (that is, player character) can have a strong effect on player behavior. If the player character looks like a rock star and is holding a guitar, the player might expect for her character to be able to play music. If the player character has a sword, the player would expect to be able to hit things and run into combat. If the player character walks around in a wizard hat and long robe while holding a book instead of a weapon, the player would be encouraged to stay back from direct combat and focus on spells.

Non-Player Characters

Non-player characters (NPCs) in games are one of the most complex and flexible forms of indirect player guidance, and that guidance can take many forms.

Modeling Behavior

NPC characters can model several different types of behavior. In games, behavior modeling is the act of demonstrating a specific behavior and allowing the player to see the consequences of that behavior. Figure 12.4 shows various examples of behavior modeling in the game *Kya: Dark Lineage* by Atari. Types of modeling include:

• **Negative behavior:** In modeling negative behavior, the NPC does something that the player should avoid doing and demonstrates the consequences. In image A of Figure 12.4, circled in red, one of the Nativs has stepped onto one of the circular traps on the ground and has been caught (it then lifted the Nativ up and flew him back to the enemies pursuing both the Nativs and the player).

• **Positive behavior:** The other Nativ in image A (circled in green) jumped over a trap, showing how to avoid it. This is modeling positive behavior, showing the player how to act properly in the game world. Image B shows another example; the Nativ has stopped immediately before a place in the level where air currents blow swiftly from left to right in timed pulses, even though the air isn't blowing yet. The Nativ waits for the air current to blow, and once it stops, he continues running. This models for the player that she should stop before these air current areas, get the timing right, and then continue.

• **Safety:** In images C and D, the Nativ is jumping onto or into something that looks quite dangerous. However, because of his willingness to jump, the player knows that it is safe to follow.

Figure 12.4. NPC Nativs modeling behavior in *Kya: Dark Lineage*



Emotional Connections

Another way in which NPCs influence player behavior is through the emotional connections that the player develops with them.

In the *Journey* images shown in Figure 12.5, the player is following the NPC because of an emotional connection. The beginning of Journey is very lonely, and the NPC in these images is the first emotive creature that the player has encountered on her journey through the desert. When she encounters the creature, it flies around her joyfully and then takes (shot A) off (shot B). In this situation, a player will almost always follow the NPC.

Figure 12.5. Emotional connections in *Journey*



It is also possible to cause the player to follow an NPC because of a negative emotional connection. For example, the NPC could steal something from the player and run, causing the player to chase him in order to retrieve her property. In either case, the reaction of the player is to follow the NPC, and this can be used to guide the player to another location.

Teaching New Skills and Concepts

While direct and indirect guidance usually focus on moving the player through the virtual locations of the game; this final section is devoted to guiding the player to a better understanding of how to play the game.

When games were simpler, it was possible to present the player with a simple diagram of the controls or even to just let them experiment. In *Super Mario Bros.* for the Nintendo Entertainment System (NES), one button caused Mario to jump, and the other button caused him to run (and to shoot fireballs once he picked up a fire flower). Through just a small amount of experimentation, the player could easily understand the functions of the A and B buttons on the NES controller. Modern controllers, however, typically have two analog sticks (that can also be clicked like buttons), one 8-direction D-Pad, six face buttons, two shoulder buttons, and two triggers. Even with all of these possible controls, many modern games have so many

possible interactions allowed to the player that individual controller buttons have various uses based on the current context, as was mentioned when discussing pop-ups in the direct guidance section.

With so much complexity in some modern games, it becomes critical to teach the player how to play the game as they go along. An instruction booklet won't cut it anymore; now the player needs to be guided though experiences that are properly *sequenced*.

Sequencing

Sequencing is the art of gently presenting new information to the player, and most examples follow the basic style shown in Figure 12.6. The figure shows several steps in the sequence from *Kya: Dark Lineage* that first introduces the player to a hovering mechanic that is used many times throughout the game:

• **Isolated introduction:** The player is introduced to a new mechanic that she must use in order to continue in the game. In image A of Figure 12.6, air is constantly blowing upward, and the player must press and hold X to drop down far enough to go under the upcoming wall. There is no time pressure here, and nothing progresses until she holds X and passes under the wall.

• **Expansion:** Image B of Figure 12.6 shows the next step of this sequence. Here, the player is presented with walls blocking both the top and the bottom of the tunnel, so she must learn to hover in the middle of the tunnel by tapping the X button. However, there is still no penalty for failing to do so correctly.

• **Adding danger:** In image C of Figure 12.6, some danger has been added. The red surface of the floor will harm the player if she gets too close; however, the roof is still completely safe, so not pressing X will keep the player safe. Next, in image D, the ceiling is dangerous, and the floor is completely safe, so if the player is still building her skills, she can simply hold the X button and glide forward along the floor.

• **Increased difficulty:** Images E and F of Figure 12.6 show the final stages of this introduction sequence. In image E, the ceiling is still safe, but the player must navigate through the narrow channel ahead. Image F also requires navigation through a narrow channel, but now the danger has been expanded to both ceiling and floor. The player must demonstrate mastery of the X tapping mechanic to hover safely through the tunnel.[4]

[4] Figure 12.6 also shows the use of color contrast to convey information about safety. The color of the tunnel shifts from green to red to show increasing danger, and in image F, the purple light at the end of the tunnel signifies to the player that this trial will be ending soon.

Figure 12.6. The sequence teaching hovering in Kya: Dark Lineage



I've used several images from *Kya: Dark Lineage* throughout this chapter because it is one of the best examples I have ever seen of this kind of sequencing. In the first six minutes of gameplay, the player learns about movement, jumping, avoiding traps, avoiding thorns, the ability to dribble and kick ball-like animals to defuse traps, avoiding horizontal air gusts, base jumping, the hovering shown above, stealth, and about a dozen other mechanics. All of them are taught using sequencing like that shown above, and at the end of playing through the introduction for the first time, I remembered all of them.

This is common in many different games. In the *God of War* series, every time Kratos receives a new weapon or spell, he is told how to use it through pop-up text messages, but then he is immediately shown through sequencing as well. If it's a spell like a lightning strike that could either be used to power devices or electrocute enemies, the player is first asked to use it for the non-combat purpose (e.g., the player would receive the lightning spell in a room with locked doors and must use the lightning to activate devices to open the

doors). Then, the player is presented with a combat that is easily won using the new spell. This not only gives the player experience using the spell in combat but also demonstrates how powerful the spell is, making the player feel powerful as well.

Integration

Once the player understands how to use the new game mechanic in isolation (as described in the previous examples), it's time to teach her how to combine it with other mechanics. This can be done explicitly (for example, the player could be told that casting the lightning spell in water will expand its range from a 6 feet radius when used outside the water to the entire pool of water when used in the water) or implicitly (for example, the player could be placed in combat in a pool of water and would notice herself that when she used the lightning spell, everything in the water was electrocuted, not just those enemies within 6 feet). When later in the game the player attained a spell that allowed her to drench her enemies and cause a temporary pool of water, she would immediately realize that this also allowed her to expand the reach of her lightning spell.

Summary

There are many more methods of player guidance than could fit in this chapter, but I hope that it gave you a good introduction not only to some specific methods but also the reasoning behind why those methods are used. As you design your games, remember to keep player guidance in mind at all times. This can be one of the toughest things to do because to you, as the designer, every game mechanic will seem obvious. It is so difficult to break out of your own perspective that most game companies will seek dozens or hundreds of one-time testers to play their game throughout the development process. It is critically important as a designer to always find new people to test your game and give you feedback on the quality of the guidance from the perspective of someone who has never seen the game before. Games developed in isolation without

the benefit of naïve testers will often either be too difficult for new players or at least have uneven, staggered rises in difficulty that cause frustration. As described in Chapter 10, “Game Testing,” it is critically important to test early, test often, and test with new people whenever you can.

Chapter 13. Puzzle Design

Puzzles are an important part of many digital games as well as an interesting design challenge in their own right. This chapter starts by exploring puzzle design through the eyes of one of the greatest living puzzle designers, Scott Kim.

The latter part of the chapter explores various types of puzzles that are common in modern games, some of which might not be what you would expect.

As you'll learn through this chapter, most single-player games have some sort of puzzle in them, though multiplayer games often do not. The primary reason for this is that both single-player games and puzzles rely on the game system to provide challenge to the player, whereas multiplayer digital games (that are not cooperative) more often rely on other human players to provide the challenge. Because of this parallel between single-player games and puzzles, learning about how to design puzzles will help you with the design of any game in which you intend to have a single-player or cooperative mode.

Scott Kim on Puzzle Design

Scott Kim is one of today's leading puzzle designers. Since 1990, he has written puzzles for magazines such as *Discover, Scientific American,* and *Games*, and he has designed the puzzle modes of several games including *Bejeweled 2*. He has lectured about puzzle design at both the TED conference and the Game Developers Conference. His influential full-day workshop, The Art of Puzzle Design[1]—which he delivered with Alexey Pajitnov (the creator of Tetris) at the 1999 and 2000 Game Developers Conferences—has shaped many game designers ideas about puzzles for over a decade. This chapter explores some of the content of that workshop.

[1] Scott Kim and Alexey Pajitnov, “The Art of Puzzle Game Design” (presented at the Game Developers Conference, San Jose, CA, March 15, 1999), http://www.scottkim.com/thinkinggames/GDC99/index.html

What Is a Puzzle?

Kim states that his favorite definition of puzzle is also one of the simplest:

“A puzzle is fun, and it has a right answer.”[2]

[2] Scott Kim, “What Is a Puzzle?” Accessed January 17, 2014, http://www.scottkim.com/thinkinggames/whatisapuzzle/.

This differentiates puzzles from toys, which are fun but don’t have a right answer, and from games, which are fun but have a goal rather than a specific correct answer. Kim sees puzzles as separate from games, though I personally see them as more of a highly developed subset of games. Though this definition of puzzles is very simple, some important subtleties lie hidden therein.

A Puzzle Is Fun

Kim states that there are three elements of fun for puzzles:[2]

• **Novelty:** Many puzzles rely on a certain specific insight to solve them, and once the player has gained that insight, finding the puzzle’s solution is rather simple. A large part of the fun of solving a puzzle is that flash of insight, the joy of creating a new solution. If a puzzle lacks novelty, the player will often already have the insight required to solve it before even starting the puzzle, and thus that element of the puzzle’s fun is lost.

• **Appropriate difficulty:** Just as games must seek to give the player an adequate challenge, puzzles must also be matched to the player’s skill, experience, and type of creativity. Each player

approaching a puzzle will have a unique level of experience with puzzles of that type and a certain level of frustration that she is willing to experience before giving up. Some of the best puzzles in this regard have both an adequate solution that is of medium difficulty and an expert solution that requires advanced skill to discover. Another great strategy for puzzle design is to create a puzzle that appears to be simple though it is actually quite difficult. If the player perceives the puzzle to be simple, she'll be less likely to give up.

• **Tricky:** Many great puzzles cause the player to shift her perspective or thinking to solve them. However, even after having that perspective shift, the player should still feel that it will require skill and cunning to execute her plan to solve the puzzle. This is exemplified in the puzzle-based stealth combat of Klei Entertainment's *Mark of the Ninja*, in which the player must use insight to solve the puzzle of how to approach a room full of enemies and then, once she has a plan, must physically execute that plan with precision.[3]

[3] Nels Anderson, "Of Choice and Breaking New Ground: Designing Mark of the Ninja" (presented at the Game Developers Conference, San Francisco, CA, March 29, 2013). Nels Anderson, the lead designer of *Mark of the Ninja*, spoke in this talk about narrowing the gulf between intent and execution. They found that making it easier for a player to execute on her plans in the game shifted the skill of the game from physical execution to mental planning, making the game more puzzle-like and more interesting to players. He has posted a link to his slides and his script for the talk on his blog at http://www.above49.ca/2013/04/gdc-13-slides-text.html, accessed March 6, 2014. His talk is also available for free on the GDC Vault at http://gdcvault.com.

And It Has a Right Answer

Every puzzle needs to have a right answer, though many puzzles have several right answers. One of the key elements of a great

puzzle is that once the player has found the right answer, it is clearly obvious to her that she is right. If the correctness of the answer isn't easily evident, the puzzle can seem muddled and unsatisfying.

Genres of Puzzles

Kim identifies four genres of puzzle (see Figure 13.1),[4] each of which causes the player to take a different approach and use different skills. These genres are at the point of intersection between puzzles and other activities. For example, a story puzzle is the mixture of a narrative and a series of puzzles.

[4] Scott Kim and Alexey Pajitnov, "The Art of Puzzle Game Design," slide 7.

Figure 13.1. Kim's four genres of puzzles[5]



[5] Scott Kim and Alexey Pajitnov, "The Art of Puzzle Game Design," slide 7.

• **Action:** Action puzzles like *Tetris* have time pressure and allow players a chance to fix their mistakes. They are the combination of an action game with a puzzle mindset.

• **Story:** Story puzzles like *Myst*, the *Professor Layton* series, and most hidden-object games[6] have puzzles that players must solve to progress through the plot and explore the environment. They combine narrative and puzzles.

[6] *Myst* was one of the first CD-ROM adventure games, and was the number one best-selling CD-ROM game until *The Sims* took that title. The *Professor Layton* series of games is an ongoing series for Nintendo's handheld platforms that wraps many individual puzzles inside an overarching mystery story. Hidden-object games are a popular genre of game where a player is given a list of objects to find

hidden in a complicated scene. They often have mystery plots that they player is attempting to solve by finding the objects.

• **Construction:** Construction games invite the player to build an object from parts to solve a certain problem. One of the most successful of these was *The Incredible Machine*, in which players built Rube Goldberg-like contraptions to cause the cats in each scene to run away. Some construction games even include a construction set that allows the player to devise and distribute her own puzzles. They are the intersection of construction, engineering, and spatial reasoning with puzzles.

• **Strategy:** Many strategy puzzle games are the solitaire versions of the kinds of puzzles that players encounter in games that are traditionally multiplayer. These include things like bridge puzzles (which present players with various hands in a bridge game and ask how play should proceed) and chess puzzles (which give players a few chess pieces positioned on a board and ask how the player could achieve checkmate in a certain number of moves). These combine the thinking required for the multiplayer version of the game with the skill building of a puzzle to help players train to be better at the multiplayer version.

Kim also holds that there are some pure puzzles that don't fit in any of the other four genres. This would include things like *Sudoku* or crossword puzzles.

The Four Major Reasons that People Play Puzzles

Kim's research and experience have led him to believe that people primarily play puzzles for the following reasons:[7]

[7] Scott Kim and Alexey Pajitnov, "The Art of Puzzle Game Design," slide 8.

• **Challenge:** People like to feel challenged and to feel the joy of overcoming those challenges. Puzzles are an easy way for players

to feel a sense of achievement, accomplishment, and progress.

• **Mindless distraction:** Some people seek big challenges, but others are more interested in having something interesting to do to pass the time. Several puzzles like *Bejeweled* and *Angry Birds* don't provide the player with a big challenge but rather a low-stress interesting distraction. Puzzle games of this type should be relatively simple and repetitive rather than relying on a specific insight (as is common in puzzles played for challenge).

• **Character and environment:** People like great stories and characters, beautiful images, and interesting environments. Puzzle games like *Myst*, *The Journeyman Project*, the *Professor Layton* series, and *The Room* series rely on their stories and art to propel the player through the game.

• **Spiritual journey:** Finally, some puzzles mimic spiritual journeys in a couple of different ways. Some famous puzzles like *Rubik's Cube* can be seen as a rite of passage—either you've solved one in your life or you haven't. Many mazes work on this same principle. Additionally, puzzles can mimic the archetypical hero's journey: the player starts in regular life, encounters a puzzle that sends her into a realm of struggle, fights against the puzzle for a while, gains an epiphany of insight, and then can easily defeat the puzzle that had stymied her just moments earlier.

Modes of Thought Required by Puzzles

Puzzles require players to think in different ways to solve them, and most players have a particular mode of thought that they prefer to engage in (and therefore a favorite class of puzzle). Figure 13.2 illustrates these concepts, and the list that follows explains each mode.

Figure 13.2. The modes of thought that Scott Kim has found are often used in puzzles including examples of each mode and of puzzles that use two modes of thought simultaneously[8]

Image

[8] Scott Kim and Alexey Pajitnov, “The Art of Puzzle Game Design.” Slide 9.

Single-Mode Puzzle Types

• **Word:** There are many different kinds of word puzzles. Most rely on the player having a large and varied vocabulary. Word puzzles are often particularly good if you’re designing games for older adults, since most people’s vocabularies peak later in life.

• **Image:** Image puzzle types include jigsaw, hidden-object, and 2D/3D spatial puzzles. Image puzzles tend to exercise the parts of the brain connected to visual/spatial processing and pattern recognition.

• **Logic:** Logic puzzles like *Master Mind*/*Bulls & Cows* (described in Chapter 11, “Math and Game Balance”), riddles, and deduction puzzles cause the player to exercise their logical reasoning. Many games are based on *deductive* reasoning: the top-down elimination of several false possibilities, leaving only one that is true (e.g., a player reasoning “I know that all of the other suspects are innocent, so Colonel Mustard must have killed Mr. Boddy”). These include *Clue*, *Bulls & Cows*, and *Logic Grid* puzzles. There are far fewer games that use *inductive* reasoning: the bottom-up extrapolation from a specific certainty to a general probability (e.g., a player reasoning “The last five times that John bluffed in poker, he habitually scratched his nose; John is scratching his nose now, so he’s probably bluffing”). Deductive logic leads to certainty, while inductive logic makes an educated guess based on reasonable probability. The certainty of the answers has traditionally made deductive logic more attractive to puzzle designers.

Mixed-Mode Puzzle Types

• **Word / Image:** Many games like *Scrabble*, rebuses (like the one in Figure 13.3), and word searches incorporate both the word and image modes of thought to solve. *Scrabble* is a mixed-mode puzzle, but crossword puzzles are not, because in *Scrabble* the player is determining where to place the word and attempting to arrange it relative to score multipliers on the board. These are two acts of visual/spatial reasoning and decision-making that are not needed to play a crossword puzzle.

• **Image / Logic:** Sliding block puzzles, laser mazes, and puzzles like those shown in the second category of Figure 13.3 fit this category.

• **Logic / Word:** Most riddles fall into this category, including the classic "Riddle of the Sphinx," which is the first riddle in Figure 13.3. It was given by the sphinx to Oedipus in the classic Greek tragedy *Oedipus Rex* by Sophocles.

Figure 13.3. Various mixed-mode puzzles (solutions are at the end of the chapter)



Kim's Eight Steps of Digital Puzzle Design

Scott Kim describes eight steps that he typically goes through when designing a puzzle:[9]

[9] Scott Kim and Alexey Pajitnov, "The Art of Puzzle Game Design," slide 97.

1. Inspiration: Just like a game, inspiration for a puzzle can come from anywhere. Alexey Pajitnov has stated that his inspiration for Tetris was the mathematician Solomon Golomb's concept of pentominoes (12 different shapes, each made of five blocks, that could be fit together into an optimal space-filling puzzle) and the desire to use them in an action game. However, there were too many

different five-block pentomino shapes, so he reduced it to the seven four-block tetrominoes found in Tetris.

2. Simplification: Usually you need to go through some form of simplification to get from your original inspiration to a playable puzzle.

a. Identify the core puzzle mechanic, the essential tricky skill required.

b. Eliminate any irrelevant details, narrow the focus.

c. Make pieces uniform. For example, if you're dealing with a construction puzzle, move the pieces onto a uniform grid to make it easier for the player to manipulate.

d. Simplify the controls. Make sure that the controls for the puzzle are appropriate to the interface. Kim talks about how great a *Rubik's Cube* feels in real life but how terrible it would be to manipulate a digital version with a mouse and keyboard.

3. Construction set: Build a tool that makes construction of puzzles quick and easy. Many puzzles can be built and tested as paper prototypes, but if that isn't the case for your puzzle, this is the first place that you will need to do some programming. Regardless of whether it is paper or digital, an effective construction set can make the creation of additional levels much, much easier for you. Find what tasks are repetitive time-wasters in the puzzle construction process and see if you can't make reusable parts or automated processes for them.

4. Rules: Define and clarify the rules. This includes defining the board, the pieces, the ways that they can move, and the ultimate goal of the puzzle or level.

5. Puzzles: Make some levels of the puzzle. Make sure that you create different levels that explore various elements of your design and game mechanics.

6. Testing: Just like a game, you don't know how players will react to a puzzle until you place it in front of them. Even with his many years of experience, Kim still finds that some puzzles he expects to be simple are surprisingly difficult, while some he expects to be difficult are easily solved. Playtesting is key in all forms of design. Usually, step 6 leads the designer to iteratively return to steps 4 and 5 and refine previous decisions.

7. Sequence: Once you have refined the rules of the puzzle and have several levels designed, it's time to put them in a meaningful sequence. Every time you introduce a new concept, it should be done in isolation, requiring the player to use just that concept in the most elementary way. Then you can progressively increase the difficulty of the puzzle that must be solved using that concept. Finally, you can create puzzles that mix that concept with other concepts that the player already understands. This is very similar to the sequencing in Chapter 12, "Guiding the Player," that is recommended for teaching any new game concept to a player.

8. Presentation: With the levels, rules, and sequence all created, it's now time to refine the look of the puzzle. Presentation also includes refinements to the interface and to the way that information is displayed to the player.

Seven Goals of Effective Puzzle Design

You need to keep several things in mind when designing a puzzle. Generally, the more of these goals that you can meet, the better puzzle you will create:

• **User friendly:** Puzzles should be familiar and rewarding to their players. Puzzles can rely on tricks, but they shouldn't take advantage of the player or make the player feel stupid.

• **Ease of entry:** Within one minute, the player must understand how to play the puzzle. Within a few minutes, the player should be immersed in the experience.

• **Instant feedback:** The puzzle should be “juicy” in the way that Kyle Gabler (co-creator of *World of Goo* and *Little Inferno*) uses the word: The puzzle should actively react to player input in a way that feels physical, active, and energetic.

• **Perpetual motion:** The player should constantly be prodded to take the next step, and there should be no clear stopping point. When I worked at Pogo.com, all of our games ended with a *Play Again* button instead of a game over screen. Even a simple thing like that can keep players playing for longer.

• **Crystal-clear goals:** The player should always clearly understand the primary goal of the puzzle. However, it’s also useful to have advanced goals for players to discover over time. The puzzle games *Hexic* and *Bookworm* are examples of puzzles that have very clear initial goals and also include advanced expert goals that veteran players can discover and enjoy over time.

• **Difficulty levels:** The player should be able to engage the puzzle at a level of difficulty that is appropriate to her skill. Just like all games, appropriate difficulty is critical to making the experience fun for players.

• **Something special:** Most great puzzle games include something that makes them unique and interesting. Alexey Pajitnov’s game *Tetris* combines apparent simplicity with the chance for deep strategy and steadily increasing intensity. Both *World of Goo* and *Angry Birds* have incredibly juicy, reactive gameplay.

Puzzle Examples in Action Games

There are a huge number of puzzles within modern AAA game titles. Most of these fall into one of the following categories.

Sliding Blocks / Position Puzzles

These puzzles usually take place in third-person action games and require the player to move large blocks around a gridded floor to create a specific pattern. An alternative version of this used in some games involves positioning mirrors that are used to bounce light or laser beams from a source to a target. One variation that is commonly introduced is a slippery floor that causes the blocks to move continuously until they hit a wall or other obstacle.

• **Game examples:** *Soul Reaver*, *Uncharted*, *Prince of Persia: The Sands of Time*, *Tomb Raider*, several games in *The Legend of Zelda* series

Physics Puzzles

These puzzles all involve using the physics simulation built in to the game to move objects around the scene or hit various targets with either the player character or other objects. This is the core mechanic in the *Portal* series and has become increasingly popular as reliable physics engines like Havok and the Nvidia PhysX system (built in to Unity) have become ubiquitous in the industry.

• **Game examples:** *Portal*, *Half-Life 2*, *Super Mario Galaxy*, *Rochard*, *Angry Birds*

Traversal

These puzzles show you a place in the level that you need to reach but often make it less than obvious how to get there. The player must frequently take detours to unlock gates or open bridges that will allow her to reach her objective. Racing games like *Gran Turismo* can also be seen as traversal puzzles; the player must discover the perfect racing line that will enable her to complete each lap as efficiently and quickly as possible. This is critically important in the “Burning Lap” puzzles of the *Burnout* series in which players are asked to traverse a racecourse that includes sections of oncoming traffic and hairpin turns without making a single mistake.

• **Game examples:** *Uncharted*, *Tomb Raider*, *Assassin's Creed*, *Oddworld: Abe's Oddyssee*, *Gran Turismo*, *Burnout*, *Portal*

Stealth

An extension of traversal puzzles that became important enough to merit its own genre, stealth puzzles ask the player to traverse a level while also avoiding detection by enemy characters, who are usually patrolling a predetermined path or following a specific schedule. Players usually have a way to disable the enemy characters, though this can also lead to detection if performed poorly.

• **Game examples:** *Metal Gear Solid*, *Uncharted*, *Oddworld: Abe's Oddyssee*, *Mark of the Ninja*, *Beyond Good and Evil*, *The Elder Scrolls V: Skyrim*, *Assassin's Creed*

Chain Reaction

These games include physics systems in which various components can interact, often to create explosions or other mayhem. Players use their tools to set traps or other series of events that will either solve a puzzle or gain them an advantage over attacking enemies. The *Burnout* series of racing games include a Crash Mode that is a puzzle game where the player must drive her car into a specific traffic situation and cause the greatest amount of monetary damage through a fantastic multicar collision.

• **Game examples:** *Pixel Junk Shooter*, *Tomb Raider (2013)*, *Half-Life 2*, *The Incredible Machine*, *Magicka*, *Red Faction: Guerilla*, *Just Cause 3*, *Bioshock*, *Burnout*

Boss Fights

Many boss fights, especially in classic games, involve some sort of puzzle where the player is required to learn the pattern of reactions and attacks used by a boss and determine a series of actions that

would exploit this pattern and defeat the boss. This is especially common in third-person action games by Nintendo like those in the *Zelda*, *Metroid*, and *Super Mario* series. One element that is very common in this kind of puzzle is the *rule of three:*

1. The first time the player performs the correct action to damage the boss, it is often a surprise to her.

2. The second time, she is experimenting to see if she now has the insight to defeat the puzzle/boss.

3. The third time, she is demonstrating her mastery over the puzzle and defeats the boss.

Most bosses throughout the *Legend of Zelda* series since *The Ocarina of Time* can be defeated in three attacks, as long as the player understands the solution to the puzzle of that boss.

• **Game examples:** *The Legend of Zelda*, *God of War*, *Metal Gear Solid*, *Metroid*, *Super Mario 64/Sunshine/Galaxy*, *Guacamelee*, *Shadow of the Colossus*, multiplayer cooperative raids in *World of Warcraft*

Summary

As you've seen in this chapter, puzzles are an important aspect of many games that have single-player modes or multiplayer co-op. As a game designer, puzzle design is not a large departure from the skills you've already learned, but there are some subtle differences. When designing a game, the most important aspect is the moment-to-moment gameplay, whereas in puzzle design, the solution and the moment of insight are of primary importance. (In an action puzzle like *Tetris*, however, insight and solution happen with the drop and placement of every piece.) In addition, when the player solves a puzzle, it is important that she can tell that she has found the right answers; but in games interesting decisions rely of there being

uncertainty in the player's mind about the outcome or correctness of decisions.

Regardless of the differences between designing puzzles and games, the iterative design process is as critical for puzzles as it is for all other kinds of interactive experiences. As a puzzle designer, you will want to make prototypes and playtest just as you would for a game, however, with puzzles, it is even more critical that your playtesters have not seen the puzzle before (because they will have already had the moment of insight).

To close, Figure 13.4 shows the solutions to the puzzles in Figure 13.3. I didn't want to give away the answer by saying so, but the insight of the matchstick puzzle is that it actually requires all three modes of thought: logic, image, and word.

Figure 13.4. Mixed-mode puzzle solutions for the puzzles shown in Figure 13.4



Chapter 16. Thinking in Digital Systems

If you've never programmed before, this chapter will be your introduction to a new world: one where you have the ability and skills to make digital prototypes of the games you imagine.

This chapter describes the mindset you need to have when approaching programming projects. It gives you exercises to explore that mindset and helps you think about the world in terms of systems of interconnected relationships and meaning.

At the conclusion of this chapter, you will be in the right mindset to explore the challenges of the "Digital Prototyping" part of this book.

Systems Thinking in Board Games

In the first part of the book, you learned that games are created from interconnected systems. In games, these systems are encoded into the rules of the game and into the players themselves, meaning that all players bring certain expectations, abilities, knowledge, and social norms to the games that they play. For example, when you think about a standard pair of six-sided dice, there are specific expected and unexpected behaviors that the dice carry with them in most board games:

• **Common expected behaviors of 2d6 (two six-sided dice) in board games**

1. Each die is rolled to generate a random number between 1 and 6 (inclusive).

2. The dice are often rolled together, especially if they are the same color and size.

3. When rolled together, the dice are usually summed. For example, a 3 on one die and a 4 on the other would sum to a total of 7.

4. If “doubles” are rolled (that is, both dice show the same value), there is sometimes a special benefit for the player.

• **Common unexpected behaviors of 2d6 in board games**

1. A players will not just place the dice on the values that she would prefer to have.

2. The dice must stay on the table and must land completely flat on a side to be considered a valid roll. Otherwise, they are rerolled.

3. Once rolled, the dice are generally not touched for the rest of that player’s turn.

4. Dice are generally not thrown at other players (or eaten).

While it may seem somewhat pedantic to explore such simple, often unwritten, rules in detail, it serves to show how many of the rules of board games are not actually present in the rule book, rather they are based on the shared understanding of *fair play* among the players. This idea is incumbent in the concept of the magic circle, and it’s a large part of what makes it so easy for a group of children to spontaneously create a game that they all intuitively understand how to play. Most human players carry within them massive preconceptions about how games are played.

Computer games, however, rely on specific instructions to do absolutely everything. At their core—regardless of how powerful they have become over the past several decades—computers are mindless machines that follow very specific instructions several million times per second. It is up to you, the programmer, to provide the computer with a semblance of intelligence by encoding your ideas into very simple instructions for it to follow.

An Exercise in Simple Instructions

One classic exercise for budding computer science students to help them understand how to think in terms of very simple instructions involves telling another person how to stand up from a prone position. You'll need a friend for this.

Ask your friend to lie on his back on the floor, and once he is there, tell him to only follow your exact instructions to the letter. Your goal is to give your friend instructions that will move him into a standing position, however, you cannot use any complex commands like "stand up." Instead, you must only use the kind of simple commands that you might imagine giving to a robot. For example:

• Bend your left elbow closed 90 degrees.

• Extend your right leg halfway.

• Place your left hand on the ground with your palm facing downward.

• Point your right arm at the television.

In reality, even these simple instructions are drastically more complex than anything that could be sent to most robots, and they're pretty open to interpretation. However, for the sake of this exercise, this level of simplicity will suffice.

Give it a try.

How long did it take you to give your friend the right instructions to stand up? If you and your friend try to follow both the rules and the spirit of the exercise, it will take quite a while. If you try it with different people, you'll find that it takes much, much longer if your friend doesn't know ahead of time that you have the end goal of getting them into a standing position.

How old were you the first time you were asked by a member of your family to set the table for a meal? I think I was only about four when my parents decided that I could handle that complex task with my

only instruction being “Please set the table for dinner.” Based on the exercise that you just completed, imagine how many simple instructions you would have to give to a person to recreate the complex task of setting the table, yet children are often able to do this before they start elementary school.

What This Means to Digital Programming

Now, of course, I didn’t give you that exercise to discourage you. In fact, the following two chapters are meant to be really inspirational! Rather, it was given to help you understand the mentality of computers and to set up several metaphors for aspects of computer programming. Let’s take a look.

Computer Language

When I gave you the list of four example commands that you could give, I was outlining the parameters of the language that you could use to talk to your friend. Obviously, this was a pretty loose language definition. Throughout this book, we will be using the programming language *C#* (pronounced “see sharp”), and thankfully, its language definition is far more specific. We explore C# much more later in this part of the book, but suffice to say that I have taught thousands of students several different programming languages over more than a decade, and my experience has shown me that C# is one of the best languages for someone to learn as their first programming language. Though it requires slightly more diligence than simpler languages like Processing or JavaScript, it gives learners a far better understanding of core development concepts that will help them throughout their game prototyping and development careers, and it enforces good coding practices that will eventually make your code development faster and easier.

Code Libraries

In the previous exercise, you can see that it would have been much easier to have been able to tell your friend to “stand up” rather than going through the trouble of having to give so many low-level commands. In that case, “stand up” would have been a multipurpose high-level instruction that could be used to tell your friend what you wanted regardless of the starting position that he was in. Similarly, “please set the table” is a common, high-level instruction that generates the desired outcome regardless of what meal is bring prepared, how many people will be eating, or even what household you are in. In C#, collections of high-level instructions for common behaviors are called *code libraries*, and there are hundreds of them available to you as a C# and Unity developer.

The most common code library that you will use is the collection of code that tailors C# to work properly with the Unity development environment. In your code, this extremely powerful library will be imported under the name `UnityEngine`. The `UnityEngine` library includes code for the following:

• Awesome lighting effects like fog and reflections

• Physics simulations that cover gravity, collisions, and even cloth simulation

• Input from mouse, keyboard, gamepad, and touch-based tablets

• Thousands of other things

In addition, there are thousands of free (and paid) code libraries out there to help make your coding easier. If the thing you want to do is pretty common (e.g., moving an object across the screen smoothly over the course of one second), there’s a good chance that someone has already written a great code library to do so (in this case, the free library iTween by Bob Berkebile, http://itween.pixelplacement.com/index.php).

The prevalence of great code libraries for Unity and C# means that *you* can concentrate on writing code for the new, unique aspects of

your games rather than reinventing the wheel every time you start a new game project. In time, you will also start collecting commonly used bits of your own code into libraries that you will use across multiple projects. In this book, we will start doing so by creating a code library called `ProtoTools` that will grow in capability across several projects in this book.

Development Environment

The Unity game development environment is an essential part of your development experience. The Unity application can best be thought of as an environment in which to collect and compose all of the *assets* that you create for a game. In Unity, you will bring together 3D models, music and audio clips, 2D graphics and textures, and finally the C# scripts that you author. None of these assets are created directly within Unity, rather it is through Unity that they are all composed together into a cohesive computer game. Unity will also be used to position game objects in three-dimensional space, handle user input, set up a virtual camera in your scene, and finally compile all of these assets together into a working, executable game. The capabilities of Unity are discussed extensively in Chapter 17, "Introducing the Unity 5 Development Environment."

Breaking Down Complex Problems into Simpler Ones

One of the key things you must have noticed in the exercise is that the exclusion from giving complex commands like "stand up" meant that you needed to think about breaking complex commands down into smaller, more discrete commands. Although this activity was difficult in the exercise, in your programming, you will find the skill of breaking complex tasks into simpler ones to be one of the greatest tools that you have for tackling the challenges that you face and helping you make the games you want one small piece at a time. This is a skill that I use every day in the development of my games, and I promise that it will serve you well. As an example, let's break

down the Apple Picker game that you will make in Chapter 28, “Prototype 1: Apple Picker” into simple commands..

Game Analysis: Apple Picker

Apple Picker is the first prototype that you will make in this book (built in Chapter 28, “Prototype 1: Apple Picker”). It is based on the game play of the classic Activision game *Kaboom!*, which was designed by Larry Kaplan and was published by Activision in 1981.[1] Many clones of *Kaboom!* have been made through the years, and ours is a somewhat less violent version. In the original game, the player moved buckets back and forth in an attempt to catch bombs being dropped by a “Mad Bomber.” In our version, the player uses a basket to collect apples that are falling from a tree (see Figure 16.1).

[1] http://en.wikipedia.org/wiki/Kaboom!_(video_game)

Figure 16.1. The Apple Picker game you will make in Chapter 28



In this analysis, we will look at each of the *GameObjects*[2] in Apple Picker, analyze each of their behaviors, and break those behaviors down to simple commands in flowchart form. This will demonstrate how simple commands can lead to complex behavior and fun gameplay. I recommend searching for “play Kaboom!” online to see whether you can find an online version of the game to play before digging into this analysis, but the game is simple enough that doing so is not necessary. You can also find a version of the Apple Picker game prototype on the http://book.prototools.net website under Chapter 16, though the Apple Picker game is only a single endless level, whereas *Kaboom!* had eight distinct difficulty levels.

[2] A GameObject is Unity’s name for an object that is active in a game. These can contain many components like a 3D model, texture information, collision information, C# code, and so on.

Apple Picker Basic Gameplay

The player controls the three baskets at the bottom of the screen and is able to move them left and right using the mouse. The apple tree moves back and forth rapidly while dropping apples, and the player must catch the apples using her baskets before they hit the ground. For each apple that the player catches, she earns points, but if even a single apple hits the ground, it and all other remaining apples will disappear, and the player will lose a basket. When the player loses all three baskets, the game is over. (There are a few other rules in the original *Kaboom!* game about the number of points earned per bomb (apple) and how the various levels progress, but those are unimportant for this analysis.)

Apple Picker GameObjects

In Unity terminology, any object in the game—usually meaning anything that you see on screen—is a GameObject. We can also use this term in discussing the elements seen in the screenshot shown in Figure 16.2. For later consistency with our Unity projects, I will capitalize the name of all GameObjects (e.g., Apples, Baskets, and AppleTree) in the following list.

Figure 16.2. Apple Picker with GameObjects labeled



A. Baskets: Controlled by the player, the Baskets move left and right following the player's mouse movements. When a Basket hits an Apple, the Apple is caught, and the player gains points.

B. Apples: The Apples are dropped by the AppleTree and fall straight down. If an Apple collides with any of the three Baskets, it is caught and disappears from the screen (granting the player some points). If an Apple passes off the bottom of the screen, it disappears, and it causes all other Apples on screen to disappear as

well. This destroys one of the Baskets (starting at the bottom). Once this is resolved, the AppleTree starts dropping Apples again.

C. AppleTree: The AppleTree moves left and right randomly while dropping Apples. The Apples are dropped at a regular interval, so the only randomness in the behavior is the left and right movement.

Apple Picker GameObject Action Lists

In this analysis, we're not going to consider the difficulty level or the round structure that are present in the original *Kaboom!* game. Instead we will focus on the moment-to-moment actions taken by each GameObject.

Basket Actions

Basket actions include the following:

• Move left and right following the player's mouse.

• If any Basket collides with an Apple, catch the Apple.[3]

[3] It would also be possible to make this collision and reaction part of the Apple actions, but I have chosen to make it part of Basket.

That's it! The Baskets are very simple.

Apple Actions

Apple actions include the following:

• Fall down.

• If an Apple hits the bottom of the screen, the round ends.[4]

[4] In the final Apple Picker game, ending a round will cause all the Apples on screen to disappear and will delete one of the Baskets

before starting the next round, but that's not important to worry about at this point.

The Apples are also very simple.

AppleTree Actions

AppleTree actions include the following:

• Move left and right randomly.

• Drop an Apple every 0.5 seconds.

The AppleTree is pretty simple too.

Apple Picker GameObject Flowcharts

A flowchart is often a good way to think about how the flow of actions and decisions works in your game. Let's look at some for Apple Picker. Though the following flowcharts refer to things like adding points and ending the round, right now, we're just looking at the actions that take place in a single round, so we're not worrying about how those kinds of scoring and round actions actually work.

Basket Flowchart

In Figure 16.3 the behavior of the Basket has been outlined in a flowchart. The game loops through this flowchart every *frame* (which is at least 30 times every second). This is shown by the oval that is at the top left of the chart. Actions are shown in boxes (e.g., *Match Left/Right Mouse Movement*), and decisions are shown as diamonds. See the sidebar "Frames in Computer Games" to learn more about what constitutes a frame.

Figure 16.3. Basket flowchart



Frames in Computer Games

The term *frame* comes from the world of film. Historically, films were composed of strips of celluloid containing thousands of individual pictures (known as frames). When those pictures were shown in quick succession (at either 16 or 24 frames per second [*fps*]), it produced the illusion of movement. Later, on televisions, the movement was constructed from a series of electronic images projected onto the screen, which were also called frames (and operated at about 30 fps in the United States).

When computer graphics became fast enough to show animation and other moving images, each individual image shown on the computer screen was also called a frame. In addition, all of the computation that takes place leading up to showing that image on screen is also part of that frame. When Unity runs a game at 60 fps, it is not only displaying a different image on screen 60 times per second; in that time, it is also calculating the tremendous amount of math required to properly move objects between one frame to the next.

Figure 16.3 shows a flowchart of all the computation that would go into moving the Basket from one frame to the next.

Apple Flowchart

The Apple has a pretty simple flowchart as well (see Figure 16.4).

Figure 16.4. Apple flowchart



AppleTree Flowchart

The AppleTree flowchart is slightly more complex (see Figure 16.5) because the AppleTree has two decisions to make each frame:

• Does it change direction?

• Does it drop an Apple?

Figure 16.5. AppleTree flowchart



The decision of whether to change direction could just as easily come before or after the actual movement. For the purposes of this chapter, either would have worked.

Summary

As you've now seen, digital games can be broken down into a set of very simple decisions and commands. This task is implicit in how I approached creating the prototypes for this book, and it is something that you will do yourself when you approach your own game design and development projects.

In Chapter 28, we expand upon this analysis and show how these action lists can be converted into lines of code that make your Baskets move, your Apples fall, and your AppleTrees run around like a Mad Bomber dropping Apples.

Chapter 17. Introducing The Unity devELOPMENT environment

This is the start of your programming adventure.

In this chapter, you download Unity, the game development environment that you will use throughout the rest of this book. We talk about why Unity is a fantastic game development tool for any budding game designer or developer and why we've chosen C# as the language for you to learn. You also take a look at the sample project that ships with Unity, learn about the various window panes in the Unity interface, and move these panes into a logical arrangement that will match the examples you see in the rest of the book.

Downloading Unity

First things first, let's start downloading Unity. The Unity installer is over 1 GB in size, so depending on your Internet speed, this could take anywhere from a few minutes to a couple of hours. After you've gotten this process started, we can move on to talking about Unity.

As of this writing, the latest major version of Unity is Unity 5. Under the current Unity release plan, there is a new version released every 90 days. As I am writing this now, the current version is 5.4.2. Regardless of version, Unity is always available for free from Unity's store:

http://store.unity.com

This should take you to a page that with links to several different versions of Unity (see Figure 17.1). Unity is available for both PC and OS X, and it is nearly identical on both platforms. The free

Personal version can handle everything that we do in this book. Click Download Now for the Personal version to start the process.

Figure 17.1. The web page to download Unity



This will cause your computer to download the Unity Download Assistant, a small program (less than 1MB on Windows) that will download the rest of Unity when you run it. You can find the Unity Download Assistant in your Downloads folder.

On OS X

To install Unity on OS X, follow these steps:

1. Open the UnityDownloadAssistant-5.x.x.dmg file that was downloaded. This will open a folder.

2. Double-click the *Unity Download Assistant.app* inside this folder to launch it (Figure 17.2A).

3. OS X will ask you if you are sure that you want to launch this application, since it was downloaded from the Internet. Click *Open* to confirm (Figure 17.2B).

4. On the **Install, activate and get creating with Unity** screen that appears, click *Continue*.

5. In order to install Unity, you must agree to the terms of service by clicking *Agree*.

6. On the screen shown in Figure 17.2C, be sure to check the following options:

• **Unity 5.x.x** – The current Unity version

• **Documentation** – Trust me, you'll definitely need this!

• **Standard Assets** – A number of useful assets that ship with Unity. Includes some nice particle effects, terrain stuff, etc.

• **Example Project** – We'll be looking at this in this chapter.

• **WebGL Build Support** – This is now the only way to get your Unity projects online, and we use it later in the book.

You may need to enter the account password for your Mac or Windows PC to allow Unity to install.

7. The Download Assistant will ask you where to install Unity. I'd recommend your main hard drive. Then, click Continue.

The Download Assistant will start its long process of downloading. With the options I recommended, my download size was several hundred megabytes, so this may take a while. While you're waiting, I'd recommend reading the next section "Introducing our Development Environment."

Figure 17.2. Installation steps for OS X



On Windows

To install Unity on Windows, follow these steps:

1. Open the UnityDownloadAssistant-5.x.x.exe app that was downloaded.

2. Widows will ask you if you want to allow this app to make changes t your PC. Click *Yes* to confirm (Figure 17.3A).

3. On the first screen of the installer, click *Next* >.

4. In order to install Unity, you must check the box next to *I accept the terms of the License Agreement* and then click *Next* >.

5. You should probably be running a 64-bit version of Windows by now, so select 64-bit on the next screen. However, if you are still running 32-bit Windows, choose 32-bit. To make sure, open Windows Settings and choose the *System* icon (which looks like a computer). Then click *About* in the listing on the left. Next to *System type*, you should see either 64-bit or 32-bit (Figure 17.3B). Once you've selected a version, click *Next* >.

6. On the screen shown in Figure 17.3C, be sure to check the following options:

• **Unity 5.x.x** – The current Unity version

• **Documentation** – Trust me, you'll definitely need this!

• **Standard Assets** – A number of useful assets that ship with Unity. Includes some nice particle effects, terrain stuff, etc.

• **Example Project** – We'll be looking at this in this chapter.

• **WebGL Build Support** – This is now the only way to get your Unity projects online, and we use it later in the book.

• **(Microsoft Visual Studio Community 2015)** – I recommend against checking this for now. Visual Studio is a more robust code editor than MonoDevelop (which is included with Unity) that is now available to install along with Unity, but I would not recommend it at this time because this entire book uses MonoDevelop examples. However, if you have a lot of experience using Visual Studio already, you may want to download this.

7. The Download Assistant will ask you where to install Unity. I'd recommend the default location of *C:\Program Files\Unity*. Then, click *Next* >.

The Download Assistant will start its long process of downloading. With the options I recommended, you might be downloading about 3GB of data, so it could take a while. While you're waiting, I'd

recommend reading the next section “Introducing our Development Environment.”

Figure 17.3. Installation steps for Windows



Introducing Our Development Environment

Before you can begin prototyping in earnest, you first need to become familiar with Unity, our chosen development environment. Unity itself can really be thought of as a synthesis program; while you will be bringing all the elements of your game prototypes together in Unity, the actual production of the assets will largely be done in other programs. You will program in MonoDevelop; model and texture in a 3D modeling program like Maya, Autodesk 3ds Max, or Blender; edit images in a photo editor such as Photoshop, Affinity Photo, or GIMP; and edit sound in an audio program such as Pro Tools or Audacity. Because a large section of this book is about programming and learning to program in C# (pronounced “see-sharp”), you’ll be spending most of the time with tutorials using MonoDevelop, but it’s still critically important to understand how to use Unity and how to effectively set up your Unity environment.

Why Choose Unity?

There are many game development engines out there, but we’ve chosen to focus on Unity for several reasons:

• **Unity is free:** With the free Personal version of Unity, you can create and sell games that run on several platforms. As of the writing of this edition of the book, there are almost no features of Unity Plus or Unity Pro that are not included for free in Unity Personal. The one caveat is that if you work for a company or organization that made over $100,000 last year, you must purchase Unity Plus ($35/month), and if your organization made over $200,000 last year, you must purchase Unity Pro ($125/month). The Plus and Pro versions do

allow you slightly better analytics, the ability to set the splash screen when your app launches, more concurrent players in multiplayer games, and a dark editor skin, but that's about it. For a game designer just learning to prototype, the free version is really all that you need.

Tip: Unity Pricing

Unity has made several changes to their pricing structure since the first edition of the book was written, so I would always recommend exploring their current pricing structure at http://store.unity.com.

• **Write once, deploy anywhere:** The free version of Unity can build applications for OS X, PC, the Internet via WebGL, Linux, iOS, Apple tvOS, Android, Samsung TV, Tizen, and Windows Store—all from the same code and files. This kind of flexibility is at the core of Unity; in fact, it's what the product and company are named for. Professionals can even use Unity to create games for the PlayStation 4, Xbox One, and several other game consoles.

• **Great support:** In addition to excellent documentation, Unity has an incredibly active and supportive development community. Hundreds of thousands of developers are using Unity, and many of them contribute to the discussions on Unity forums across the web. The official Unity forum is at: https://forum.unity3d.com/.

• **It's awesome!:** My students and I have joked that Unity has a "make awesome" button. Although this is not strictly true, there are several phenomenal features built in to Unity that will make your games both play and look better by simply checking an option box. Unity engineers have already handled a lot of the difficult game programming tasks for you. Collision detection, physics simulation, pathfinding, particle systems, draw call batching, shaders, the game loop, and many other tough coding issues are all included. All you need to do is make a game that takes advantage of them!

Why Choose C#?

Within Unity, you have the choice to use one of two programming languages: JavaScript or C#.

JavaScript

JavaScript is often seen as a language for beginners; it's easy to learn, the syntax is forgiving and flexible, and it's also used for scripting web pages. JavaScript was initially developed in the mid-1990s by Netscape as a "lite" version of the Java programming language. It was used as a scripting language for web pages, though that often meant that various JavaScript functions worked fine in one web browser but didn't work at all in another. The syntax of JavaScript was the basis for HTML5 and is very similar to Adobe Flash's ActionScript 3. Despite all of this, it is actually JavaScript's flexibility and forgiving nature that make it an inferior language for this book. As one example, JavaScript uses *weak typing*, which means that if we were to create a variable (or container) named *bob*, we could put anything we wanted into that variable: a number, a word, an entire novel, or even the main character of our game. Because the JavaScript variable bob wouldn't have a variable type, Unity would never really know what kind of thing bob was, and bob could change at any time. These flexibilities in JavaScript make scripting more tedious and prevent programmers from taking advantage of some of the most powerful and interesting features of modern languages.

C#

C# was developed in 2000 as Microsoft's response to Java. They took a lot of the modern coding features of Java and put them into a syntax that was much more familiar to and comfortable for traditional C++ developers. This means that C# has all the capabilities of a modern language. For you experienced programmers, these features include function virtualization and delegates, dynamic

binding, operator overloading, lambda expressions, and the powerful Language INtegrated Query (LINQ) query language among many others. For those of you new to programming, all you really need to know is that working in C# from the beginning will make you a better programmer and prototyper in the long run. In my prototyping class at the University of Southern California, I taught using both JavaScript and C#, and I found that students who were taught C# consistently produced better game prototypes, exhibited stronger coding practices, and felt more confident about their programming abilities than their peers who had been taught JavaScript in prior semesters of the class.

Runtime Speed of Each Language

If you've had some experience programming, you might assume that C# code in Unity would execute faster than code written in JavaScript. This assumption would come from the understanding that C# code is usually compiled while JavaScript is interpreted (meaning that compiled code is turned into a computer's machine language by a compiler as part of the coding process, while interpreted code is translated on-the-fly as the player is playing the game, making interpreted code generally slower; this is discussed more in Chapter 18, "Introducing our Language: C#"). However, in Unity, every time you save a file of either C# or JavaScript code, Unity imports it, converts either of the two languages to the same Common Intermediate Language (CIL), and then compiles that CIL into machine language. So, regardless of the language you use, your Unity game prototypes will execute at the same speed.

On the Daunting Nature of Learning a Language

There's no way around it, learning a new language is tough. I'm sure that's one of the reasons that you bought this book rather than just trying to tackle things on your own. Just like Spanish, Japanese, Mandarin, French, or any other human language, there are going to

be things in C# that don't make any sense at first, and there are places that I'm going to tell you to write something that you don't immediately understand. There will also probably be a point where you are just starting to understand some things about the language but feel utterly confused by the language as a whole (which is the exact same feeling you'd have if you took one semester of Spanish class and then tried to watch soap operas on Telemundo). This feeling comes for almost all of my students about halfway through the semester, and by the end of the semester, every one of them feels much more confident and comfortable with both C# and game prototyping.

Rest assured, this book is here for you, and if you read it in its entirety, you will emerge with not only a working understanding of C# but also several simple game prototypes that you can use as foundations on which to build your own projects. The approach that I take in this book comes from many semesters of experience teaching "nonprogrammers" how to find the hidden coder within themselves and, more broadly, how to convert their game ideas into working prototypes. As you'll see throughout this book, that approach is composed of three steps:

1. Concept introduction: Before asking you to code anything for each project, I'll tell you what we're doing and why. This general concept of what you're working towards in each tutorial will give you a framework on which to hang the various coding elements that are introduced in the chapter.

2. Guided tutorial: You'll be guided step by step through a tutorial that will demonstrate these concepts in the form of a playable game. Unlike some other approaches, we will be compiling and testing the game throughout the process so that you can identify and repair bugs (problems in the code) as you go, rather than trying to fix all of them at the end. Additionally, I'll even guide you to create some bugs so that you can see the errors they cause and become familiar with them; this will make it easier when you encounter your own bugs later.

3. Lather, rinse, repeat: In many tutorials, you'll be asked to repeat something. For instance, in Chapter 30: "Space SHMUP," a top-down shooter game like *Galaga*, the tutorial will guide you through the process of making one single enemy type, and then it will ask you to create three others on your own. Don't skip this part! This repetition will really drive the concept home, and it will help your understanding solidify later.

Pro Tip: 90% of Bugs are Just Typos

I've spent so much time helping students fix bugs that now I can very quickly spot a typo in code. The most common include the following:

• **Misspellings:** If you type even one letter wrong, the computer won't have any idea what you're talking about.

• **Capitalization:** To your C# compiler, `A` and `a` are two completely different letters, so `variable`, `Variable`, and `variAble` are all completely different words.

• **Missing semicolons:** Just like almost every sentence in English should end in a period, nearly every statement in C# should end in a semicolon (`;`). If you leave the semicolon out, it will often cause an error on the next line. FYI: It's a semicolon because the period was needed for decimal numbers and what's called "dot syntax" in variable names and subnames (e.g., varName.subVarName.subSubVarName).

Earlier, I mentioned that most of my students feel confused and daunted by C# at about the midway point of the semester, and it's at exactly that time that I assign them the Classic Games Project. They are asked to faithfully recreate the mechanics and game feel of a classic game over the course of four weeks. Some great examples have included *Super Mario Bros.*, *Metroid*, *Castlevania*, *Pokemon*, and even *Crazy Taxi*. By being forced to work things out on their own, to schedule their own time, and to dig deeply into the inner

workings of these seemingly simple games, the students come to realize that they understand much more C# than they thought, and that is the time that everything really falls into place. The key component here is that the thought process changes from “I’m following this tutorial” to “I want to do this... now how do I make it happen?” At the end of this book, you will be prepared to tackle your own game projects (or your own Classic Game Project, if you want). The tutorials in this book can be a fantastic starting point on which to build your own games.

Launching Unity for the First Time

When you first launch Unity, you will need to set up some things.

1. On Windows, you may be asked to allow Unity to communicate through the firewall to the Internet. You should allow this.

2. You will be asked to sign into your Unity Account. If you don’t have one, you need to create it now.

3. On the next screen, you should choose *Unity Personal* and click *Next*.

4. This will cause a License agreement screen to appear. You will not be allowed to install Unity Personal if the company or organization you represent earned more than $100,000 in gross revenue in the previous fiscal year. As a reader of this book, you should probably choose “I don’t use Unity in a professional capacity.” and click *Next*.

5. Click the Getting started tab at the top of the Unity screen and check out the video there. It will give you a little information about getting started. Don’t worry if it’s a little fast. We’ll be going over all of this.

The Example Project

To access the example project, do the following:

1. Click the Projects tab in the Unity launch window, and you should see a Standard Assets Example Project listed. Click the name of this example project, and it should open.

2. When the project opens, you should see something like Figure 17.4. Click the Play button to play this scene (highlighted by a light blue rectangle in Figure 17.4).

Figure 17.4. The Example Project open in Unity with the Play button highlighted in light blue.



While playing, you can hit ESC at any time to bring up the menu of various scenes. When I played, there was a bug with the *Characters > First Person Character* scene where the mouse cursor would disappear when playing that scene (probably intentional) but then not reappear upon switching to another scene (probably a bug), so I'd play the First Person Character scene last. Then you can hit ESC to get your mouse cursor back and click the Play button again to stop.

To be honest, I think that the Unity 4 example project, Angry Bots, did a much better job of showing off the engine, but this project does show you some of the breadth of what Unity can do.

To see video of more exciting games made with Unity, I'd recommend going to their YouTube channel either by searching online for "YouTube Unity" or by going directly to https://www.youtube.com/user/Unity3D.

Setting Up the Unity Window Layout

The last thing we need to do before we start actually making things in Unity is to get our environment laid out properly. Unity is very

flexible, and one of those flexibilities is that it allows you to arrange its window panes however you like. You can see several window layouts by choosing various options from the Layout pop-up menu in the top right corner of the Unity window (see Figure 17.5).

Figure 17.5. Position of the layout pop-up menu and selection of the *2 by 3* layout



1. Choose *2 by 3* from the layout pop-up menu shown in Figure 17.5. This will be the starting point for making our layout.

2. Before doing anything else, let's make the Project pane look a little cleaner. Click on the options pop-up for the Project pane (shown in the blue rectangle in Figure 17.6) and choose *One Column Layout*. This will convert the Project pane to the hierarchical list view used throughout this book.

Figure 17.6. Choosing the *One Column Layout* for the Project pane



Unity enables you to both move window panes around and adjust the borders between them. As shown in Figure 17.7, you can move a pane by dragging its tab (the arrow cursor) or adjust a border between panes by dragging the border between them (the left-right resize arrow).

Figure 17.7. Two types of cursors for moving and resizing Unity's window panes



When you drag a pane by its tab, a small ghosted version will appear (see Figure 17.8). Some locations will cause the pane to snap into place. When this happens, the ghosted version of the tab will appear in the new location.

Figure 17.8. Ghosted and snapped panes when moving them around the Unity window



3. Play around with moving the window panes—using dragging and resizing—until your window looks like Figure 17.9.

Figure 17.9. Proper layout for the Unity window... but it's still missing something



4. Now the last thing we need to add is the Console pane. From the menu bar, choose *Window* > *Console*. Then drag the Console pane below the Hierarchy pane. This will put the Console pane below the Hierarchy pane but not below the Project pane.

5. Click the tab of the Project pane and drag it to the right. You should see it snap into position over the left half of the Hierarchy pane. Release the mouse button, and you should see something similar to the final layout shown in Figure 17.10.

Figure 17.10. Final layout of the Unity window, including the Console pane



6. Now you just need to save this layout in the Layout pop-up menu so that you don't have to go through all that again. Click the Layout pop-up menu and choose *Save Layout...*, as shown in Figure 17.11.

Figure 17.11. Saving the layout



7. Save this layout with the name *Game Dev*, using whatever technique on your platform that pushes the layout name to the top of the menu. On OS X, that would be putting a space before the *G.* On

a Windows machine, that would be putting an underscore before the *G.* By putting a space or an underscore at the beginning of the name, you make sure that this layout is sorted to the top of the menu. Now, any time you need to return to this layout, you can simply choose it from this pop-up menu.

Learning Your Way Around Unity

Before we can really get into coding things, you need to get to know the various window panes that you've just arranged. Refer back to Figure 17.10 as we discuss each pane:

• **Scene pane**: The Scene pane allows you to navigate around your scene in 3D and to select, move, rotate, and scale objects.

• **Game pane**: The Game pane is where you will preview your actual gameplay; it's the window in which you played the Example Project. This pane also shows you the view from the Main Camera in your scene.

• **Hierarchy pane**: The Hierarchy pane shows you every GameObject that is included in your current scene. For now, you can think of each scene as a level of your game. Everything that exists in your scene, from the camera to your player-character, is a GameObject.

• **Project pane**: The Project pane contains all of the assets that are part of your project. An asset is any kind of file that is part of your project, including images, 3D models, C# code, text files, sounds, and fonts among many others. The Project pane is a reflection of the contents of the Assets folder within your Unity project folder on your computer hard drive. These assets are not necessarily in your current scene.

• **Inspector pane**: Any time you click on an asset in the Project pane or a GameObject in the Scene or Hierarchy panes, you will be able to see and edit information about it in the Inspector pane.

• **Console pane**: The Console pane will allow you to see messages from Unity about errors or bugs in your code as well as messages from yourself that will help you understand the inner workings of your own code.[1] We will use the Console pane extensively in the Chapter 19, "Hello World: Your First Program," and Chapter 20, "Variables and Components."

[1] Unity's `print()` and `Debug.Log()` functions allow you to print messages to the Console pane.

Summary

That's it for setup. Now, let's move on to actually developing! As you've seen in this chapter, Unity can create some pretty stunning visuals and compelling gameplay. Though the processes of making beautiful 3D models and shaders (code for the graphics card to make things look amazing) are outside the scope of this book, it's important for you to know the extent of Unity's graphical capabilities. In the next chapter, you'll write your first Unity program.

Chapter 18. Introducing Our Language: C#

This chapter introduces you to the key features of C# and describes some important reasons why it was chosen as the language for this book. It also examines the basic syntax of C#, explaining what is meant by the structure of some simple C# statements.

By the end of this chapter, you will better understand C# and be ready to tackle the more in-depth chapters that follow.

Understanding the Features of C#

As covered in Chapter 16, "Thinking in Digital Systems," programming consists of giving the computer a series of simple commands, and C# is the language through which we will do so. However, there are lots of different programming languages out there, each of which has benefits and drawbacks. Some of the features of C# are that it is

• A compiled language

• Managed code

• Strongly typed

• Function-based

• Object-oriented

Each of these features is described further in the following sections, and each will help you in various ways.

C# Is a Compiled Language

When most people write computer programs, they are not actually writing in a language that the computer itself understands. In fact, each computer chip on the market has a slightly different set of very simple commands that it understands, known as *machine language*. This language is very, very fast for the chip to execute, but it is incredibly difficult for a person to read. For example, the machine language line `000000 00001 00010 00110 00000 100000` would certainly mean something to the right computer chip, but it means next to nothing to human readers. You might have noticed, however, that every character of that machine code is either a 0 or 1. That's because all the more complex types of data—numbers, letters, and so on—have been converted down to individual *bits* of data (i.e., ones or zeros). If you've ever heard of people programming computers using punch cards, this is exactly what they were doing: For most formats of binary punch cards, physically punching a hole in card stock represented a one, while an unpunched hole represented a zero.

For people to be able to write code more easily, human-readable programming languages—sometimes called authoring languages—were created. You can think of an authoring language as an intermediate language meant to act as a go-between from you to the computer. Authoring languages like C# are logical and simple enough for a computer to interpret while also being close enough to written human languages to allow programmers to easily read and understand them.

There is also a major division in authoring languages between *compiled* languages such as BASIC, C++, C#, and Java and *interpreted* languages such as JavaScript, Perl, PHP, and Python (see Figure 18.1).

Figure 18.1. A simple taxonomy of programming languages



In an *interpreted* language, authoring and executing code is a two-step process:

• The programmer writes the code.

• Then, each time any player plays the game, the code is converted from the authoring language to machine language in real time on the player's machine.

The good thing about this is that it enables code portability, because the authoring code can be interpreted specifically for the type of computer on which it is running. For example, the JavaScript of a given web page will run on almost any modern computer regardless of whether the computer is running OS X, Windows, Linux, or one of many mobile operating systems like iOS, Android, Windows Phone, and so on. However, this flexibility also causes the code to execute more slowly due to: the time it takes to interpret the code on the player's computer, the authoring language not being well optimized for the device on which it will run, and a host of other reasons. Because the same interpreted code is run on all devices, it is impossible to optimize for the specific device on which it happens to be running. It is for this reason that the 3D games created in an interpreted language like JavaScript run so much more slowly than those created in a compiled language, even when running on the same computer.

When using a *compiled* language, such as C#, there are three separate steps to the programming process:

• The programmer writes the code in an authoring language like C#.

• A compiler convers the code from an authoring language to a compiled application in machine language for a specific kind of machine.

• The computer executes the compiled application

This added middle process of compilation converts the code from the authoring language into an executable (that is, an application or app) that can be run directly by a computer without the need for an interpreter. Because the compiler has both a complete understanding of the program and a complete understanding of the execution platform on which the program will run, it is possible to incorporate many optimizations into the process. In games, these optimizations translate directly into higher frame rates, more detailed graphics, and more responsive games. Most high-budget games are authored in a compiled language because of this optimization and speed advantage, but this means that a different executable must be compiled for each execution platform.

In many cases, compiled authoring languages are only suited for specific execution platforms. For instance, **Objective C** is Apple Computer's proprietary authoring language for making applications for both OS X and iOS. This language is based on C (a predecessor of C++), but it includes a number of features that are unique to OS X or iOS development. Similarly, XNA was a flavor of C# developed by Microsoft specifically to enable students to author games for both Windows-based personal computers and the Xbox 360.

As mentioned in Chapter 17, "Introducing the Unity 5 Development Environment," Unity enables C# (as well as Boo and a JavaScript flavor named UnityScript) to be used to create games. Any of these three languages are compiled into a Common Intermediate Language (CIL) in an additional compilation step, and that CIL is then compiled to target any number of platforms, from iOS to Android to Mac, Windows PC, game consoles such as the PlayStation and Xbox, and even interpreted languages such as WebGL (a specific form of JavaScript used in web pages). This additional CIL step ensures that Unity programs are able to be compiled across many platforms even if they are written in UnityScript or Boo, but I still find C# to be vastly superior to the other two.

The ability to write once and compile anywhere is not unique to Unity, but it is one of Unity Technologies' core goals for Unity, and it is better integrated into Unity than any other game development software I have seen. However, as a game designer, you will still need to think seriously about the design differences between a game meant for a handheld phone controlled by touch, a game meant to run on a personal computer controlled by mouse and keyboard, or a game built for virtual or augmented reality, so you will usually have slightly different code for the different platforms.

C# Is Managed Code

More traditional compiled languages such as BASIC, C++, and Objective-C require programmers to directly manage memory, obliging a programmer to manually allocate and de-allocate memory any time she creates or destroys a variable.[1] If a programmer doesn't manually de-allocate RAM in these languages, her programs will have a "memory leak" and eventually allocate all of the computer's RAM, causing it to crash.

[1] Memory allocation is the process of setting aside a certain amount of random-access memory (RAM) in the computer to enable it to hold a chunk of data. While computers now often have hundreds of gigabytes (GB) of hard drive space, they still usually have less than 20GB of RAM. RAM is *much* faster that hard drive memory, so all applications pull assets like images and sounds from the hard drive, allocate some space for them in RAM, and then store them in RAM for fast access.

Luckily for us, C# is *managed code*, which means that the allocation and de-allocation of memory is handled automatically.[2] You can still cause memory leaks in managed code, but it is more difficult to do so accidentally.

[2] One disadvantage of managed code is that it makes it very difficult for you to control exactly when memory is deallocated and reclaimed. This can sometimes lead to a very brief hitch in the frame

rate of a game on less powerful devices such as cell phones, but it's usually not noticeable.

C# Is Strongly Typed

Variables are covered much more in later chapters, but there are a couple things that you should know now. First, a variable is just a named container for a value. For instance, in algebra, you may have seen an expression like this:

```
x = 5
```

In this one line, we have created a variable, named it `x`, and assigned it the value `5`. Later, if asked the value of `x + 2`, I'm sure you could tell me that the answer is `7` because you remember that `x` was holding the value `5` and know to add 2 to that value. That is exactly what variables do for you in programming.

In most interpreted languages, like JavaScript, a single variable can hold any kind of data. The variable `x` could hold the number `5` one minute, an image the next, and a sound file thereafter. This capability to hold any kind of value is what is meant when we say that a programming language is *weakly typed*.

C#, in contrast, is *strongly typed*. This means that when we initially create a variable, we must tell it at that moment what kind of value it can hold:

```
int x = 5;
```

In the preceding statement, we have created a variable, named it `x`, told it that it is exclusively allowed to hold integer values (that is, positive or negative numbers without a decimal point), and assigned it the integer value `5`. Although this might seem like it would make it more difficult to program, strong typing enables the compiler to make several optimizations and makes it possible for the authoring environment, MonoDevelop, to perform real-time syntax checking on the code you write (much like the grammar checking that is

performed by Microsoft Word). This also enables and enhances code-completion, a technology in MonoDevelop that enables it to predict the words you're typing and provide you with valid completion options based on the other code that you've written. With code-completion, if you're typing and see MonoDevelop suggest the correct completion of the word, you simply press Tab to accept the suggestion. Once you're used to this, it can save you hundreds of keystrokes every minute.

C# Is Function-Based

In the early days of programming, a program was composed of a single series of commands. These programs were run directly from beginning to end much like the directions you would give to a friend who was trying to drive to your house:

1. From school, head north on Vermont.

2. Head west on I-10 for about 7.5 miles.

3. At the intersection with I-405, take the 405 south for 2 miles.

4. Take the exit for Venice Blvd.

5. Turn right onto Sawtelle Blvd.

6. My place is just north of Venice on Sawtelle.

As authoring languages improved, repeatable sections were added to programming in the form of things like *loops* (a section of code that repeats itself) and *subroutines* (an otherwise inaccessible section of code that is jumped to, executed, and then returned from).

The development of *functional languages* allowed programmers to name chunks of code and thereby encapsulate functionality (that is, group a series of actions under a single function name). For example, if in addition to giving someone detailed directions to your house as described in the preceding list, you also asked him to pick

up some milk for you on the way, he would know that if he saw a grocery store on the way, he should: stop the car, get out, walk to find milk, pay for it, return to his car, and continue on his way to your house. Because your friend already knows how to buy milk, you just need to request that he do so rather than giving him explicit instructions for every tiny step. This could look something like this:

"Hey man, if you see a store on the way, could you please `BuySomeMilk()`?"

In this statement, you have encapsulated all of the instructions to buy milk into the single function named `BuySomeMilk()`. The same thing can be done in any functional language. When the computer is processing C# and encounters a function name followed by parentheses, it will *call* that function (that is, it will execute all of the actions encapsulated in the function). You will learn much more about functions in Chapter 24, "Functions and Parameters."

The other fantastic thing about functions is that once you have written the code for the function `BuySomeMilk()` a first time, you should never have to write it again. Even if you're working on a completely different program, you can often copy and paste functions like `BuySomeMilk()` and reuse them without having to write the whole thing again from scratch. Throughout the tutorial chapters of this book, you will be writing a C# script named `Utils.cs` that includes several reusable functions.

C# Is Object-Oriented

Many years after functions were invented, the idea of *object-oriented programming* (OOP) was created. In OOP, not only functionality but also data are encapsulated together into something called an object, or more correctly a class. This is covered extensively in Chapter 26, "Classes," but here's a metaphor for now.

Consider a group of various animals. Each animal has specific information that it knows about itself. Some examples of this data

could be its species, age, size, emotional state, level of hunger, current location, and so on. Each animal also has certain things that it can do: eat, move, breath, etc. The data about each animal are analogous to variables in code, while the actions that can be performed by the animal are analogous to functions.

Before OOP, an animal represented in code could hold information (i.e., variables) but could not perform any actions. Those actions were performed by functions that were not directly connected to the animal. A programmer could write a function named `Move()` that could move any kind of animal, but she would have to write several lines of code in that function that determined what kind of animal it was and what type of movement was appropriate for it. For example: dogs walk, fish swim, and birds fly. Any time a new animal was added to the program, `Move()` would need to be changed to accommodate the new type of locomotion, and `Move()` would thereby grow larger and more complex as each type of animal was added.

Object orientation changed all of this by introducing the ideas of *classes* and *class inheritance*. A *class* combines both variables and functions into one whole object. In OOP, instead of having a huge `Move()` function that can handle any animal, there is instead a much smaller, more specific `Move()` function attached to each animal. This eliminates the need for you to expand `Move()` every time a new species is added, and it eliminates the need for all of the type-checking of species in the non-OOP version of `Move()`. Instead, each new animal class is given its own small `Move()` function when it is created.

Object orientation also includes the concept of *class inheritance*. This enables classes to have *subclasses* that are more specific, and it allows the subclasses to either inherit or override functions in their *superclasses*. Through inheritance, a single `Animal` class could be created that included declarations of all the data types that are shared by all animals. This class would also have a `Move()` function, but it would be nonspecific. In subclasses of `Animal`, like `Dog` or `Fish`, the function `Move()` could be overridden to cause specific behavior

like walking or swimming. This is a key element of modern game programming, and it will serve you well when you want to create something like a basic `Enemy` class that is then further specified into various subclasses for each individual enemy type that you want to create.

Reading and Understanding C# Syntax

Just like any other language, C# has a specific syntax that you must follow. Take a look at these example statements in English:

• The dog barked at the squirrel.

• At the squirrel the dog barked.

• The dog at the squirrel. barked

• barked The dog at the squirrel.

Each of these English statements has the same words and punctuation, but they are in a different order, and the punctuation and capitalization is changed. Because you are familiar with the English language, it is easy for you to tell that the first is correct and the others are just wrong. Another way of examining this is to look at it more abstractly as just the parts of speech:

• [Subject] [verb] [object].

• [Object] [subject] [verb].

• [Subject] [object]. [verb]

• [verb] [Subject] [object].

When parts of speech are rearranged like this, doing so alters the *syntax* of the sentence, and the latter three sentences are incorrect because they have *syntax errors*.

Just like any language, C# has specific syntax rules for how statements must be written. Let's examine this simple statement in detail:

```
int x = 5;
```

As explained earlier, this statement does several things:

• Declares a variable named `x` of the type `int`

Any time a statement starts with a variable type, the second word of the statement becomes the name of a new variable of that type (see the Chapter 20, "Variables and Components"). This is called *declaring* a variable.

• Defines the value of `x` to be `5`

The `=` symbol is used to *assign* values to variables, which is also called *defining* the variable. When doing so, the variable name is on the left, and the value assigned is on the right.

• Ends with a semicolon (`;`)

Every simple statement in C# must end with a semicolon (`;`). This is similar in use to the period at the end of sentences in the English language.

Note

Why not end C# statements with a period? Computer programming languages are meant to be very clear. The period is not used at the end of statements in C# because it is already in use in numbers as a decimal point (for example, the period in `3.14159`). For clarity, the only use of the semicolon in C# is to end statements.

Now, let's add a second simple statement:

```
int x = 5;
int y = x * ( 3 + x );
```

You already understand the first statement, so now we'll examine the second. The second statement does the following:

• Declares a variable named `y` of the type `int`

• Adds `3 + x` (which is 3 + 5, for a result of 8)

Just like in algebra, *order of operations* follows parentheses first, meaning that `3 + x` is evaluated first because it is surrounded by parentheses. The sum is 8 because the value of `x` was set to `5` in the previous statement. In Appendix B, "Useful Concepts," you can read the section "Operator Precedence and Order of Operations," to learn more about order of operations in C#, but the main thing to remember for your programs is that if there is *any* doubt in your head about the order in which things will occur, you should use parentheses to remove doubt (and increase the readability of your code).

• Multiplies `x * 8` (`x` is 5, so the result is 40)

If there had been no parentheses, order of operations would handle multiplication and division *before* addition and subtraction. This would have resulted in `x * 3 + 5`, which would become `5 * 3 + 5`, then `15 + 5`, and finally `20`.

• Defines the value of `y` to be `40`

• Ends with a semicolon (`;`)

This chapter finishes with a breakdown of one final couplet of C# statements. In this example, the statements are now numbered. Line numbers can make it much simpler to reference a specific line in code, and it is my hope that they will make it easier for you to read and understand the code in this book when you're typing it into your computer. The important thing to remember is that **you do not need**

to type the line numbers into MonoDevelop. MonoDevelop will automatically number (and renumber) your lines as you work:

```
1  string greeting = "Hello World!";
2  print( greeting );
```

These statements deal with *strings* (a series of characters like a word or sentence) rather than integers. The first statement (numbered `1`):

• Declares a variable named `greeting` of the type `string`

`string` is another type of variable just like `int`.

• Defines the value of `greeting` to be `"Hello World!"`

The double quotes around `"Hello World!"` tell C# that the characters in between them are to be treated as a *string literal* and not interpreted by the compiler to have any additional meaning. Putting the string literal `"x = 10"` in your code will **not** define the value of `x` to be `10` because the compiler knows to ignore all string literals between quotes.

• Ends with a semicolon (`;`)

The second statement (numbered `2`):

• Calls the function `print()`

As discussed earlier, functions are named collections of actions. When a function is *called*, the function executes the actions it contains. As you might expect, `print()` contains actions that will output a string to the Console pane. Any time you see a word in code followed by parentheses, it is either calling or defining a function. Writing the name of a function followed by parentheses calls the function, causing that functionality to execute. You'll see an example of defining a function in the next chapter.

• Passes `greeting` to `print()`

Some functions just do things and don't require parameters, but many require that you *pass* something in. Any variable placed between the parentheses of a function call is *passed* into that function as an *argument*. In this case, the string `greeting` is passed into the function `print()`, and the characters `Hello World!` are output to the Console pane.

• Ends with a semicolon (`;`)

Every simple statement ends with a semicolon.

Summary

Now that you understand a little about C# and about Unity, it's time to put the two together into your first program. The next chapter takes you through the process of creating a new Unity project, creating C# scripts, adding some simple code to those scripts, and manipulating 3D GameObjects.

Chapter 19. Hello World: Your First Program

Welcome to coding.

By the end of this chapter, you'll have created your own new project and written your first bits of code. We start with the classic "Hello World" project that has been a traditional first program since long before I started coding, and then we move on to something with more of a Unity flair to it.

Creating a New Project

Now that we've got the Unity window set up properly (in the previous chapter), let's leave AngryBots behind and make our own project. Not surprisingly, you start this by creating a new project.

Appendix A, "Standard Project Setup Procedure," contains detailed instructions that show you how to set up Unity projects for the chapters in this book. At the start of each project, you will see a sidebar like the one here. Please follow the directions in the sidebar to create the project for this chapter.

Set Up the Project for this Chapter

Following the standard project setup procedure in Appendix A, create a new Project in Unity.

- **Project name:** Hello World
- **Scene name:** (none yet)
- **C# Script names:** (none yet)

You should read the whole procedure in Appendix A, but for now, you only need to create the project. The scene and C# scripts will be created as part of this chapter.

When you create a project in Unity, you're actually making a folder that will hold all the files that comprise your project. As you can see once Unity has finished creating the project, the new project comes with an open scene containing only a Main Camera and a *Directional Light* in the Project pane. Before doing anything else, save your scene by choosing *File* > *Save Scene* from the menu bar. Unity will automatically choose the correct place to save the scene, so just name it *_Scene_0* and click Save.[1] Now your saved scene appears in the Project pane.

[1] The underscore (_) at the beginning of the scene name *_Scene_0* will cause the scene to always be sorted to the top if the Project pane. I also often change the name of *Main Camera* to *_MainCamera* to achieve the same sorting benefit in the Hierarchy pane.

Right-click anywhere in the Project pane and choose *Reveal in Finder* (or *Show in Explorer* for Windows) as shown in Figure 19.1.

Figure 19.1. The blank canvas of a new Unity project (showing Reveal in Finder in the Project pane pop-up menu)

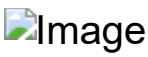


Tip

Performing a right-click on an OS X mouse or trackpad is not as straightforward as it is on a Windows PC. For information on how to do so, check out the "Right Click on Macintosh" section of Appendix B, "Useful Concepts."

Selecting *Reveal in Finder* will open a Finder window (or Explorer window) showing you the contents of your Project folder (see Figure 19.2).

Figure 19.2. The project folder for Hello World as it appears in the OS X Finder



As you can see in the image in Figure 19.2, the Assets folder holds everything that appears in the Project pane inside of Unity. In theory, you can use the Assets folder and the Project pane interchangeably (for example, if you drop an image into the Assets folder, it appears in the Project pane and vice versa), but I highly recommend working exclusively with the Project pane rather than the Assets folder. Making changes in the Assets folder directly can lead to problems if you're not careful, and the Project pane is generally safer. In addition, it is very important that you not touch the Library, ProjectSettings, or Temp folders. Doing so could cause unexpected behavior from Unity and could possibly even damage your project.

Warning

NEVER CHANGE THE NAME OF YOUR PROJECT FOLDER WHILE UNITY IS RUNNING If you change the name of the project folder while Unity is running, it will crash in a very ungraceful way. Unity does a lot of file management in the background while it's running, and changing a folder name on it will almost always cause a crash. If you want to change your project folder name, quit Unity, change the folder name, and launch Unity again.

Switch back to Unity now.

Making a New C# Script

It is time. Now you're going to write your first chunk of code. We'll be talking a lot more about C# in later chapters, but for now, just copy what you see here. Click the *Create* button in the Project pane and choose *Create* > *C# Script* (as shown in Figure 19.3). A new script will be added to the Project pane, and its name will automatically be highlighted for you to change. Name this script *HelloWorld* (make sure there's no space between the two words) and press Return to set the name.

Figure 19.3. Creating a new C# script and viewing that script in MonoDevelop



Double-click the name or the icon of the HelloWorld script to launch MonoDevelop, our C# editor. When you first open it, your script should already look exactly like the one in the image above except for line 8. Type two tabs and the code `print("Hello World");` into line 8 of your script in MonoDevelop. Make sure to spell and capitalize everything correctly and to put a semicolon (;) at the end of the line. Your HelloWorld script should now look exactly like the following code listing. In code listings throughout the book, anything new that you need to type will be in **`bold weight`**, and code that is already there is in `normal weight`.

Each line in the following code syntax also has a line number preceding it. As you can see in Figure 19.3, MonoDevelop will automatically show you line numbers for your code, so you do not need to type them yourself. There are just here in the book to help make the code listings more clear.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class HelloWorld : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
```

```
 8         print("Hello World");
 9     }
10
11     // Update is called once per frame
12     void Update () {
13
14     }
15 }
```

Note

Your version of MonoDevelop may automatically add extra spaces in some parts of the code. For example, it may have added a space between `print` and `(` in line 8 of the `Start()` function. This is okay, and you shouldn't be too concerned about it. In general, while capitalization matters tremendously to programming, spaces are more flexible. In addition, a series of several spaces (or several line brakes/returns) will be interpreted by the computer as just one, so you can add extra spaces and returns if it makes your code more readable (though extra returns may make your line numbers different from those in the code listings).

You should also not be too upset if your line numbers differ from the ones in the examples. As long as the code is the same, the line numbers don't really matter.

Now, save this script by choosing *File* > *Save* from the MonoDevelop menu bar and switch back to Unity.

This next part's a bit tricky, but you'll soon be used to it because it is so often done in Unity. Click and hold on the name of the *HelloWorld* script in the Project pane, drag it over on top of the *Main Camera* in the scene Hierarchy pane, and release the mouse button as is shown in Figure 19.4. When you are dragging the script, you will see the words *HelloWorld (Monoscript)* following the mouse, and when

you release the mouse button over *Main Camera*, the *HelloWorld (Monoscript)* words will disappear.

Figure 19.4. Attaching the HelloWorld C# script to the Main Camera in the Hierarchy pane



Dragging the HelloWorld script onto Main Camera *attaches* the script to Main Camera as a *component.* All objects that appear in the scene hierarchy pane (for example, Main Camera) are known as GameObjects, and GameObjects are made up of components. If you now click Main Camera in the Hierarchy pane, you should see HelloWorld (Script) listed as one of Main Camera's components in the Inspector pane. As you can see in Figure 19.5, the Inspector pane shows several components of the Main Camera, including its Transform, Camera, GUILayer, Flare Layer, Audio Listener, and HelloWorld (Script). GameObjects and components are covered in much more detail in later chapters.

Figure 19.5. The HelloWorld script now appears in the Inspector pane for Main Camera.



Now, just click the *Play* button (the triangle facing to the right at the top of the Unity window) and watch the magic!

The script we wrote printed *Hello World!* to the Console pane, as shown in Figure 19.6. You'll notice that it also printed *Hello World!* to the small gray bar at the bottom-left corner of the screen. This probably isn't the most magical thing that's ever happened in your life, but you have to start somewhere, and that we have. As a wise old man once said, you've taken your first step into a larger world.

Figure 19.6. Hello World! printed to the Console pane



`Start()` Versus `Update()`

Now let's try moving the `print()` function call from `Start()` to `Update()`. Go back to MonoDevelop and edit your code as shown in the following code listing.

Adding the two forward slashes (`//`) to the beginning of line 8 converts everything on line 8 that follows the slashes to a *comment.* Comments are completely ignored by the computer and are used to either disable code (as you are now doing to line 8) or to leave messages for other humans reading the code (as you can see on lines 6 and 11). Adding two slashes before a line (as we've done to line 8) is referred to as *commenting out* the line. Type the statement `print("Hello World!");` into line 13 to make it part of the `Update()` function.

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class HelloWorld : MonoBehaviour {
 5
 6     // Use this for initialization
 7     void Start () {
 8         // print("Hello World!"); // This line is now ignored.
 9     }
10
11     // Update is called once per frame
12     void Update () {
13         print("Hello World!");
14     }
15 }
```

Save the script (replacing the original version) and try pressing the Play button again. You'll see that *Hello World!* is now printed many, many times in rapid succession (see Figure 19.7). You can press the Play button again to stop execution now, and you'll see that Unity stops generating *Hello World!* messages.

Figure 19.7. `Update()` causes *Hello World!* to be printed once every frame



`Start()` and `Update()` are both special functions in Unity's version of C#. `Start()` is called once on the first frame that an object exists, whereas `Update()` is called every frame, hence the single message of Figure 19.6 versus the multiple messages of Figure 19.7. Unity has a whole list of these special functions that are called at various times. Many of them will be covered later in the book.

Tip

If you want to see each repeat of the same message only once, you can click the *Collapse* button of the Console pane (indicated by the mouse arrow in Figure 19.7), and it will ensure that each different message text appears only once.

Making Things More Interesting

Now, we're going to add more Unity style to your first program. In this example, we're going to create many, many copies of a cube. Each of these cube copies will independently bounce around and react to physics. This will demonstrate both the speed at which Unity runs and the ease with which it enables you to create content.

Start by creating a new scene. Choose *File* > *New Scene* from the menu bar. You won't notice much of a difference because we didn't really have much in _Scene_0 other than the script on the camera, but if you click the Main Camera, you'll see it no longer has a script attached, and you'll also notice that the beginning of the Unity window's title bar has changed from *_Scene_0.unity* - to *Untitled* -. As always, the first thing you should do is save this new scene.

Choose *File* > *Save Scene* from the menu bar and name this *_Scene_1*.

Now, choose *GameObject* > *3D Object* > *Cube* from the menu bar. This will place a GameObject named *Cube* in the Scene pane (and in the Hierarchy pane). If it's difficult to see Cube in the Scene pane, try double-clicking its name in the Hierarchy pane, which should focus the scene on Cube. For more information, read the "Changing the Scene View" sidebar later in this chapter that covers how to manipulate the view of the Scene pane.

If you click Cube in the Hierarchy pane, you should see it selected in the Scene pane and see its components appear in the Inspector pane (see Figure 19.8). The primary purpose of the Inspector pane is to enable you to view and edit the components that comprise any GameObject. This Cube GameObject has Transform, Mesh Filter, Box Collider, and Mesh Renderer components:

• **Transform:** The *Transform* component sets the position, rotation, and scale of the GameObject. This is the only component that is required in every GameObject. While looking at this, make sure that the Cube's Position x, y, and z values are set to 0.

• **Cube (Mesh Filter):** The *Mesh Filter* component gives the GameObject its three-dimensional shape, which is modeled as a mesh composed of triangles. 3D models in games are generally a surface that is hollow inside. Unlike a real egg (which is filled with a yolk and albumen), a 3D model of an egg would just be a mesh simulating an empty eggshell. The Mesh Filter component attaches a 3D model to the GameObject. In the case of Cube, the Mesh Filter is using a simple 3D cube model that is built into Unity, but you can also import complex 3D models into the Project pane to bring more complex meshes into your game.

• **Box Collider:** *Collider* components enable a GameObject to interact with other objects in the physics simulation that Unity runs. There are several different kinds of colliders, the most common of which are: Sphere, Capsule, Box, and Mesh (in increasing order of

computational complexity; i.e., a *Mesh Collider* is much more difficult for the computer to calculate than a *Box Collider*). A GameObject with a collider component (and no *Rigidbody* component) acts as an immovable object in space that other GameObjects can run into.

• **Mesh Renderer:** While the Mesh Filter provides the actual geometry of the GameObject, the *Mesh Renderer* component makes that geometry appear on screen. Without a renderer, nothing in Unity will appear on screen. Renderers work with the Main Camera to convert the 3D geometry of the Mesh Filter into the pixels you actually see on screen.

Figure 19.8. The new Cube GameObject visible in the Scene and Hierarchy panes



Now we're going to add one more component to this GameObject: a *Rigidbody*. With the Cube still selected in the hierarchy, choose *Component* > *Physics* > *Rigidbody* from the menu bar, and you'll see a Rigidbody component added to the Inspector:

• **Rigidbody:** The *Rigidbody* component tells Unity that we want physics to be simulated for this GameObject. This includes physical forces like gravity, friction, collisions, and drag. A Rigidbody enables a GameObject with a Collider component to move through space. Without a Rigidbody, even if the GameObject is moved by adjusting its transform, the Collider component of the GameObject will not move. You must attach a Rigidbody component to any GameObject that you want to both move and properly collide with other colliders.

Now, if you click the Play button, you'll see the box fall due to gravity.

All the physical simulations in Unity are based on the metric system. This means that:

• 1 unit of distance = 1 meter (for example, the units for the position of a transform).

• 1 unit of mass = 1 kilogram (for example, the units of mass of a Rigidbody).

• The default gravity of –9.8 = 9.8 m/s^2 in the downward (negative y) direction.

• An average human character is about 2 units (2 meters) tall.

Click the Play button again to stop the simulation.

Your scene came with a Directional Light already included. This is why the box is lit so brightly. For now, this is all you need, but we'll talk more about the various kinds of lights in later chapters.

Making a Prefab

Now, we're going to make Cube into a *prefab*. A prefab is a reusable element in a project that can be instantiated (cloned into existence) any number of times. You can think of a prefab as a mold for a GameObject, and each GameObject made from that prefab is called an *instance* of the prefab (hence the word *instantiate*). To make a prefab, click Cube in the Hierarchy pane, drag it over to the Project pane, and release the mouse button (see Figure 19.9).

Figure 19.9. Making Cube into a prefab



You'll see that a couple of things have just happened:

1. A prefab named Cube has been created in the Project pane. You can tell it's a prefab by the blue cube icon next to it. (The prefab icon is always a cube regardless of the shape of the prefab itself.)

2. The name of the Cube GameObject in the hierarchy has turned blue. If a GameObject has a blue name in the hierarchy it means that that GameObject is an instance of a prefab (which is like a copy made from the prefab mold).

Just for the sake of clarity, let's rename the Cube prefab in the Project pane to *Cube Prefab*. Click once on the Cube prefab to select it. Then click a second time to rename it (you can also try pressing Return once it's selected to rename it) and then change the name to *Cube Prefab*. You'll see that because the instance in the Hierarchy panel had the default name *Cube*, its name changes as well. If you had renamed the instance in the Hierarchy to be different from the name of the prefab, the instance name would not have been affected.

Now that we've got our prefab set up, we don't actually need the instance in the scene any more. Click *Cube Prefab* in the Hierarchy pane (not the Project pane!). Then choose *Edit > Delete* from the menu bar, and you will see the cube disappear from your scene.

It's time to get our hands dirty with some more code.

Choose *Assets > Create > C# Script* from the menu bar and rename the newly created script *CubeSpawner* (making sure that there are two capital letters and no spaces in the name). Double-click the CubeSpawner script to open MonoDevelop, add the bolded code shown here, and save it:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CubeSpawner : MonoBehaviour {
 5     public GameObject       cubePrefabVar;
 6
 7     // Use this for initialization
 8     void Start () {
 9         Instantiate( cubePrefabVar );
10     }
11
12     // Update is called once per frame
13     void Update () {
14
```

```
15     }
16 }
```

As with our previous script, this needs to be attached to something to run, so in Unity, drag the CubeSpawner script over to Main Camera just as you did previously in Figure 19.4.

Now, click the Main Camera in the Hierarchy pane. You'll see that a Cube Spawner (Script) component has been added to the Main Camera GameObject (see Figure 19.10).

Figure 19.10. The CubeSpawner script component in the Inspector pane for Main Camera



You can also see that there is a variable called *Cube Prefab Var* in this component (though it really should be cubePrefabVar, as explained in the warning below). That comes from the `"public GameObject cubePrefabVar;"` statement you typed on line 5. In general, if a variable of a script is labeled `"public"`, it will appear in the Inspector pane.

Warning

VARIABLE NAMES LOOK DIFFERENT IN THE INSPECTOR
Someone at Unity thought it would look nice to change the capitalization and spacing of variable names in the Inspector pane. I have no idea why this has lasted into the current version, but it means that your variable names like `cubePrefabVar` will incorrectly appear in the Inspector as *Cube Prefab Var*. Be careful to always refer to your variable names properly in your programming and please ignore the strange capitalization and spacing that you see in the Inspector. Throughout the book, I refer to variables by their proper name in code rather than the names that appear in the Inspector.

As you can see in the Inspector, cubePrefabVar currently has no value assigned. Click the circular target to the right of the cubePrefab variable value (as shown by the arrow cursor in Figure 19.10), this will bring up the *Select GameObject* dialog box from which you can select a prefab to be assigned to this variable. Make sure that the *Assets* tab is selected. (The Assets tab shows GameObjects in your Project pane, while the Scene tab shows GameObjects in your Hierarchy.) Double-click CubePrefab to select it (see Figure 19.11).

Figure 19.11. Selecting *Cube Prefab* for the cubePrefabVar variable of the CubeSpawner script



Now, you can see in the Inspector that the value of cubePrefabVar is *Cube Prefab* from the Project pane. To double-check this, click the value *Cube Prefab* in the Inspector, and you'll see that *Cube Prefab* is highlighted yellow in the Project pane.

Click the Play button.

You'll see that a single *Cube Prefab(Clone)* GameObject is instantiated in the Hierarchy. Just like we saw in the Hello World script, the `Start()` function is called once, and it creates a single instance (or clone) of the Cube Prefab. Now, switch to MonoDevelop, comment out the `Instantiate( cubePrefabVar )` call on line 9 in the `Start()` function and add an `Instantiate( cubePrefabVar );` statement to line 14 in the `Update()` function, as shown in the following code.

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CubeSpawner : MonoBehaviour {
 5     public GameObject        cubePrefabVar;
 6
 7     // Use this for initialization
```

```
 8     void Start () {
 9        // Instantiate( cubePrefabVar );
10     }
11
12     // Update is called once per frame
13     void Update () {
14        Instantiate( cubePrefabVar );
15     }
16 }
```

Save the CubeSpawner script, switch back to Unity, and press Play again. As shown in Figure 19.12, this will give you cubes galore.

Figure 19.12. Creating a new instance of the CubePrefab every Update() quickly adds up to a lot of cubes!



This is an example of the power of Unity. Very quickly, we were able to get up to speed and make something cool and interesting. Now, let's add some more objects to the scene for the cubes to interact with. Press Play again to stop playback.

In the Hierarchy, click the *Create* pop-up menu and choose *3D Object* > *Cube*. Rename this cube *Ground*. With a GameObject selected in the Scene pane or Hierarchy pane, pressing the W, E, or R keys will allow you to translate (move), rotate, or scale the GameObject. This will show *gizmos* (the arrows, circles, and such shown around the cube in Figure 19.13) around Ground. In translation (W) mode, clicking and dragging on one of the arrows will move the cube exclusively along the axis of that arrow (x, y, or z). The colored elements of the rotation and scale gizmos lock the transformation to a specific axis in similar ways. See the "Changing the Scene View" sidebar for information about how to use the hand tool shown in Figure 19.13.

Figure 19.13. The translate (position), rotate, and scale gizmos. Q, W, E, R, and T are the keys that select each tool. The T tool is used

for positioning 2D and GUI GameObjects.



???Jeremy—Above, should be 19.13, right, not 19.15 (or 18.15). Thanks, Chris

---

Changing the Scene view

The first tool on the toolbar shown in Figure 19.13—known as the *hand tool*—is used to manipulate the view shown in the Scene pane. The Scene pane has its own invisible *scene camera* that is different from the Main Camera in the Hierarchy. The hand tool has several different abilities. Select the hand tool (by either clicking it or pressing Q on your keyboard) and try the following:

• Left-dragging (that is, clicking and dragging using the left mouse button) in the Scene pane will move the position of the scene camera without changing the position of any of the objects in the scene. To be technical, the scene camera is moved in a plane perpendicular to the direction that the camera is facing (that is, perpendicular to the camera's forward vector).

• Right-dragging in the Scene pane will rotate the scene camera to look in the direction of your drag. The scene camera stays in the same position when right-dragging.

• Holding the Option key (or Alt key on a PC) will change the cursor over the Scene pane from a hand to an eye, and left-dragging with the Option key held will cause the Scene view to rotate around objects in the Scene pane (this is known as orbiting the camera around the scene). When Option-left-dragging, the position of the scene camera changes, but the location that the scene camera is looking at does not.

• Scrolling with the scroll wheel on your mouse will cause the scene camera to zoom in and out of the scene. Zooming can also be done

by Option-right-dragging in the Scene pane.

The best way to get a feel for the hand tool is to try moving around the scene using the different methods described in this sidebar. After you have played with it a little, it should become second nature to you.

Try moving Ground to a Y position of -4 and setting its scale in the X and Z dimensions to 10. Throughout the book, I will suggest positions, rotations, and scales using this format.



Ground here is the name of the GameObject, and *(Cube)* is the type of the GameObject. *P:[0,-4,0]* means to set the X position to 0, the Y position to -4, and the Z position to 0. Similarly, *R:[0,0,0]* means to keep the X, Y, and Z rotations set to 0. *S:[10,1,10]* means to set the X scale to 10, the Y scale to 1, and the Z scale to 10. You can either use the tools and gizmos to make these changes or just type them into the Transform component of the GameObject's Inspector.

Feel free to play around with this and add more objects. The instances of Cube Prefab will bounce off of the *static* objects that you put into the scene (see Figure 19.14). As long as you don't add a Rigidbody to any of the new shapes, they should be static (i.e., solid and immovable). When you're done, be sure to save your scene!

Figure 19.14. The scene with static shapes added



Summary

In about 20 pages, you've gone from nothing to having a working Unity project with a little programming in it. Admittedly, this project was pretty small, but I hope that it has served to show you the raw

speed at which Unity can operate as well as the speed at which you can get something running in Unity.

The next chapter will continue your introduction to C# and Unity by introducing you to variables and increasing your knowledge of the most common components that can be added to a GameObject.

Chapter 20. Variables and Components

This chapter introduces you to the many variable and component types used throughout Unity C# programming. By the end of the chapter, you will understand many of the common types of C# variables and some important variable types that are unique to Unity.

This chapter also introduces you to Unity's GameObjects and components. Any object in a Unity scene is a GameObject, and the components that comprise them enable everything from the positioning of a GameObject to physics simulation, special effects, displaying a 3D model on screen, and character animation..

Introducing Variables

To recap a bit of Chapter 18, "Introducing our Language: C#," a *variable* is just a name that can be defined to be equal to a specific value. This is actually a concept that comes from the study of algebra. In algebra, for instance, you can be given the definition:

x = 5

This *defines* the *variable* `x` to be equal to the *value* 5. In other words, it assigns the value 5 to the name `x`. Then, if later, you encounter the definition:

y = x + 2

Then you now know that the value of the variable `y` is 7 (because `x` = 5 and 5 + 2 = 7). `x` and `y` are called *variables* because their value can be redefined at any time, and the order in which these definitions

occur matters. Take a look at these definitions. (*Comments* are included after two slashes [*//*] in the following lines to help explain what each statement is doing.)

```
x = 10      // x is now equal to the value 10
y = x – 4   // y is now 6 because 10-4 = 6
x = 12      // x is now equal to the value 12, but y is still 6
z = x + 3   // z is now 15 because 12+3 = 15
```

After this sequence of definitions, the values assigned to `x`, `y`, and `z` are 12, 6, and 15, respectively. As you can see, even though `x` changed value, `y` was not affected because `y` is defined as the value 6 before `x` is assigned the new value 12, and `y` is not retroactively affected.

Strongly Typed Variables in C#

Instead of being able to be assign any kind of value to any variable, C# variables are *strongly typed*, meaning that they can only accept a specific type of value. This is necessary because the computer needs to know how much space in memory to allocate to each variable. A large image can take up many megabytes or even gigabytes of space, while a Boolean value (which can only hold either a 1 or a 0) only really requires a single bit. Even just a single megabyte is equivalent to 8,388,608 bits!

Declaring and Defining Variables in C#

In C#, a variable must be both declared and defined for it to have a usable value.

Declaring a variable creates it and gives it a name and type. However, this does not give the variable a value.

```
bool   bravo;
int    india;
```

```
float   foxtrot;
char    charlie;
```

Defining a variable gives that variable a value. Here are some examples of how to use these declared variables:

```
bravo = true;
india = 8;
foxtrot = 3.14f; // The f makes this numeric literal a float, as described later
charlie = 'c';
```

Whenever you write a specific value in your code (e.g., `true`, `8`, or `'c'`), that specific value is called a *literal*. In the previous code listing, `true` is a bool literal, `8` is an int literal, `3.14f` is a float literal, and `'c'` is a char literal. By default, MonoDevelop shows these literals in a bright orange color (though `true` is colored teal for esoteric reasons), and each variable type has certain rules about how its literals are represented. Check out each of the variable types in the following sections for more information on this.

Declaration Before Definition

You must first declare a variable before you can define it, although this is often done on the same line:

```
string   sierra = "Mountain";
```

In general, Unity will complain and throw a compiler error if you try to access (i.e., read) a variable that has been declared but has not yet been defined.

Important C# Variable Types

Several different types of variables are available to you in C#. Here are a few important ones that you'll encounter frequently. All of these basic C# variable types begin with a lowercase letter, whereas most

Unity data types begin with an uppercase letter. For each, I've listed information about the variable type and an example of how to declare and define the variable.

bool: A 1-Bit True or False Value

The term *bool* is short for Boolean. At their heart, all variables are composed of bits that can either be set to true or false. A bool is 1 bit in length, making it the smallest possible variable.[1] Bools are extremely useful for logic operations like `if` statements and other conditionals, which are covered in the next two chapters. In C#, bool literals are limited to the lowercase keywords `true` and `false`:

[1] Because of the way that C# handles memory, a single bool actually takes up more than 1 bit of memory, but a bool only requires a single bit of memory.

```
bool verified = true;
```

int: A 32-Bit Integer

Short for integer, an *int* can store a single integer number (integers are numbers without any fractional value like `5`, `2`, & `-90`). Integer math is very accurate and *very* fast. An int in Unity can store a number between –2,147,483,648 and 2,147,483,647 with 1 bit used for the positive or negative sign of the number and 31 bits used for the numerical value. An int can hold any integer value between these two numbers (inclusive):

```
int nonFractionalNumber = 12345;
```

float: A 32-bit Decimal Number

A floating-point number,[2] or *float*, is the most common form of decimal number used in Unity. It is called "floating point" because it is stored using a system similar to *scientific notation*. Scientific

notation is the representation of numbers in the form $a \cdot 10^b$ (for example, 300 would be written $3 \cdot 10^2$, and 12,345 would be written $1.2345 \cdot 10^4$). Floating-point numbers are stored in a similar format as $a \cdot 2^b$. When storing numbers this way in memory, 1 bit represents whether the number is positive or negative, 23 bits are allocated to the significand (the *a* part of the number), and 8 are allocated to exponent to which the number is raised or lowered (the *b* part). This means that there will be significant gaps in the precision of very large numbers and any number between 1 and –1 that is difficult to represent as a power of 2. For instance, there is no way to accurately represent 1/3 using a float.

[2] http://en.wikipedia.org/wiki/Floating_point

Most of the time, the imprecise nature of floats doesn't matter much in your games, but it can cause small errors in things like collision detection; so in general, keeping elements in your game larger than 1 unit and smaller than several thousand units in size will make collisions a little more accurate. Float literals must be either a whole number or a decimal number followed by an `f`. This is because C# assumes that any decimal literal without a trailing `f` is a *double* (which is a float data type with double the precision) instead of the single-precision floats that Unity uses. Floats are used in all built-in Unity functions instead of doubles to bring about the fastest possible calculation, though this comes at the expense of accuracy:

float notPreciselyOneThird = 1.0f/3.0f;

Tip

If you see the following compile-time error in your code

error CS0664: Literal of type double cannot be implicitly converted to type 'float'. Add suffix 'f' to create a literal of this type

it means that somewhere you have forgotten to add the `f` after a float literal.

---

char: A 16-Bit Single Character

A *char* is a single character represented by 16 bits of information. Chars in Unity's C# use Unicode[3] values for storing characters, enabling the representation of over 110,000 different characters from over 100 different character sets and languages (including, for instance, all the characters in Simplified Chinese). A char literal is surrounded by single-quote marks (a.k.a. apostrophes):

[3] http://en.wikipedia.org/wiki/Unicode

```
char theLetterA = 'A';
```

string: A Series of 16-Bit Characters

A *string* is used to represent everything from a single character to the text of an entire book. The theoretical maximum length of a string in C# is over 2 billion characters, but most computers will encounter memory allocation issues long before that limit is reached. To give some context, there are a little over 175,000 characters in the full version of Shakespeare's play *Hamlet*,[4] including stage directions, line breaks, and so on. This means that Hamlet could be repeated over 12,000 times in a single string. A string literal is surrounded by double-quote marks:

[4] http://shakespeare.mit.edu/hamlet/full.html

```
string theFirstLineOfHamlet = "Who's there?";
```

It's also possible to access the individual chars of a string using *bracket access*:

char theCharW = theFirstLineOfHamlet[0]; // W is the 0th char in the string
char theChart = theFirstLineOfHamlet[6]; // t is the 6th char in the string

Placing a number in brackets after the variable name retrieves the character in that position of the string (without affecting the original string). When using bracket access, counting starts with the number 0; so in the preceding example, `W` is the 0th character of the first line of *Hamlet*, and `t` is the 6th character. You will encounter bracket access much more in Chapter 23, "Lists and Arrays."

Tip

If you see any of the following compile-time errors in your code

error CS0029: Cannot implicitly convert type 'string' to 'char'
error CS0029: Cannot implicitly convert type 'char' to 'string'
error CS1012: Too many characters in character literal
error CS1525: Unexpected symbol '<internal>'

it usually means that somewhere you have accidentally used double quotes (`" "`) for a char literal or single quotes (`' '`) for a string literal. String literals always require double quotes, and char literals always require single quotes.

---

class: The Definition of a New Variable Type

A *class* defines a new type of variable that can be best thought of as a collection of both variables and functionality. All the Unity variable types and components listed in the "Important Unity Variable Types" section later in this chapter are examples of classes. Chapter 26, "Classes," covers classes in much greater detail.

The Scope of Variables

In addition to variable type, another important concept for variables is *scope*. The scope of a variable refers to the range of code in which the variable exists and is understood. If you declare a variable in one part of your code, it might not have meaning in another part. This is a complex issue that will be covered throughout this book. If you want to learn about it progressively, just read the book in order. If you want to get a lot more information about variable scope right now, you can read the section “Variable Scope” in Appendix B, “Useful Concepts.”

Naming Conventions

The code in this book follows a number of rules governing the naming of variables, functions, classes, and so on. Although none of these rules are mandatory, following them will make your code more readable not only to others who try to decipher it but also to yourself if you ever need to return to it months later and hope to understand what you wrote. Every coder follows slightly different rules—my personal rules have even changed over the years—but the rules I present here have worked well for both me and my students, and they are consistent with most C# code that I’ve encountered in Unity:

1. Use *camelCase* for pretty much everything (see the camelCase sidebar).

Camel Case

camelCase is a common way of writing variable names in programming. It allows the programmer or someone reading her code to easily parse long variable names. Here are some examples:

• aVeryLongNameThatIsEasierToReadBecauseOfCamelCase

• variableNamesStartWithALowerCaseLetter

classNamesStartWithACapitalLetterThe key feature of camelCase is that it allows multiple words to be combined into one with a medial capital letter at the beginning of each original word. It is named camelCase because it looks a bit like the humps on a camel's back.

---

2. Variable names should start with a lowercase letter (e.g., `someVariableName`).

3. Function names should start with an uppercase letter (e.g., `Start()`, `Update()`).

4. Class names should start with an uppercase letter (e.g., `GameObject`, `ScopeExample`).

5. Private variable names can start with an underscore (e.g., `_hiddenVariable`).

6. Static variable names can be all caps with snake_case (e.g., `NUM_INSTANCES`). As you can see, snake_case combines multiple words with an underscore in between them.

For your later reference, this information is repeated and expanded in the "Naming Conventions" section of Appendix B.

Important Unity Variable Types

Unity has a number of variable types that you will encounter in nearly every project. All of these variable types are actually classes and follow Unity's naming convention that all class types start with an uppercase letter. For each of the Unity variable types, you will see information about how to create a new *instance* of that class (see the sidebar on class instances) followed by listings of important variables and functions for that data type. For most of the Unity classes listed in this section, the variables and functions are split into two groups:

• **Instance variables and functions:** These variables and functions are tied directly to a single instance of the variable type. If you look at the *Vector3* information that follows, you will see that `x`, `y`, `z`, and `magnitude` are all instance variables of Vector3, and each one is accessed by using the name of a Vector3 variable, a period, and then the name of the instance variable (for example, `position.x`). Each Vector3 instance can have different values for these variables. Similarly, the `Normalize()` function acts on a single instance of Vector3 and sets the `magnitude` of that instance to one.

• **Static class variables and functions:** Static variables are tied to the class definition itself rather than being tied to an individual instance. These are often used to store information that is the same across all instances of the class (for example, `Color.red` is always the same red color) or to act on multiple instances of the class without affecting either (for example, `Vector3.Cross( v3a, v3b )` is used to calculate the cross product of two Vector3s and return that value as a new Vector3 without changing either `v3a` or `v3b`).

For more information on any of these Unity types, check out the Unity documentation links referenced in the footnotes.

Class Instances And Static Functions

Just like the prefabs that you saw in Chapter 19, "Hello World: Your First Program," classes can also have *instances*. An instance of any class (also known as a *member* of the class) is a data object that is of the type defined by the class.

For example, you could define a class `Human`, and everyone you know would be an instance of that class. Several functions are defined for all humans (for example, `Eat()`, `Sleep()`, `Breathe()`).

Just as you differ from all other humans around you, each instance of a class differs from the other instances. Even if two instances have perfectly identical values, they are stored in different locations in computer memory and seen as two distinct objects. (To continue

the human analogy, you could think of them as identical twins.) Class instances are referred to by *reference*, not value. This means that if you are comparing two class instances to see whether they are the same, the thing that is compared is their *location in memory*, not their values (just as two identical twins have different names).

It is, of course, possible to reference the same class instance using different variables. Just as the person I might call “daughter” would also be called “granddaughter” by my parents, a class instance can be assigned to any number of variable names yet still be the same data object, as is shown in the following code:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 // Defining the class Human
 5 public class Human {
 6     public string    name;
 7     public Human     partner;
 8 }
 9
10 public class Family : MonoBehaviour {
11     // public variable declaration
12     public Human husband;
13     public Human wife;
14
15     void Start() {
16         // Initial state
17         husband = new Human();
18         husband.name = "Jeremy Gibson";
19         wife = new Human();
20         wife.name = "Melanie Schuessler";
21
22         // My wife and I get married
23         husband.partner = wife;
24         wife.partner = husband;
25
```

```
26        // We change our names
27        husband.name = "Jeremy Gibson Bond";
28        wife.name = "Melanie Schuessler Bond";
29
30        // Because wife.partner refers to the same instance as
husband,
31        //  the name of wife.partner has also changed
32        print(wife.partner.name);
33        // prints "Jeremy Gibson Bond"
34    }
35 }
```

It is also possible to create *static functions* on the class `Human` that are able to act on one or more instances of the class. The following static function `Marry()` allows you to set two humans to be each other's partner with a single function as is shown in the following code.

```
35  // This replaces line 35 from the previous code listing,
deleting the "}"
36
37    static public void Marry(Human h0, Human h1) {
38        h0.partner = h1;
39        h1.partner = h0;
40    }
41 }
```

With this function, it would now be possible to replace lines 23 and 24 from the initial code listing with the single line `Human.Marry( wife, husband );` . Because `Marry()` is a static function, it can be used almost anywhere in your code. You will learn more about static functions and variables later in the book.

Vector3: A Collection of Three Floats

Vector3[5] is a very common data type for working in 3D. It is used most commonly to store the three-dimensional position of objects in Unity. Follow the footnote for more detailed information about Vector3s.

[5] http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html

Vector3 position = new Vector3(0.0f, 3.0f, 4.0f); // Sets the x, y, & z values

Vector3 Instance Variables and Functions

As a class, each Vector3 instance also contains a number of useful built-in values and functions:

```
print( position.x ); // 0.0, The x value of the Vector3
print( position.y ); // 3.0, The y value of the Vector3
print( position.z ); // 4.0, The z value of the Vector3
print( position.magnitude ); // 5.0, The distance of the Vector3 from 0,0,0
                             //  Magnitude is another word for "length".
position.Normalize(); // Sets the magnitude of position to 1, meaning that the
                      //  x, y, & z values of position are now [0.0, 0.6, 0.8]
```

Vector3 Static Class Variables and Functions

In addition, several static class variables and functions are associated with the Vector3 class itself:

```
print( Vector3.zero );    // (0,0,0), Shorthand for: new Vector3( 0, 0, 0 )
print( Vector3.one );     // (1,1,1), Shorthand for: new Vector3( 1, 1, 1 )
print( Vector3.right );   // (1,0,0), Shorthand for: new Vector3( 1,
```

0, 0)
print(Vector3.up); // (0,1,0), Shorthand for: new Vector3(0, 1, 0)
print(Vector3.forward); // (0,0,1), Shorthand for: new Vector3(0, 0, 1)
Vector3.Cross(v3a, v3b);// Computes the cross product of the two Vector3s
Vector3.Dot(v3a, v3b); // Computes the dot product of the two Vector3s

This is only a sampling of the fields and methods affiliated with Vector3. To find out more, check out the Unity documentation referenced in the footnote.

Color: A Color with Transparency Information

The *Color*[6] variable type can store information about a color and its transparency (alpha value). Colors on computers are mixtures of the three primary colors of light: red, green, and blue. These are different from the primary colors of paint you may have learned as a child (red, yellow, and blue) because color on a computer screen is *additive*, rather than *subtractive*. In a subtractive color system like paint, mixing more and more different colors together will move the mixed color toward black (or a really dark, muddy brown). By contrast, in an additive color system (like a computer screen, theatrical lighting design, or HTML colors on the Internet), adding more and more colors together will get brighter and brighter until the final mixed color is eventually white. The red, green, and blue components of a color in C# are stored as floats that range from `0.0f` to `1.0f` with `0.0f` representing none of that color channel and `1.0f` representing as much of that color channel as possible:

[6] http://docs.unity3d.com/Documentation/ScriptReference/Color.html

// Colors are defined by floats for the Red, Green, Blue, Alpha channels
Color darkGreen = new Color(0f, 0.25f, 0f); // If no alpha info is

passed in,

// the alpha value is assumed to
// be 1 (fully opaque)
Color darkRedTranslucent = new Color(0.25f, 0f, 0f, 0.5f);

As you can see, there are two different ways to define a color, one with three parameters (red, green, and blue) and one with four parameters (red, green, blue, and alpha).[7] The alpha value sets the transparency of the color. A color with an alpha of `0.0f` is fully transparent, and a color with an alpha of `1.0f` is fully opaque.

[7] The ability of the new `Color()` function to accept either three or four different arguments is called *function overloading*, and you can read more about it in Chapter 24, "Functions and Parameters."

Color Instance Variables and Functions

Each channel of a color can be referenced through instance variables:

print(Color.yellow.r); // 1, The red value of the yellow Color
print(Color.yellow.g); // 0.92f, The green value of the yellow Color
print(Color.yellow.b); // 0.016f, The blue value of the yellow Color
print(Color.yellow.a); // 1, The alpha value of the yellow Color

Color Static Class Variables and Functions

Several common colors are predefined in Unity as static class variables:

// Primary Colors: Red, Green, and Blue
Color.red = new Color(1, 0, 0, 1); // Solid red
Color.green = new Color(0, 1, 0, 1); // Solid green
Color.blue = new Color(0, 0, 1, 1); // Solid blue

```
// Secondary Colors: Cyan, Magenta, and Yellow
Color.cyan    = new Color(0, 1, 1, 1); // Cyan, a bright greenish blue
Color.magenta = new Color(1, 0, 1, 1); // Magenta, a pinkish purple
Color.yellow  = new Color(1, 0.92f, 0.016f, 1); // A nice-looking yellow
// As you can imagine, a standard yellow would be new Color(1,1,0,1), but in
//  Unity's opinion, this color looks better.

// Black, White, and Clear
Color.black   = new Color(0, 0, 0, 1); // Solid black
Color.white   = new Color(1, 1, 1, 1); // Solid white
Color.gray    = new Color(0.5f, 0.5f, 0.5f, 1) // Gray
Color.grey    = new Color(0.5f, 0.5f, 0.5f, 1) // British spelling of gray
Color.clear   = new Color(0, 0, 0, 0); // Completely transparent
```

Quaternion: Rotation Information

Explaining the inner workings of the *Quaternion*[8] class is far beyond the scope of this book, but you will be using them often to set and adjust the rotation of objects through the Quaternion `GameObject.transform.rotation`, which is part of every GameObject. Quaternions define rotations in a way that avoids gimbal lock, a problem with standard x, y, z (or Euler, pronounced “oiler”) rotations where one axis can align with another and limit rotation possibilities. Most of the time, you will be defining a Quaternion by passing in Euler rotations and allowing Unity to convert them into the equivalent Quaternion:

[8] http://docs.unity3d.com/Documentation/ScriptReference/Quaternion.html

```
Quaternion lookUp45Deg = Quaternion.Euler( -45f, 0f, 0f );
```

In cases like this, the three floats passed into `Quaternion.Euler()` are the number of degrees to rotate around the x, y, and z axes (respectively colored red, green, and blue in Unity). GameObjects, including the Main Camera in a scene, are initially oriented to be looking down the positive z-axis. The rotation in the preceding code would rotate the camera –45 degrees around the red x-axis, causing it to then be looking up at a 45° angle relative to the positive z-axis. If that last sentence was confusing, don't worry about it too much right now. Later, you can try going into Unity and changing the x, y, and z rotation values in the Transform Inspector for a GameObject and see how it alters the object's orientation.

Quaternion Instance Variables and Functions

You can also use the instance variable `eulerAngles` to cause a quaternion to return its rotation information to you in Euler angles as a Vector3:

print(lookUp45Deg.eulerAngles); // (-45, 0, 0), the Euler rotation

Mathf: A Library of Mathematical Functions

Mathf[9] isn't really a variable type as much as it's a fantastically useful library of math functions. All of the variables and functions attached to Mathf are static; you cannot create an instance of Mathf. There are far too many useful functions in the Mathf library to list here, but a few include the following:

[9] http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html

```
Mathf.Sin(x);        // Computes the sine of x
Mathf.Cos(x);        // .Tan(), .Asin(), .Acos(), & .Atan() are also available
Mathf.Atan2( y, x );  // Gives you the angle to rotate around the z-axis to
                      //   change something facing along the x-axis to
```

face

// instead toward the point x, y.[10]

print(Mathf.PI); // 3.141593; the ratio of circumference to diameter
Mathf.Min(2, 3, 1); // 1, the smallest of the three numbers (float or int)
Mathf.Max(2, 3, 1); // 3, the largest of the three numbers (float or int)
Mathf.Round(1.75f); // 2, rounds up or down to the nearest number
Mathf.Ceil(1.75f); // 2, rounds up to the next highest integer number
Mathf.Floor(1.75f); // 1, rounds down to the next lowest integer number
Mathf.Abs(-25); // 25, the absolute value of -25

10
http://docs.unity3d.com/Documentation/ScriptReference/Mathf.Atan2.html

Screen: Information about the Display

Screen[11] is another library like Mathf that can give you information about the specific computer screen that your Unity game is using. This works regardless of device, so screen will give you accurate info whether you're on a PC, OS X, an iOS device, or an Android tablet:

11
http://docs.unity3d.com/Documentation/ScriptReference/Screen.html

```
print( Screen.width );     // Prints the width of the screen in pixels
print( Screen.height );    // Prints the height of the screen in pixels
Screen.showCursor = false; // Hides the cursor
```

SystemInfo: Information about the Device

SystemInfo[12] will tell you specific information about the device on which the game is running. It includes information about operating system, number of processors, graphics hardware, and more. I recommend following the link in the footnote to learn more:

[12] http://docs.unity3d.com/Documentation/ScriptReference/SystemInfo.html

```
print( SystemInfo.operatingSystem ); // Mac OS X 10.8.5, for example
```

GameObject: The Type of Any Object in the Scene

GameObject[13] is the base class for all entities in Unity scenes. Anything you see on screen in a Unity game is a subclass of the `GameObject` class. GameObjects can contain any number of different *components*, including all of those referenced in the next section: "Unity GameObject Components." However, GameObjects also have a few important variables beyond what is covered there:

[13] http://docs.unity3d.com/Documentation/ScriptReference/GameObject.html

```
GameObject gObj = new GameObject("MyGO"); // Creates a new GameObject named MyGO
print( gObj.name );   // MyGO, the name of the GameObject gObj
Transform trans = gObj.GetComponent<Transform>(); // Defines trans to be a
                                                  //  reference to the Transform
                                                  //  Component of gObj
Transform trans2 = gObj.transform; // A shortcut to access the same Transform
gObj.SetActive(false);   // Makes gObj inactive, rendering it invisible and
                         //  preventing it from running code.
```

The *method*[14] `gObj.GetComponent<Transform>()` shown here is of particular importance because it can enable you to access any of the components attached to a GameObject. You will sometimes see methods with angle brackets `<>` like `GetComponent<>()`. These are called *generic methods* because they are a single method designed to be used with many different data types. In the case of `GetComponent<Transform>()`, the data type is `Transform`, which tells `GetComponent<>()` to find the Transform component of the GameObject and return it to you. This can also be used to get any other component of the GameObject by typing that component type inside the angle brackets instead of Transform. Examples include the following:

[14] "Function" and "method" have the same basic meaning. The only difference is that "function" is the word for a standalone function while "method" refers to a function that is part of a class.

```
Renderer rend = gObj.GetComponent<Renderer>(); // Gets the Renderer component
Collider coll = gObj.GetComponent<Collider>(); // Gets the Collider component
HelloWorld hwInstance = gObj.GetComponent<HelloWorld>();
```

As shown in the third line of the preceding code listing, `GetComponent<>()` can also be used to return the instance of any C# class that you've attached to the GameObject. If there were an instance of the HelloWorld C# script class attached to gObj, then `gObj.Getcomponent<HelloWorld>()` would return it. This technique is used several times throughout this book.

Unity GameObjects and Components

As mentioned in the previous section, all on-screen elements in Unity are *GameObjects*, and all GameObjects contain one or more components (a Transform component is *always* included). When you select a GameObject in either the Hierarchy pane or the Scene pane

of Unity, the components of that GameObject are displayed in the Inspector pane, as shown in Figure 20.1.

Figure 20.1. The Inspector pane showing various important components.



Transform: Position, Rotation, and Scale

Transform[15] is a mandatory component that is present on all GameObjects. Transform handles critical GameObject information like *position* (the location of the GameObject), *rotation* (the orientation of the GameObject), and *scale* (the size of the GameObject). Though the information is not displayed in the Inspector pane, Transform is also responsible for the parent/child relationships in the Hierarchy pane. When one object is the child of another, it moves with that parent object as if attached to it.

[15] http://docs.unity3d.com/Documentation/Components/class-Transform.html

MeshFilter: The Model You See

A *MeshFilter*[16] component attaches a 3D mesh in your Project pane to a GameObject. To see a model on screen, the GameObject must have both a MeshFilter that handles the actual 3D mesh information and a MeshRenderer that combines that mesh with a shader or material and displays the image on screen. The MeshFilter creates a skin or surface for a GameObject, and the MeshRenderer determines the shape, color, and texture of that surface.

[16] http://docs.unity3d.com/Documentation/Components/class-MeshFilter.html

Renderer: Allows You to See the GameObject

A *Renderer*[17] component—in most cases, a MeshRenderer—allows you to see the GameObject in the Scene and Game panes. The MeshRenderer requires a MeshFilter to provide 3D mesh data as well as at least one Material if you want it to look like anything other than an ugly magenta blob (Materials apply textures to objects, and when no Material is present, Unity defaults to solid magenta to alert you to the problem). Renderers bring the MeshFilter, the Material(s), and lighting together to show the GameObject on screen.

[17] http://docs.unity3d.com/Documentation/Components/class-MeshRenderer.html

Collider: The Physical Presence of the GameObject

A *Collider*[18] component enables a GameObject to have a physical presence in the game world and collide with other objects. There are four different kinds of Collider components in Unity:

[18] http://docs.unity3d.com/Documentation/Components/comp-DynamicsGroup.html

• Sphere Collider:[19] The fastest collision shape. A ball or sphere.

[19] http://docs.unity3d.com/Documentation/Components/class-SphereCollider.html

• Capsule Collider:[20] A pipe with spheres at each end. The second fastest type.

[20] http://docs.unity3d.com/Documentation/Components/class-CapsuleCollider.html

• Box Collider:[21] A rectangular solid. Useful for crates and other boxy things.

[21] http://docs.unity3d.com/Documentation/Components/class-BoxCollider.html

• Mesh Collider:[22] A collider formed from a 3D mesh. Although useful and accurate, they are much, much slower than any of the other three. Also, only Mesh Colliders with Convex set to true can collide with other Mesh Colliders.

[22] http://docs.unity3d.com/Documentation/Components/class-MeshCollider.html

Physics and collision are handled in Unity via the NVIDIA PhysX engine. Although this does usually provide very fast and accurate collisions, be aware that all physics engines have limitations, and even PhysX will sometimes have issues with fast-moving objects or thin walls.

Colliders are covered in much more depth later in this book. You can also learn more about them from the Unity documentation.

Rigidbody: The Physics Simulation

The *Rigidbody*[23] component controls the physics simulation of your GameObject. The Rigidbody component simulates acceleration and velocity every *FixedUpdate* (generally every 50th of a second) to update the position and rotation of the Transform component over time. It also uses the Collider component to handle collisions with other GameObjects. The Rigidbody component can also model things like gravity, drag, and various forces like wind and explosions. Set `isKinematic` to true if you want to directly set the position of your GameObject without using the physics provided by Rigidbody.

[23] http://docs.unity3d.com/Documentation/Components/class-Rigidbody.html

For the position of a Collider component to move with its GameObject, the GameObject *must* have a Rigidbody. Otherwise—

as far as Unity's PhysX physics simulation is concerned—the collider will not move. In other words, if a Rigidbody is not attached, the GameObject will appear to move across the screen, but in PhysX, the location of the Collider component will not be updated, and therefore the physical presence of the GameObject will remain in the same location.

Script: The C# Scripts That You Write

All C# scripts are also GameObject components. One of the benefits of scripts being components is that you can attach more than one script to each GameObject, a capability that we will take advantage of in some of the tutorials in Part III of this book. Later in the book, you will read much more about Script components and how to access them.

Warning

VARIABLE NAMES WILL CHAGE IN THE INSPECTOR In Figure 20.1, you can see that the name of the script is *Scope Example (Script)*, but that breaks the naming rules for classes, because class names cannot have spaces in them.

The actual script name in my code is a single word in camelCase: `ScopeExample`. I'm not sure why exactly, but in the Inspector, the spelling of class and variable names is changed from their actual spelling in the C# scripts you write by the following rules:

• The class name `ScopeExample` becomes *Scope Example (Script)*.

• The variable `trueOrFalse` becomes *True Or False*.

• The variable `graduationAge` becomes *Graduation Age*.

• The variable `goldenRatio` becomes *Golden Ratio*.

This is an important distinction, and it has confused some of my students in the past. Even though the names appear differently in the Inspector, the variable names in your code have not been changed.

Summary

This was a long chapter with a lot of information in it, and you may need to read it again or refer back to it later once you've had some more experience with code. However, all of this information will prove invaluable to you as you continue through this book and as you start writing your own code. Once you understand the GameObject/Component structure of Unity and how to take advantage of the Unity Inspector to set and modify variables, you'll find that your Unity coding moves a lot faster and more smoothly.

Chapter 21. Boolean Operations and Conditionals

Most people have heard that computer data is, at its base level, composed entirely of 1s and 0s—bits that are either true or false. However, only programmers really understand how much of programming is about boiling a problem down to a true or false value and then responding to it.

In this chapter, you learn about Boolean operations like AND, OR, and NOT; you learn about comparison statements like `>`, `<`, `==`, and `!=`; and you come to understand conditionals like `if` and `switch`. These all lie at the heart of programming.

Booleans

As you learned in the previous chapter, a bool is a variable that can hold a value of either true or false. Booleans were named after George Boole, a mathematician who worked with true and false values and logical operations (now known as "Boolean operations"). Though computers did not exist at the time of his research, computer logic was fundamentally based on it.

In C# programming, bools are used to store simple information about the state of the game (for example, `bool gameOver = false;`) and to control the flow of the program through the `if` and `switch` statements covered later in this chapter.

Boolean Operations

Boolean operations allow programmers to modify or combine bool variables in meaningful ways.

`!` (The NOT Operator)

The ! (either pronounced “not” or “bang”) operator reverses the value of a bool. False becomes true, and true becomes false:

```
print( !true );     // Outputs: false
print( !false );    // Outputs: true
print( !(!true) );  // Outputs: true    (the double negative of true is true)
```

! is also sometimes referred to as the *logical negation operator* to differentiate it from ~ (the bitwise not operator), which is explained in the “Bitwise Boolean Operators and Layer Masks” section of Appendix B, “Useful Concepts.”

&& (The AND Operator)

The && operator returns true only if both operands are true:

```
print( false && false );    // false
print( false && true  );    // false
print( true  && false );    // false
print( true  && true  );    // true
```

|| (The OR Operator)

The || operator returns true if either operand is true as well as if both are true:

```
print( false || false );    // false
print( false || true  );    // true
print( true  || false );    // true
print( true  || true  );    // true
```

Shorting Versus Non-Shorting Boolean Operators

The standard forms of AND and OR (&& and ||) are *shorting* operators, which means that once the operator has determined the return value, it will return that value without executing the rest of the

code. The following code listing includes several examples of how this works.

Tip

In the following code listing, a double slash followed by a lowercase letter (e.g., `// a`) to the right of a line indicates that there is an explanation of that line following the code listing. Throughout the book, the explanations will usually be at the end of the code listing, though in this example, they are included in the middle of the code to aid clarity.

```
 1 // This function prints "--true" and returns a true value.
 2 bool printAndReturnTrue() {
 3     print( "--true" );
 4     return( true );
 5 }
 6
 7 // This function prints "--false" and returns a false value.
 8 bool printAndReturnFalse() {
 9     print( "--false" );
10     return( false );
11 }
12
13 void ShortingOperatorTest() {
14     // The first half of this test uses the shorting && and ||
15     bool tfAnd = ( printAndReturnTrue() && printAndReturnFalse()
);      // a
16     print( "tfAnd: "+tfAnd );
```

a. This line will print `--true` and `--false` before setting `tfAnd` to false. Because the first argument that the shorting `&&` operator evaluates is true, it must also evaluate the second argument to determine that the result is false.

```
17
18    bool tfAnd2 = ( printAndReturnFalse() && printAndReturnTrue()
);    // b
19    print( "tfAnd2: "+tfAnd2 );
```

b. This line only prints `--false` before setting `tfAnd2` to false. Because the first argument that the shorting `&&` operator evaluates is false, it returns false without evaluating the second argument at all. On this line, `printAndReturnTrue()` is *not* executed.

```
20
21    bool tfOr = ( printAndReturnTrue() || printAndReturnFalse()
);     // c
22    print( "tfOr: "+tfOr );
```

c. This line only prints `--true` before setting `tfOr` to true. Because the first argument that the shorting `||` operator evaluates is true, it returns true without evaluating the second.

```
23
24    bool tfOr2 = ( printAndReturnFalse() || printAndReturnTrue()
);    // d
25    print( "tfOr2: "+tfOr2 );
26
```

d. This line prints `--false` and `--true` before setting `tfOr2` to true. Because the first argument that the shorting `||` operator evaluates is false, it must evaluate the second argument to determine which value to return.

```
27
28    // The second half of this test uses the nonshorting & and |
29    bool tfAnd3 = ( printAndReturnFalse() & printAndReturnTrue()
);    // e
30    print( "tfAnd3: "+tfAnd3 );
```

e. The non-shorting `&` operator will evaluate both arguments regardless of the value of the first argument. As a result, this line

prints `--false` and `--true` before setting `tfAnd3` to false.

```
31
32     bool tfOr3 = (printAndReturnTrue() | printAndReturnFalse()
);       // f
33     print( "tfOr3: "+tfOr3 );
34
35 }
```

f. The non-shorting `|` operator will evaluate both arguments regardless of the value of the first argument. This line prints `--true` and `--false` before setting `tfOr3` to true.

It is useful to know about both shorting and non-shorting operators when writing your code. Shorting operators (`&&` and `||`) are much more commonly used, but `&` and | can be used when you want to ensure that you evaluate all of the arguments of a Boolean operator.

If you want, I recommend entering this code into Unity and running the debugger to step through the behavior and really understand what is happening. To learn about the debugger, read Chapter 25, “Debugging.”

Bitwise Boolean Operators

`|` and `&` are also sometimes referred to as bitwise OR and bitwise AND because they can also be used to perform bitwise operations on integers. These are useful for a few esoteric things having to do with collision detection in Unity that you can learn more about in the “Bitwise Boolean Operators and Layer Masks” section of Appendix B, “Useful Concepts.”

Combination of Boolean Operations

It is often useful to combine various Boolean operations in a single line:

bool tf = true || false && false;

However, care must be taken when doing so because order of operations extends to Boolean operations as well. In C#, the order of operations for Boolean operations is as follows:

`!` - NOT
`&` - Non-shorting AND / Bitwise AND
`|` - Non-shorting OR / Bitwise OR
`&&` - AND
`||` - OR

This means that the previous line would be interpreted by the compiler as:

bool tf = true || (false && false);

The `&&` comparision is executed before the `||` comparison every time. Had you ignored order of operations and interpreted this line left-to-right, you might have expected the result to be false (e.g., `(true || false) && false` is false), but without any parentheses, the line actually evaluates to true!

Tip

Regardless of the order of operations, you should use parentheses for clarity in your code as often as possible. Good readability is essential in your code if you ever plan to work with someone else (or even if you want to read the same code yourself weeks or months later). I code by a simple rule: If there's any chance *at all* that something might be misunderstood later, I use parentheses and add comments to clarify what I am doing in the code and how it will be interpreted by the computer.

Logical Equivalence of Boolean Operations

The depths of Boolean logic are beyond the scope of this book, but suffice to say, some very interesting things can be accomplished by combining Boolean operations. In the examples of logic rules that follow `a` and `b` are bool variables. These rules hold true regardless of whether `a` and `b` are true or false and regardless of whether the shorting or non-shorting operators are used:

• Associativity: `( a & b ) & c` is the same as `a & ( b & c )`

• Commutativity: `( a & b )` is the same as `( b & a )`

• Distributivity of AND over OR: `a & ( b | c )` is the same as `( a & b ) | ( a & c )`

• Distributivity of OR over AND: `a | ( b & c )` is the same as `( a | b ) & ( a | c )`

• `( a & b )` is the same as `!( !a | !b )`

• `( a | b )` is the same as `!( !a & !b )`

If you're interested in more of these equivalencies and how they could be used, you can find many resources about Boolean logic online.

Comparison Operators

In addition to comparing Boolean values to each other, it is also possible to create a Boolean value by using comparison operators on any other values.

== (Is Equal To)

The equality comparison operator checks to see whether the values of any two variables or literals are equivalent to each other. The result of this operator is a Boolean value of either true or false:

```
 1 int i0 = 10;
 2 int i1 = 10;
 3 int i2 = 20;
 4 float f0 = 1.23f;
 5 float f1 = 3.14f;
 6 float f2 = Mathf.PI;
 7
 8 print( i0 == i1 );    // Outputs: True
 9 print( i1 == i2 );    // Outputs: False
10 print( i2 == 20 );    // Outputs: True
11 print( f0 == f1 );    // Outputs: False
12 print( f0 == 1.23f ); // Outputs: True
13 print( f1 == f2 );    // Outputs: False                          // a
```

a. The comparison in line 13 is false because Math.PI is far more accurate than 3.14f, and == requires that the values be exactly equivalent.

Warning

DON'T CONFUSE = AND == New coders are often confused by the difference between the assignment operator (`=`) and the equality operator (`==`). The assignment operator (`=`) is used to set the value of a variable while the equality operator (`==`) is used to compare two values. Consider the following code listing:

```
 1 bool f = false;
 2 bool t = true;
 3 print( f == t ); // prints: False
 4 print( f = t );  // prints: True
```

On line 3, `f` is compared to `t`, and because they are not equal, false is returned and printed. However, on line 4, `f` is assigned the value of `t`, causing the value of `f` to now be true, and true is printed.

Confusion is also sometimes an issue when talking about the two operators. To avoid confusion, I usually pronounce `i=5;` as "i equals

5," and I pronounce `i==5;` as "i is equal to 5."

See the "Testing Equality by Value or Reference" sidebar for more detailed information about how equality is handled for several different variable types.

Testing Equality by Value or Reference

Unity's version of C# will compare most simple data types *by value*. This means that as long as the values of the two variables are the same, they will be seen as equivalent. This works for all of the following data types:

• bool string

• int Vector3

• float Color

• char Quaternion

However, with more complex variable types like GameObject, Material, Renderer, and so on, C# instead checks equality *by reference*. When comparing equality by reference, it does not compare whether the values of the two variables are equal but instead checks to see if the *references* of the two variables are equal. In other words, it checks to see whether the two variables are referencing (or pointing to) the same single object in the computer's memory. In the following example of comparison by reference, we'll assume that `boxPrefab` is a pre-existing variable that references a GameObject prefab.

```
1 GameObject go0 = Instantiate<GameObject>( boxPrefab );
2 GameObject go1 = Instantiate<GameObject>( boxPrefab );
3 GameObject go2 = go0;
```

```
print( go0 == go1 ); // Output: false
print( go0 == go2 ); // Output: true
```

Even though the two instantiated `boxPrefab`s assigned to the variables `go0` and `go1` have the same values (they have the exact same default position, rotation, and so on) the `==` operator sees them as different because they are actually two different objects, and therefore reside in two different places in memory. `go0` and `go2` are seen as equal by `==` because they both refer to the exact same object. Let's continue the previous code:

```
go0.transform.position = new Vector3( 10, 20, 30);
print( go0.transform.position); // Output: (10.0, 20.0, 30.0)
print( go1.transform.position); // Output: ( 0.0,  0.0,  0.0)
print( go2.transform.position); // Output: (10.0, 20.0, 30.0)
```

Here, the position of `go0` is changed. Because `go1` is a different GameObject instance, its position remains the same. However, since `go2` and `go0` reference the same GameObject instance, `go2.transform.position` reflects the change as well.

Next, we'll change the position of `go1` to match that of `go0` (which is the same GameObject as that referenced by `go2`).

```
go1.transform.position = new Vector3( 10, 20, 30);
print( go0.transform == go1.transform);                    // Output: false
print( go0.transform.position == go1.transform.position); // Output: true
```

The transforms of the `go0` and `go1` are not equal, but their positions are equivalent because the Vector3 positions are being compared by value.

`!=` (Not Equal To)

The inequality operator returns true if two values are not equal and false if they are equal. It is the opposite of ==. When comparing objects by reference, != returns true when the two objects point to different locations in memory. (For the remaining comparisons, literal values will be used in the place of variables for the sake of clarity and space.)

```
print( 10 != 10 );          // Outputs: False
print( 10 != 20 );          // Outputs: True
print( 1.23f != 3.14f );    // Outputs: True
print( 1.23f != 1.23f );    // Outputs: False
print( 3.14f != Mathf.PI ); // Outputs: True
```

> (Greater Than) and < (Less Than)

> returns true if the value on the left side of the operator is greater than the value on the right:

```
print( 10 > 10 );          // Outputs: False
print( 20 > 10 );          // Outputs: True
print( 1.23f > 3.14f );    // Outputs: False
print( 1.23f > 1.23f );    // Outputs: False
print( 3.14f > 1.23f );    // Outputs: True
```

< returns true if the value on the left side of the operator is less than the value on the right:

```
print( 10 < 10 );          // Outputs: False
print( 20 < 10 );          // Outputs: True
print( 1.23f < 3.14f );    // Outputs: True
print( 1.23f < 1.23f );    // Outputs: False
print( 3.14f < 1.23f );    // Outputs: False
```

The characters < and > are also sometimes referred to as *angle brackets,* especially when they are used as tags in languages like HTML and XML or in generic functions in C#. However, when they are used as comparison operators, they are always called *greater*

than and *less than*. It is not possible to compare objects by reference using `>`, `<`, `>=`, or `<=`.

>= (Greater Than or Equal To) and
<= (Less Than or Equal To)

`>=` returns true if the value on the left side is greater than or equivalent to the value on the right:

```
print( 10 >= 10 );         // Outputs: True
print( 10 >= 20 );         // Outputs: False
print( 1.23f >= 3.14f );    // Outputs: False
print( 1.23f >= 1.23f );    // Outputs: True
print( 3.14f >= 1.23f );    // Outputs: True
```

`<=` returns true if the value on the left side is less than or equal to the value on the right:

```
print( 10 <= 10 );         // Outputs: True
print( 10 <= 20 );         // Outputs: True
print( 1.23f <= 3.14f );    // Outputs: True
print( 1.23f <= 1.23f );    // Outputs: True
print( 3.14f <= 1.23f );    // Outputs: False
```

Conditional Statements

Conditional statements can be combined with Boolean values and comparison operators to control the *flow* of your programs. This means that a true value can cause the code to generate one result while a false value can cause it to generate another. The two most common forms of conditional statements are `if` and `switch`.

`if` Statements

An `if` statement will only execute the code inside its braces `{}` if the value inside its parentheses `()` evaluates to true:

```
if (true) {
    print( "The code in the first if statement executed." );
}
if (false) {
    print( "The code in the second if statement executed." );
}

// The output of this code will be:
//     The code in the first if statement executed.
```

The code inside the braces `{}` of the first `if` statement executes, yet the code inside the braces of the second `if` statement does not.

Note

Statements enclosed in braces do not require a semicolon after the closing brace. Other statements that have been covered all require a semicolon at the end:

```
float approxPi = 3.14159f; // There's the standard semicolon
```

Compound statements (that is, those surrounded by braces) do not require a semicolon after the closing brace:

```
if (true) {
    print( "Hello" );     // This line needs a semicolon.
    print( "World" );     // This line needs a semicolon.
}                         // No semicolon required after the closing brace!
```

The same is true for *any* compound statement surrounded by braces.

Combining `if` Statements with Comparison and Boolean Operations

Boolean operators can be combined with `if` statements to react to various situations in your game:

```
bool night = true;
bool fullMoon = false;

if (night) {
    print( "It's night." );
}
if (!fullMoon) {
    print( "The moon is not full." );
}
if (night && fullMoon) {
    print( "Beware werewolves!!!" );
}
if (night && !fullMoon) {
    print( "No werewolves tonight. (Whew!)" );
}

// The output of this code will be:
//     It's night.
//     The moon is not full.
//     No werewolves tonight. (Whew!)
```

And, of course, `if` statements can also be combined with comparison operators:

```
if (10 == 10 ) {
    print( "10 is equal to 10." );
}
if ( 10 > 20 ) {
    print( "10 is greater than 20." );
}
if ( 1.23f <= 3.14f ) {
    print( "1.23 is less than or equal to 3.14." );
}
if ( 1.23f >= 1.23f ) {
    print( "1.23 is greater than or equal to 1.23." );
}
if ( 3.14f != Mathf.PI ) {
```

```
    print( "3.14 is not equal to "+Mathf.PI+"." );
    // + can be used to concatenate strings with other data types.
    // When this happens, the other data type is converted to a string.
}

// The output of this code will be:
//     10 is equal to 10.
//     1.23 is less than or equal to 3.14.
//     1.23 is greater than or equal to 1.23.
//     3.14 is not equal to 3.141593.
```

Warning

AVOID USING `=` IN AN `if` STATEMENT As was mentioned in the previous warning, `==` is a comparison operator that determines whether two values are equivalent. `=` is an assignment operator that assigns a value to a variable. If you accidentally use `=` in an `if` statement, the result can actually be an assignment instead of a comparison.

Sometimes Unity will catch this by giving you an error about not being able to implicitly convert a value to a Boolean. You will get that error from this code:

```
float f0 = 10f;
if ( f0 = 10 ) {
    print( "f0 is equal to 10.");
}
```

Other times, Unity will actually give you a warning stating that it found an `=` in an `if` statement and asking if you really meant to type `==`. Sometimes, however, Unity might not give you any warning, so you need to be careful and watch out for this yourself.

---

if...else

Many times, you will want to do one thing if a value is true and another if it is false. At these times, an `else` clause is added to the `if` statement:

```
bool night = false;

if (night) {
    print( "It's night." );
} else {
    print( "It's daytime. What are you worried about?" );
}

// The output of this code will be:
//     It's daytime. What are you worried about?
```

In this case, because `night` is false, the code in the `else` clause is executed.

if...else if...else

It's also possible to have a chain of `else` clauses:

```
bool night = true;
bool fullMoon = true;

if (!night) {               // Condition 1 (false)
    print( "It's daytime. What are you worried about?" );
} else if (fullMoon) {      // Condition 2 (true)
    print( "Beware werewolves!!!" );
} else {                    // Condition 3 (not checked)
    print( "It's night, but the moon is not full." );
}

// The output of this code will be:
//     Beware werewolves!!!
```

Once any condition in the `if...else if...else` chain evaluates to true, all subsequent conditions are no longer evaluated (the rest of the chain is shorted). In the previous listing, Condition 1 is false, so Condition 2 is checked. Because Condition 2 is true, the computer will completely skip Condition 3.

Nesting `if` Statements

It is also possible to nest `if` statements inside of each other for more complex behavior:

```
bool night = true;
bool fullMoon = false;

if (!night) {
    print( "It's daytime. What are you worried about?" );
} else {
    if (fullMoon) {
        print( "Beware werewolves!!!" );
    } else {
        print( "It's night, but the moon is not full." );
    }
}

// The output of this code will be:
//     It's night, but the moon is not full.
```

`switch` Statements

A `switch` statement can take the place of several `if...else` statements, but it has some strict limitations:

1. Switch statements can only compare for equality.

2. Switch statements can only compare a single variable.

3. Switch statements can only compare that variable against literals (not other variables).

Here is an example:

```
int num = 3;

switch (num) {  // The variable in parentheses (num) is the one being compared
case (0):  // Each case is a literal number that is compared against num
    print( "The number is zero." );
    break;  // Each case must end with a break statement.
case (1):
    print( "The number is one." );
    break;
case (2):
    print( "The number is two." );
    break;
default:  // If none of the other cases are true, default will happen
    print( "The number is more than a couple." );
    break;
}  // The switch statement ends with a closing brace.

// The output of this code is:
//     The number is more than a couple.
```

If one of the cases holds a literal with the same value as the variable being checked, the code in that case is executed until the `break` is reached. Once the computer hits the `break`, it exits the switch and does not evaluate any other cases.

It is also possible to have one case “fall through” to the next, but only if there are no lines of code between the two:

```
int num = 4;

switch (num) {
```

```
case (0):
    print( "The number is zero." );
    break;
case (1):
    print( "The number is one." );
    break;
case (2):
    print( "The number is a couple." );
    break;
case (3):
case (4):
case (5):
    print( "The number is a few." );
    break;
default:
    print( "The number is more than a few." );
    break;
}

// The output of this code is:
//     The number is a few.
```

In the previous code, if `num` is equal to 3, 4, or 5, the output will be `The number is a few.`

Knowing what you know about combining conditionals and `if` statements, you might question when `switch` statements are used, since they have so many limitations. They are used quite often to deal with the different possible states of a GameObject. For instance, if you made a game where the player could transform into a person, bird, fish, or wolverine, there might be a chunk of code that looked like this:

```
string species = "fish";
bool   onLand = false;

// Each different species type will move differently
```

```
public function Move() {
    switch (species) {
    case ("person"):
        Run(); // Calls a function named Run()
        break;
    case ("bird"):
        Fly();
        break;
    case ("fish"):
        if (!onLand) {
            Swim();
        } else {
            FlopAroundPainfully();
        }
        break;
    case ("wolverine"):
        Scurry();
        break;
    default:
        print( "Unknown species type: "+species );
        break;
    }
}
```

In the preceding code, the player (as a fish in water) would `Swim()`. It's important to note that the `default` case here is used to catch any species that the `switch` statement isn't ready for and that it will output information about any unexpected species it comes across. For instance, if `species` were somehow set to `"lion"`, the output would be:

```
Unknown species type: lion
```

In the preceding code syntax, you also see the names of several functions that are not yet defined (e.g., `Run()`, `Fly()`, `Swim()`). The next chapter covers the creation of your own functions.[1]

[1] For some pretty esoteric reasons, it's actually not great code style to have a different named function for each kind of movement like this. However, that's pretty far beyond the scope of this book. Once you've read this book, check out Robert Nuystrom's website and book, *Game Programming Patterns* (http://gameprogrammingpatterns.com) for some great information on good programming strategies.

Summary

Though Boolean operations may seem a bit dry, they form a big part of the core of programming. Computer programs are full of hundreds, even thousands, of branch points where the computer can do either one thing or another, and these all boil down in some way to Booleans and comparisons. As you continue through the book, you may want to return to this section from time to time if you're ever confused by any comparisons in the code you see.

Chapter 22. Loops

Computer programs are usually designed to do the same thing repeatedly. In a standard game loop, the game draws a frame to the screen, takes input from the player, considers that input, and then draws the next frame, repeating this behavior at least 30 times every second.

A loop in C# code causes the computer to repeat a certain behavior several times. This could be anything from looping over every enemy in the scene and considering the AI of each to looping over all the physics objects in a scene and checking for collisions. By the end of this chapter, you'll understand all you need to know about loops, and in the next chapter, you'll learn how to use them with arrays.

Types of Loops

C# has only four kinds of loops: `while`, `do...while`, `for`, and `foreach`. Of those, you'll be using `for` and `foreach` much more often than the others because they are generally safer and more adaptable to the challenges you'll encounter while making games:

• `while` loop: The most basic type of loop. Checks a condition before each loop to determine whether to continue looping.

• `do...while` loop: Similar to the while loop, but checks a condition *after* each loop to determine whether to continue looping.

• `for` loop: A loop statement that includes an initial statement, a variable that increments with each iteration, and an end condition. The most commonly used loop structure.

• `foreach` loop: A loop statement that automatically iterates over every element of an enumerable object or collection. This chapter

contains some discussion of `foreach`, and it is covered more extensively in the next chapter as part of the discussion of the C# collections List and array.

Set Up a Project

In Appendix A, "Standard Project Setup Procedure," detailed instructions show you how to set up Unity projects for the chapters in this book. At the start of each project chapter, you will also see a sidebar like the one here. Please follow the directions in the sidebar to create the project for this chapter.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. For information on the standard project setup procedure, see Appendix A.

- **Project name:** Loop Examples
- **Scene name:** _Scene_Loops
- **C# Script names:** Loops

Attach the script Loops to the Main Camera in the scene.

`while` Loops

The `while` loop is the most basic loop structure. However, this also means that it lacks the safety of using a more modern form of loop. In my coding, I almost never use `while` loops because of the danger that using one could create an infinite loop.

The Danger of Infinite Loops

An infinite loop occurs when a program enters a loop and is unable to escape it. Let's write one to see what happens. Open the Loops C# script in MonoDevelop (by double-clicking it in the Project pane) and add the following bolded code.

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Loops : MonoBehaviour {
 5
 6     void Start () {
 7         while (true) {
 8             print( "Loop" );
 9         }
10     }
11
12 }
```

Save this script by choosing *File* > *Save* from the MonoDevelop menu bar. After you've done this, switch back to Unity and press the triangular Play button at the top of the Unity window. See how nothing happens... see how nothing happens *forever?* In fact, you're probably going to have to *force quit* Unity now (see the sidebar for instructions). What you have just encountered is an *infinite loop*, and as you can see, an infinite loop will completely freeze Unity. It is lucky that we all run multithreaded computer operating systems now, because in the old days of single-threaded systems, infinite loops wouldn't just freeze a single application, they'd freeze the entire computer and require a restart.

How to Force Quit an Application

On OS X

Implement a force quit by doing the following:

1. Press Command-Option-Esc on the keyboard. This brings up the Force Quit window.

2. Find the application that is misbehaving. Its name will often be followed by “(not responding)” in the applications list.

3. Click that application name in the list, and then click *Force Quit*. You may need to wait a few seconds for the force quit to happen.

On Windows

Implement a force quit by doing the following:

1. Press Shift+Control+Esc on the keyboard. This brings up the Windows Task Manager.

2. Find the application that is misbehaving.

3. Click that application and then click End Task. You may need to wait a few seconds for the force quit to happen.

If you have to force quit Unity while it is running, you will lose any work that you’ve done since your last save. Because you must constantly save C# scripts, they shouldn’t be an issue, but you might have to redo unsaved changes made to your scene. For example, in _Scene_Loops, if you did not save the scene after adding the Loops C# script to the Main Camera, you will need to attach it to the Main Camera again.

So, what happened there that caused the infinite loop? To discover that, take a look at the `while` loop.

```
7       while (true) {
8           print( "Loop" );
9       }
```

Everything within the braces of a while loop will be executed repeatedly as long as the *condition clause* within the parentheses is true. On line 7, the condition is always true, so the line `print( "Loop" );` will repeat infinitely.

But, you may wonder, if this line was repeating infinitely, why did you never see “Loop” printed in the Console pane? Though the `print()` function was called many times (probably hundreds of thousands or even millions of times before you decided to force quit Unity), you were never able to see the output in the Console pane because Unity was trapped in the infinite `while` loop and was unable to redraw the Unity window (which would have needed to happen to see the changes to the Console pane).

A More Useful `while` Loop

Open the Loops C# script in MonoDevelop and modify it to read as follows:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Loops : MonoBehaviour {
 5
 6     void Start () {
 7         int i=0;
 8         while ( i<3 ) {
 9             print( "Loop: "+i );
10             i++;      // See the sidebar on Increment and Decrement Operators
11         }
12     }
13
14 }
```

Increment and Decrement Operators

On line 10 of the code listing for the “more useful” `while` loop is the first instance in this book of the *increment operator* (`++`). The increment operator increases the value of the variable that it follows by 1. So, if `i=5`, then the `i++;` statement would set the value of `i` to `6`.

There is also a *decrement operator* (`--`). The decrement operator decreases the value of the variable by 1.

The increment and decrement operators can be placed either before or after the variable name, and doing so causes the statement to be treated differently (i.e., `++i` and `i++` act slightly differently). The difference is in whether the initial value is returned (`i++`) or whether the incremented value is returned (`++i`). Here’s an example to clarify.

```
 6   void Start() {
 7       int i = 1;
 8       print( i );    // Output: 1
 9       print( i++ );  // Output: 1
10        print( i );    // Output: 2
11        print( ++i );  // Output: 3
12   }
```

As you can see, line 8 prints the current value of `i`, which is 1. Then, on line 9, the post-increment operator `i++` first returns the current value of `i`, which is printed (resulting in the 1), and then increments the `i`, setting its value to 2.

Line 10 prints the current value of `i`, which is 2. Then, on line 10, the pre-increment operator `++i` first increments the value of `i` from 2 to 3 and then returns it to the print function, which prints a 3.

Tip

In most of the examples in this chapter, the *iteration variable* used will be named `i`. The variable names `i`, `j`, and `k` are often used by programmers as iteration variables (i.e., the variable that increments

in a loop), and as a result, they are rarely used in any other code situations. Because these variables are created and destroyed so often in various loop structures, you should generally avoid using the variable names `i`, `j`, or `k` for anything else.

Save your code, switch back to Unity and click Play. This time, Unity does not get stuck in an infinite loop because the `while` condition clause (`i<3`) eventually becomes false. The output from this program to the console (minus all the extra stuff Unity throws in) is as follows:

Loop: 0
Loop: 1
Loop: 2

That is because it calls `print( i )` every time the while loop iterates. It's important to note that the condition clause is checked *before* each iteration of the loop.

`do...while` Loops

A `do...while` loop works in the same manner as a `while` loop, except that the condition clause is checked *after* each iteration. This guarantees that the loop will run at least once. Modify the code to read as follows:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Loops : MonoBehaviour {
 5
 6     void Start () {
 7         int i=10;
 8         do {
 9             print( "Loop: "+i );
10              i++;
```

```
11        } while ( i<3 );
12    }
13
14 }
```

Make sure that you change line 7 of the `Start()` function to `int i=10;`. Even though the `while` condition is not ever true (10 is never less than 3), the loop still goes through a single iteration before testing the condition clause on line 11. Had `i` been initialized to `0` here as it was in the `while` loop example, the console output would have looked the same, so we set `i=10` in line 7 to demonstrate that a `do...while` loop will always run at least once regardless of the value of `i`. It is important to place a trailing semicolon (`;`) after the condition clause in a `do...while` loop.

Save this script and try it out in Unity to see the result.

`for` Loops

In both the `while` and `do...while` examples, we needed to declare and define a variable `i`, increment the variable `i`, and then check the condition clause on the variable `i`; and each of these actions was performed by a separate statement. The `for` loop handles all of these actions in a single line. Write the following code in the Loops C# script, and then save and run it in Unity.

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Loops : MonoBehaviour {
 5
 6     void Start() {
 7         for ( int i=0; i<3; i++ ) {
 8             print( "Loop: "+i );
 9         }
10     }
```

```
11
12 }
```

The `for` loop in this example sends the same output to the Console pane as was sent by the preceding "more useful" `while` loop, yet it does so in fewer lines of code. The structure of a `for` loop requires an *initialization* clause, a *condition* clause, and an *iteration* clause to be valid. In the above example, the three clauses are as follows:

Initialization clause: `for ( int i=0; i<3; i++ ) {`

Condition clause: `for ( int i=0; i<3; i++ ) {`

Iteration clause: `for ( int i=0; i<3; i++ ) {`

The initialization clause (`int i=0;`) is executed before the `for` loop begins. It declares and defines a variable that is scoped locally to the `for` loop. This means that, the `int i` above will cease to exist once the for loop is complete. For more information on variable scoping, see the "Variable Scope" section of Appendix B, "Useful Concepts."

The condition clause (`i<3`) is checked before the first iteration of the `for` loop (just as the condition clause is checked before the first iteration of a `while` loop). If the condition clause is true, the code between the braces of the `for` loop is executed.

Once an iteration of the code between the braces of the for loop has completed, the iteration clause (`i++`) is executed (that is, after `print( i );` has executed once, `i++` is executed). Then the condition clause is checked again, and if the condition clause is still true, the code in the braces is executed again, and the iteration clause is executed again. This continues until the condition clause evaluates to false, and then the `for` loop ends.

Because `for` loops mandate that each of these three clauses be included and that they all be on the same line, it is easier to avoid writing infinite loops when working with `for` loops.

Warning

DON'T FORGET THE SEMICOLONS BETWEEN EACH CLAUSE OF THE `for` STATEMENT It is critical that the initialization, condition, and iteration clauses be separated by semicolons. This is because each is an independent clause that must be terminated by a semicolon like any independent clause in C#. Just as most lines in C# must be terminated by a semicolon, so must the independent clauses in a `for` loop.

The Iteration Clause Doesn't Have to Be ++

Though the iteration clause is commonly an increment statement like `i++`, it doesn't have to be. Any operation can be used in the iteration clause.

Decrement

One of the most common alternate iteration clauses is counting down rather than counting up. This is accomplished by using a decrement operator in a `for` loop.

```
 6    void Start() {
 7        for ( int i=5; i>2; i-- ) {
 8            print( "Loop: "+i );
 9        }
10    }
```

And the output to the Console pane would be as follows:

```
Loop: 5
Loop: 4
Loop: 3
```

`foreach` Loops

A `foreach` loop is kind of like an automatic `for` loop to use on anything that is enumerable. In C#, most collections of data are enumerable, including strings (which are a collection of chars) and the Lists and arrays covered in the next chapter. Try this example in Unity.

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Loops : MonoBehaviour {
 5
 6     void Start() {
 7         string str = "Hello";
 8         foreach( char chr in str ) {
 9             print( chr );
10         }
11     }
12
13 }
```

The Console output will print a single char from the string `str` on each iteration, resulting in:

```
H
e
l
l
o
```

The `foreach` loop guarantees that it will iterate over all the elements of the enumerable object. In this case, it iterates over each character in the string "`Hello`". `foreach` loops are covered further in the next chapter as part of the discussion of lists and arrays.[1]

[1] Though this shouldn't be a concern for you now, you should be aware that foreach loops are less performant than other kinds, meaning that they run slightly more slowly and that they generate

more allocated memory garbage that will need to be automatically collected and managed by the computer. If working on a speed- or memory-critical game on a weaker computer like a mobile phone, you may want to avoid foreach loops.

Jump Statements within Loops

A jump statement is any statement that causes code execution to jump to another location in the code. One example that has already been covered is the `break` statement at the end of each case in a `switch` statement.

`break`

`break` statements can also be used to prematurely break out of any kind of loop structure. To see an example, change your `Start()` function to read as follows:

```
 6    void Start() {
 7        for ( int i=0; i<10; i++ ) {
 8            print( i );
 9            if ( i==3 ) {
10                break;
11            }
12        }
13    }
```

Note that in the this code listing, lines 1–5 and the final line containing only a closing brace `}` (which was line 13 in prior listings) have been omitted because they are identical to the lines in previous code listings. Those lines should still be there, and you should just replace the `foreach` loop from lines 7–10 of the preceding foreach code listing with lines 7–12 if this code listing.

Run this in Unity, and you will get this output:

0
1
2
3

The `break` statement exits the `for` loop prematurely. A `break` can also be used in `while`, `do...while`, and `foreach` statements.



The lettered paragraphs below refer to lines in the preceding code that are marked with `// a` and `// b` to the right of the line (these have been bolded in the preceding code for emphasis).

a. This line shows the single-line version of an `if` statement. If there is only one line, the braces are not necessary.

b. This code only outputs `3` and `2` because on the second iteration of the loop, the `i--` decrement reduces `i` to `1`, and then the condition clause for the `if` statement on line 11 is true and breaks out of the loop.

Take the time to look at each code example above and make sure you understand why each is generating the output shown. If any look confusing, type the code into Unity and then run through it with the debugger. (The debugger is explained in detail in Chapter 25, “Debugging.”)

`continue`

`continue` is used to force the program to skip the remainder of the current iteration and continue to the next.



In the preceding code, any time that `i%90 != 0` (that is, `i/90` has a remainder other than 0), the `continue` will cause the `for` loop to move

on to the next iteration, skipping the `print( i );` line. The `continue` statement can also be used in `while`, `do...while`, and `foreach` loops.

Modulus Operator

Line 8 of the code listing for the `continue` jump statement is the first instance in this book of the *modulus* operator (`%`). Modulus (or mod) returns the remainder of dividing one number by another. For example, `12%10` would return a value of 2 because the remainder of 12/10 is 2.

Mod can also be used with floats, so `12.4f%1f` would return 0.4f, the remainder when 12.4 is divided by 1.

Summary

Understanding loops is one of the key elements of becoming a good programmer. However, it's fine if not all of this makes perfect sense now. Once you start using loops in the development of some actual game prototypes, it will become more clear to you. Just make sure that you are actually typing each code example into Unity and running it to help with your understanding of the material.

Also remember that in my coding, I most commonly use `for` and `foreach` and rarely or never use `while` and `do...while` because of the danger of infinite loops.

In the next chapter, you will learn about arrays and lists, two forms of ordered collections of similar items, and you will see how loops are used to iterate over these collections.

Chapter 23. Collections in C#

C# collections enable you to act on several similar things as a group. For example, you could store all of the enemy GameObjects in a List and loop over that List each frame to update all of their positions and states.

This chapter covers three important types of these collections in detail: Lists, arrays, and Dictionaries. By the end of this chapter, you will understand how these collection types work and which to use in various situations.

C# Collections

A collection is a group of objects that are referenced by a single variable. In regular life, collections would be things like a group of people, a pride of lions, a parliament of rooks, or a murder of crows. Just like these animal grouping terms, the collections we'll use in C# can only hold a single type of data (e.g., you couldn't include a tiger in a pride of lions), though some rarely used collections do allow multiple data types. The array type is built in to C# at a low level, while the other collection types here rely on the System.Collections.Generic code library to work, as described later in the chapter.

Commonly-Used Collections

The following is a brief overview of some of the most commonly used collections. If a collection is described in much more depth later in this chapter, a plus (+) will appear after the name of the collection type.

• **Array +:** An array is an indexed, ordered list of objects. The length of an array must be set when it is defined, and it cannot be altered,

which differentiates it from the more flexible List type. The word “array” will only be capitalized when referring to the C# class Array, which is different from primitive arrays of data described in this chapter. As the most primitive type of collection, arrays have few special class functions. However, arrays do have numeric bracket access, meaning that objects can be added to and read from the array using the array name and [] like:

```
stringArray[0] = "Hello";
stringArray[1] = "World";
print( stringArray[0]+" "+stringArray[1] ); // Output: Hello World
```

• **Dictionary +:** Dictionaries allow you to associate key/value pairs, where an object is stored based on a particular key. A real-world example of this is a library, where the *key* of the Dewey Decimal system allows readers to access the *value* of the individual books. Unlike all other collections in this chapter, Dictionaries are declared with two types (the key type and value type). Values can be added to and read from the Dictionary using bracket access (e.g., `dict["key"]`). Dictionaries include the following methods:

• `new Dictionary<Tkey, Tvalue>()` – Declare a new Dictionary with key and value types

• `Add(TKey, TValue)` – Adds the object TValue to the Dictionary with the key TKey

• `Clear()` – Remove all objects from the Dictionary

• `ContainsKey(TKey)` – Returns true if the key TKey is in the Dictionary

• `ContainsValue(TValue)` – Returns true if the value TValue is in the Dictionary

• `Count` – Property that returns the number of key/value pairs in the Dictionary

• `Remove(TKey)` – Removes the value at key TKey from the Dictionary.

• **List +:** Lists are similar to arrays, except they are flexible in length and very slightly slower for performance. In this book, List will be capitalized when referring to the C# type to help distinguish it from the common usage of "list." The List is the most commonly used collection in this book. Lists can use numeric bracket access like arrays. Lists also include the following methods:

• `new List<T >()` – Declare a new List of the type T

• `Add(T)` – Adds an object of the type T to the end of the List

• `Clear()` – Remove all objects from the Dictionary

• `Contains(T)` – Returns true if the object T is in the List

• `Count` – Property that returns the number of objects in the List

• `IndexOf(T)` – Returns the numeric index of where the object exists in the List. If the object T is not in the List, -1 is returned.

• `Remove(T)` – Removes the object T from the List

• `RemoveAt(#)` – Removes the object that is at the index # from the List

• **Queue:** As a first-in, first-out (FIFO) ordered collection, a Queue is similar to a line you might stand in at an amusement park. Objects are added to the end of the Queue with Enqueue() and removed from the beginning of the Queue with Dequeue(). Queues include the following methods:

• `Clear()` – Remove all objects from the Queue

• `Contains(X)` – Returns true if X is in the Queue somewhere

• `Count` – Property that returns the number of objects in the Queue

• `Dequeue()` – Removes and returns the object at the beginning of the Queue

• `Enqueue(X)` – Adds the object X to the end of the Queue

• `Peek()` – Returns the object at the beginning of the Queue without removing it

• **Stack:** As a first-in, last-out (FILO) ordered collection, a Stack is similar to a stack of cards. Objects are added to the top of the Stack with Push() and removed from the top of the Stack with Pop(). Stacks include the following methods:

• `Clear()` – Remove all objects from the Stack

• `Contains(X)` – Returns true if X is in the Stack somewhere

• `Count` – Property that returns the number of objects in the Stack

• `Peek()` – Returns the object at the beginning of the Queue without removing it

• `Pop()` – Removes and returns the object at the top of the Stack

• `Push(X)` – Adds the object X to the top of the Stack

Because Lists are the most used collection in this book, we'll start with them and then cover both Dictionaries and arrays.

Set Up a Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see Appendix A, "Standard Setup Procedure."

• **Project name:** Collections Project

• **Scene name:** _Scene_Collections

• **C# script names:** ArrayEx, DictionaryEx, ListEx

Attach all three C# scripts to *Main Camera* in _Scene_Collections.

Using Generic Collections

At the top of all Unity C# scripts, there are two lines that begin with the word `using`.

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

The first line at the top of each C# script gives the remainder of the script knowledge of standard Unity objects that are used throughout all Unity coding. The second line gives the script knowledge of the un-typed collection ArrayList that I don't recommend using (because of its lack of strong typing).

The List and Dictionary collection types are not actually part of the standard `using` lines that Unity automatically includes at the top of each script, so we need to add another line:

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

This line enables several *generic collections*. A generic collection is one that is strongly typed to hold a collection of a single specific data type that is specified using angle brackets. Example declarations of generic collections include:

• `public List<string> sList;`—This declares a List of strings

• `public List<GameObject> goList;`—This declares a List of GameObjects

• `public Dictionary<char,string> acronymDict;`—This declares a Dictionary of strings that have chars as keys (e.g., the char `'o'` could be used to access the string `"Okami"`).

`System.collections.generic` also defines several other generic data types that are beyond the scope of this chapter. These include the generic versions of Queue and Stack mentioned above. Unlike arrays, which are locked to a specified length, all generic collection types can adjust their length dynamically.

List

Double-click the ListEx C# script in the Project pane to open it in MonoDevelop and add the following bolded code. The `// a`-style comments on the far right side of the code listing are references to explanations listed after the code. Lines that you must type are bolded.

```
 1 using UnityEngine;                                    // a
 2 using System.Collections;                               // b
 3 using System.Collections.Generic;                                  // c
 4
 5 public class ListEx : MonoBehaviour {
 6     public List<string>       sList;                    // d
 7
 8     void Start () {
 9         sList = new List<string>();                     // e
10         sList.Add( "Experience" );                       // f
11         sList.Add( "is" );
12         sList.Add( "what" );
13         sList.Add( "you" );
14         sList.Add( "get" );
15         sList.Add( "when" );
16         sList.Add( "you" );
17         sList.Add( "don't" );
18         sList.Add( "get" );
19         sList.Add( "what" );
20         sList.Add( "you" );
21         sList.Add( "wanted." );
22         // This quote is from my professor, Dr. Randy Pausch
(1960-2008)
```

```
23
24        print( "sList Count = "+sList.Count );                    // g
25        print( "The 0th element is: "+sList[0] );                  // h
26        print( "The 1st element is: "+sList[1] );
27        print( "The 3rd element is: "+sList[3] );
28        print( "The 8th element is: "+sList[8] );
29
30        string str = "";
31        foreach (string sTemp in sList) {                          // i
32            str += sTemp+" ";
33        }
34        print( str );
35    }
36 }
```

a. The UnityEngine library enables all of the classes and types that are specific to Unity (e.g., GameObject, Renderer, Mesh). It is mandatory that it be included in any Unity C# script.

b. The System.Collections library that is at the beginning of all C# scripts enables the ArrayList type (among others). ArrayList is another type of C# collection that is similar to List except that ArrayLists are not limited to a single type of data. This enables more flexibility, but I have found it to have more detriments than benefits when compared to Lists (including a significant performance penalty).

c. The List collection type is part of the System.Collections.Generic C# library, so that library must be imported to enable the use of Lists. System.Collections.Generic enables a whole slew of generic collection types beyond just List. You can learn more about them online by searching “C# System.Collections.Generic” if you’re curious.

d. This declares the `List<string> sList`. All generic collection data types have their name followed by angle brackets `< >` surrounding a specified data type. In this case, the List is a List of strings. However, the strength of generics is that they can be used for any data type.

You could just as easily create a `List<int>`, `List<GameObject>`, `List<Transform>`, `List<Vector3>`, and so on. The type of the List must be assigned at the time of declaration.

e. The declaration of `sList` on line 6 makes `sList` a variable name that can hold a List of strings, but the value of `sList` is `null` (that is, it has no value) until `sList` is defined on line 9. Before this definition, any attempt to add elements to `sList` would have caused an error. The List definition must repeat the type of the List in the `new` statement. A newly defined List initially contains no elements and has a `Count` of zero.

f. A List's `Add()` function adds an element to the List. This will insert the string literal `"Experience"` into the 0th (pronounced "zeroth") element of the List. See the "Lists and Arrays are Zero-Indexed" sidebar for information about zero-indexed Lists.

g. A List's `Count` property returns an int representing the number of elements in the List. Output:

```
sList Count = 12
```

h. Lines 25–28 demonstrate the use of *bracket access* (e.g., `sList[0]`). Bracket access uses brackets `[]` and an integer to reference a specific element in a List or array. The integer between the brackets is known as the "index." Output:

```
The 0th element is: Experience
The 1st element is: is
The 3rd element is: you
The 8th element is: get
```

i. `foreach` (introduced in the previous chapter) is often used with Lists and other collections. Just as a string is a collection of chars, `List<string> sList` is a collection of strings. The string `sTemp` variable is scoped to the `foreach` statement, so it will cease to exist once the `foreach` loop has completed. Because Lists are strongly typed (that is, C# knows that `sList` is a `List<string>`) the elements

of `sList` can be assigned to string `sTemp` without requiring any kind of conversion. This is one of the major advantages of the List type over the nontyped ArrayList type. Output:

Experience is what you get when you don't get what you wanted.

Lists and Arrays are zero-indexed

List and array collection types are *zero indexed*, meaning that what you might think of as the “first” element is actually element [0]. Throughout the book, I will refer to this element as the 0th or “zeroth” element.

For these examples, we’ll consider the *pseudocode* collection `coll`. Pseudocode is code that is not from any specific programming language but is used to illustrate a conceptual point more easily.

```
coll = [ "A", "B", "C", "D", "E" ]
```

The *count* or *length* of `coll` is 5, and the valid indices for the elements would be from `0` to `coll.Count-1` (that is, `0`, `1`, `2`, `3`, and `4`).

```
print( coll.Count );  // 5

print( coll[0] );     // A
print( coll[1] );     // B
print( coll[2] );     // C
print( coll[3] );     // D
print( coll[5] );     // E

print( coll[6] );     // Index Out of Range Exception!!!
```

If you try to use bracket access to access an index that is not in range, you will see the following runtime error:

IndexOutOfRangeException: Array index is out of range.

It is important to keep this in mind as you're working with any collection in C#.

As always, remember to save your script in MonoDevelop when you're done editing. Then, switch back to Unity and select *Main Camera* in the Hierarchy pane. You will see that `List<string> sList` appears in the *ListEx (Script)* component of the Inspector pane. If you play the Unity scene, you can click the disclosure triangle next to `sList` in the Inspector and actually see the values that populate it. Arrays and Lists appear in the Inspector, though Dictionaries and ArrayLists do not.

Important List Properties and Methods

There are many, many properties and methods available for Lists, but these are the most often used. All of these method examples refer to the following `List<string> sL` and are noncumulative. In other words, each example starts with the List `sL` as it is defined in the following three lines, unmodified by the other examples.

```
List<string> sL = new List<string>();
sL.Add( "A" );  sL.Add( "B" );  sL.Add( "C" );  sL.Add( "D" );
// Resulting in the List: [ "A", "B", "C", "D" ]
```

Properties

• `sL[2]` (Bracket access): Returns the element of the List at the index specified by the parameter (`2`). Because `C` is the second element, `sL[2]` this returns: `C`.

• `sL.Count`: Returns the number of elements currently in the List. Because the length of a List can vary over time, `Count` is very important. The last valid index in a List is always `Count-1`. The value of `sL.Count` is `4`, so the last valid index is 3.

Methods

• `sL.Add("Hello")`: Adds the parameter `"Hello"` to the end of `sL`. In this case, `sL` becomes: `[ "A", "B", "C", "D", "Hello" ]`.

• `sL.Clear()`: Removes all existing elements from `sL` returning it to an empty state. `sL` becomes empty: `[ ]`.

• `sL.IndexOf("A")` : Finds the first instance in the `sL` of the parameter `"A"` and returns the index of that element. Because `"A"` is the 0th element of `sL`, this call returns `0`.

If the variable does not exist in the List, a `-1` is returned. This is a safe and fast method to determine whether a List contains an element.

• `sL.Insert(2, "B.5")`: Inserts the second parameter (`"B.5"`) into `sL` at the index specified by the first parameter (`2`). This shifts the subsequent elements of the List forward. In this case, this would cause `sL` to become `[ "A", "B", "B.5", "C", "D" ]`. Valid index values for the first parameter are `0` through `sL.Count`. Any value outside this range will cause a runtime error.

• `sL.Remove("C")`: Removes the specified element from the List. If there happened to be two `"C"`s in the List, only the first would be removed. `sL` becomes `[ "A", "B", "D" ]`.

• `sL.RemoveAt(0)`: Removes the element at the specified index from the List. Because `"A"` is the 0th element of the List, `sL` becomes `[ "B", "C", "D" ]`.

Converting a List to an array

• `sL.ToArray()`: Generates an array that has all the elements of `sL`. The new array will be of the same type as the List. Returns a new string array with the elements `[ "A", "B", "C", "D" ]`.

To move on to learning about Dictionaries, make sure that Unity playback is stopped and then uncheck the check box next to the

name of the *ListEx (Script)* component in the Inspector pane to make the ListEx script inactive (as is shown in Figure 23.1).

Figure 23.1. Clicking the check box to deactivate the ListEx Script component



Dictionary

Dictionaries cannot be viewed in the Inspector, yet they can be a fantastic way to store information. One of the major benefits of a Dictionary is its *constant access time*. This means that no matter how many items you insert into a Dictionary, it will take the same amount of time to find an item. Contrast this with a List or array, where you must iterate over each of the items in the linear collection one by one; as the size of the List or array grows, the amount of time that it will take to find a specific element grows as well, especially if the element you're searching for happens to be the last one in the collection.

Basic Dictionary Creation

Dictionaries pair a *key* and *value*. The key can then be used to access the value. Open the DictionaryEx C# script and enter the following code to see how this works.

```
 1 using UnityEngine;
 2 using System.Collections;
 3 using System.Collections.Generic;                    // a
 4
 5 public class DictionaryEx : MonoBehaviour {
 6     public Dictionary<string,string>    statesDict;          //
b
 7
 8     void Start () {
 9         statesDict = new Dictionary<string, string>
```

```
();                  // c
10
11        statesDict.Add( "MD", "Maryland" );                          //
d
12        statesDict.Add( "TX", "Texas" );
13        statesDict.Add( "PA", "Pennsylvania" );
14        statesDict.Add( "CA", "California" );
15        statesDict.Add( "MI", "Michigan" );
16
17        print("There are "+statesDict.Count+" elements in
statesDict.");  // e
18
19        foreach (KeyValuePair<string,string> kvp in statesDict)
{        // f
20            print( kvp.Key + ": " + kvp.Value );
21        }
22
23        print( "MI is " + statesDict["MI"] );                        // g
24
25        statesDict["BC"] = "British Columbia";                        //
h
26
27        foreach (string k in statesDict.Keys) {                      // i
28            print( k + " is " + statesDict[k] );
29        }
30    }
31
32 }
```

a. You must include the System.Collections.Generic library to use a Dictionary.

b. A Dictionary is declared with both a key and value type in a statement like this. For this Dictionary, both the key and value are strings, but any data type can be used.

c. Like a List, a Dictionary is not ready to be used until it is defined in a statement like this.

d. When you add elements to the Dictionary, you must pass in both a key and a value for each element. These five Add statements add postal codes and state names to the Dictionary for the states where I've lived in my life.

e. As with other C# collections, `Count` can be used to determine how many elements are in a Dictionary. Output:

There are 5 elements in the Dictionary.

f. `foreach` can be used on Dictionaries, but the type of the value that is iterated is a KeyValuePair<,>. The two types of the KeyValuePair<,> must match those of the Dictionary. Output:

MD: Maryland
TX: Texas
PA: Pennsylvania
CA: California
MI: Michigan

g. If you know the key, you can use it to access the value in the Dictionary via bracket access. Output:

MI: Michigan

h. Another way to add values to a Dictionary is to use bracket access as shown here. I also lived in BC for a brief time.

i. The Keys of the Dictionary can also be iterated over using a `foreach` loop. Output:

MD is Maryland
TX is Texas
PA is Pennsylvania
CA is California

MI is Michigan
BC is British Columbia

Save the DictionaryEx script, switch back to Unity, and press Play. You should see the output described. Remember that Dictionaries do not appear in the Unity inspector, so even though `statesDict` is a public variable, you will not see it in the Inspector.

Important Dictionary Properties and Methods

There are also many properties and methods available for Dictionaries, but these are the most often used. All of these method examples refer to the following `Dictionary<int, string> dIS` and are noncumulative. In other words, each example starts with the Dictionary `dIS` as it is defined in the following lines, unmodified by the other examples.

```
Dictionary<int,string> dIS;
dIS = new Dictionary<int, string>();
dIS[0] = "Zero";
dIS[1] = "One";
dIS[10] = "Ten";
dIS[1234567890] = "A lot!";
```

Another way of writing this same Dictionary declaration and definition would be:

```
dIS = new Dictionary<int, string>() {
    { 0, "Zero" },
    { 1, "One" },
    { 10, "Ten" },
    { 1234567890, "A lot!" }
};
```

This kind of combined declaration and definition of a Dictionary is one of the rare cases where you will see a semicolon at the end of a pair of braces.

Properties

• `dIS[10]` (Bracket access): Returns the element of the Dictionary array at the index specified by the parameter (`10`). Because `"Ten"` is the element at the key `10`, `dIS[10]` returns: “`Ten"`. If you try to access an element with a key that doesn’t exist, you will receive a `KeyNotFoundException` runtime error that will crash your code.

• `dIS.Count`: Returns the number of key/value pairs currently in the Dictionary. Because the length of a Dictionary can vary over time, `Count` is very important.

Methods

• `dIS.Add(12,"Dozen")`: Adds the value `"Dozen"` to the Dictionary at the key `12`.

• `dIS[13] = "Baker's Dozen"`: Adds the value `"Baker's Dozen"` to the Dictionary at the key `13`. If you use bracket access to set an already-existing, key, it will replace the value. For example `dIS[0] = "None"` would replace the value at key `0` with `"None"`.

• `dIS.Clear()`: Removes all existing key/value pairs from `dIS`, leaving it empty.

• `dIS.ContainsKey(1)` : Returns true if the key `1` is in the Dictionary. This is a very fast call because the Dictionary is designed to find things quickly by key. Keys in the Dictionary are exclusive, so you can only have one value for each key.

• `dIS.ContainsValue("A lot!")` : Returns true if the value `"A lot!"` is in the Dictionary. This is a slow call because a Dictionary is optimized to find things by key, not value. Values are also non-exclusive, meaning that several keys could hold the similar values.

• `dIS.Remove(10)` : Removes the key/value pair at the key `10` from the Dictionary.

It is sometimes desirable to set the equivalent of something like a Dictionary in the Inspector. In that case, I often create a List of a complex class that includes both a key and a value. You can see an example of this in Chapter 31, "Space SHMUP Expansion."

Before moving on to looking at arrays, make sure that Unity playback is stopped and then uncheck the check box next to the name of the *DictionaryEx (Script)* component in the Inspector pane to make the DictionaryEx script inactive

Array

An array is the simplest collection type, which also makes it the fastest. Arrays do not require any libraries to be imported (via the `using` command) to work because they are built into core C#. In addition, arrays have multidimensional and jagged forms that can be very useful.

Basic Array Creation

Arrays are of a fixed length that must be determined when the array is defined. Double-click the ArrayEx C# script in the Project pane to open it in MonoDevelop and enter the following code:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class ArrayEx : MonoBehaviour {
 5     public string[]       sArray;                          // a
 6
 7     void Start () {
 8         sArray = new string[10];                           // b
 9
10         sArray[0] = "These";                               // c
11         sArray[1] = "are";
12         sArray[2] = "some";
```

```
13        sArray[3] = "words";
14
15        print( "The length of sArray is: "+sArray.Length
);            // d
16
17        string str = "";
18        foreach (string sTemp in sArray) {                    // e
19            str += "|"+sTemp;
20        }
21        print( str );
22    }
23 }
```

a. Unlike a List or Dictionary, an array in C# isn't actually a separate data type, rather it's a collection formed from any existing data type by adding brackets after the type name. The type of `sArray` above is not declared as `string`; it is `string[]`, a collection of multiple `strings`. Note: While `sArray` is declared to be an array here, its length is not yet set.

b. Here, `sArray` is defined as a `string[]` with a length of 10. When an array is defined, its length is filled with elements of the default value for that data type. For `int[]` or `float[]`, the default would be `0`. For `string[]` and other complex data types like `GameObject[]`, each element of the array is filled with `null` (which indicates no that value has been assigned).

c. Rather than using the `Add()` method like Lists, standard arrays use bracket access for assignment of value as well as retrieval of value from the array.

d. Rather than using `Count` like the generic C# collections, arrays use the property `Length`. It is important to note (as you can see from the preceding code output) that `Length` returns the entire length of the array, including both defined elements (for example, `sArray[0]` through `sArray[3]` above) and elements that are empty (that is, still their default, undefined value as are `sArray[4]` through `sArray[9]` in the previous code). Output:

The length of sArray is 10.

e. `foreach` works for arrays just as it does for other C# collections. The only difference is that the array may have empty or null elements, and `foreach` will iterate over them. Output:

|These|are|some|words||||||

When you run the code, be sure to have Main Camera selected in the Hierarchy pane. This will enable you to open the disclosure triangle next to `sArray` in the ArrayEx (Script) component of the Inspector pane and see the elements of `sArray`.

The code output looks like this:

The length of sArray is: 10
|These|are|some|words||||||

Empty Elements in the Middle of an Array

One thing allowed by arrays that is not possible in Lists is an empty element in the middle of an array. This would be useful in a game if you had something like a scoring track where each player had a marker on the track but it was possible to have empty spaces in between the markers.

Modify lines 12 and 13 of the previous code as follows:

```
10        sArray[0] = "These";
11        sArray[1] = "are";
12        sArray[3] = "some";
13        sArray[6] = "words";
```

The code output would look like this: `|These|are||some|||words|||`

As you can see from the output, `sArray` now has empty elements at indices 2, 4, 5, 7, 8, and 9. As long as the index (for example, 0, 1, 3, and 6 here) of the assignment is within the valid range for the array,

you can use bracket access to place a value anywhere in the array, and the `foreach` loop will handle it gracefully.

Attempting to assign a value to an index that outside of the defined range for the array (for example, `sArray[10] = "oops!";` or `sArray[99] = "error!";`) will lead to the following runtime error: `IndexOutOfRangeException: Array index is out of range.`

Attempting to access a non-existent array index will also give you the same runtime error. For example `print( sArray[20] );` would give you an `IndexOutOfRangeException`.

Return the code back to its original state:

```
10        sArray[0] = "These";
11        sArray[1] = "are";
12        sArray[2] = "some";
13        sArray[3] = "words";
```

Empty Array Elements and `foreach`

Play the project again and look at the output, which has returned to its previous state:

|These|are|some|words|||||

The `str += "|"+sTemp;` statement on line 19 concatenates (that is, adds) a pipe `(|)` to the end of `str` before each element of the array. Even though `sArray[4]` through `sArray[9]` are still the default value of `null`, they are counted by foreach and iterated over. This is a good opportunity to use a `break` jump statement to escape the foreach loop early. Modify the code as follows:

```
18      foreach (string sTemp in sArray) {
19          str += "|"+sTemp;
20          if (sTemp == null) break;
21      }
```

The new code output is as follows: `|These|are|some|words|`

When C# iterates over `sArray[4]`, it will still concatenate `"|"+null` onto the end of `str` but will then check `sArray[4]`, see that it is `null`, and break out of the foreach loop, preventing it from iterating over `sArray[5]` through `sArray[9]`.

Important Array Properties and Methods

Of the properties and methods available for arrays, these are the most often used. All of these refer to the following array and are noncumulative.

```
string[] sA = new string[] { "A", "B", "C", "D" };
// Resulting in the Array: [ "A", "B", "C", "D" ]
```

Here you see the array initialization expression that allows the declaration, definition, and population of an array in a single line (as opposed to the array definition statement shown in the code listing under *Basic Array Creation* above). Note that when using this form of the array initialization expression, the `Length` of the array is implied by the number of elements between the braces and does not need to be specified; in fact, if you use braces to define an array, you cannot use the brackets in the array declaration to specify a length that is different from the number of elements between the braces.

Properties

• `sA[2]` (bracket access): Returns the element of the array at the index specified by the parameter (`2`). Because `"C"` is the second element of `sA`, this returns: `"C"`.

If the index parameter is outside of the valid range of the array (which for `sA` is 0 to 3), it will generate an `IndexOutOfRangeException` runtime error.

• `sA[1] = "Bravo"` (bracket access used for assignment): Assigns the value on the right side of the `=` assignment operator to the specified position in the array, replacing the previous value. `sA` would become `[ "A", "Bravo", "C", "D" ]`.

If the index parameter is outside of the valid range of the array, it will generate an `IndexOutOfRangeException` runtime error.

• `sA.Length`: Returns the total capacity of the array. Elements will be counted regardless of whether they have been assigned or are still default values. Returns `4`.

Static Methods

The static methods here are part of the System.Array class (that is defined by the System.Collections library) and can act on arrays to give them some of the abilities of Lists.

• `System.Array.IndexOf( sA, "C" )`: Finds the first instance in the array `sA` of the element `"C"` and returns the index of that element. Because `"C"` is the second element of `sA`, this returns `2`.

If the variable does not exist in the Array, a `-1` is returned. This is often used to determine whether an array contains a specific element.

• `System.Array.Resize(` ref `sA, 6 )`: This is a C# method that adjusts the length of an array. The first parameter is a reference to the array instance (which is why the `ref` keyword is required), and the second parameter is the new length. `sA` would then become `[ "A", "B", "C", "D", null, null ]`.

If the second parameter specifies a Length that is shorter than the original array, the extra elements will be culled. `System.Array.Resize( ref sA, 2 )` would cause `sA` to become `[ "A", "B" ]`. `System.Array.Resize()` does not work for the multidimensional arrays described later in this chapter.

Converting an Array to a List

• `List<string> sL = new List<string>( sA )`: This line will create a List `sL` that duplicates all the elements of `sA`.

It is also possible to use the array initialization expression to declare, define, and populate a List in one line, but it's a little convoluted:

List<string> sL = new List<string>(new string[] { "A", "B", "C" });

This declares, defines, and populates an anonymous new `string[]` array that is immediately passed into the `new List<string>()` function.

To prepare for the next example, deactivate the ArrayEx script by clicking the check box next to its name in the Inspector pane for Main Camera.

Multidimensional Arrays

It is possible—and often useful—to create multidimensional arrays that have two or more indices. This means that instead of just one index number in the brackets, the array could use two or more. This would be useful for creating a two-dimensional grid that could hold one item in each grid square.

Create a new C# script named Array2dEx and attach it to Main Camera. Open Array2dEx in MonoDevelop and enter the following code:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Array2dEx : MonoBehaviour {
 5
 6     public string[,]       sArray2d;
```

```
 7
 8     void Start () {
 9         sArray2d = new string[4,4];
10
11         sArray2d[0,0] = "A";
12         sArray2d[0,3] = "B";
13         sArray2d[1,2] = "C";
14         sArray2d[3,1] = "D";
15
16         print( "The Length of sArray2d is: "+sArray2d.Length );
17     }
18 }
```

The code will yield the following output: `The Length of sArray2d is: 16`

As you can see, `Length` is still only a single int, even though this is a multidimensional array. `Length` here is now just the total number of elements in the array, so it is the coder's responsibility to keep track of each separate dimension of the array.

Now, let's create a nicely formatted output of the values in sArray2d array. When we're done, it should look something like this:

```
|A| | |B|
| | |C| |
| | | | |
| |D| | |
```

As you can see above, the A is in the 0th row, 0th column ([0,0]), the B is in the 0th row, 3rd column ([0,3]) and so on. To implement this, add the following bolded lines to the code:

```
16         print( "The Length of sArray2d is: "+sArray2d.Length );
17
18         string str = "";
19         for ( int i=0; i<4; i++ ) {                              // a
20             for ( int j=0; j<4; j++ ) {
```

```
21              if (sArray2d[i,j] != null) {                    // b
22                  str += "|"+sArray2d[i,j];
23              } else {
24                  str += "|_";
25              }
26          }
27          str += "|"+"\n";                                    // c
28      }
29      print( str );
30  }
31 }
```

a. Lines 19 and 20 demonstrate the use of two nested for loops to iterate over a multidimensional array. When nested in this manner, the code will:

1. Start with `i`=0

2. Iterate over all `j` values from 0 to 3

3. Increment to `i`=1

4. Iterate over all `j` values from 0 to 3

5. Increment to `i`=2

6. Iterate over all `j` values from 0 to 3

7. Increment to `i`=3

8. Iterate over all `j` values from 0 to 3

This will guarantee that the code moves through the multidimensional array in an orderly manner. Keeping the grid example, it will move through all the elements in a row (by incrementing `j` from 0 to 3) and then advance to the next row by incrementing `i` to the next value.

b. Lines 21–25 check to see whether the string at `sArray[i,j]` has a value other than `null`. If so, it concatenates a pipe and `sArray2d[i,j]` onto `str`. If the value is `null`, a pipe and one space are concatenated onto `str`. The pipe character is found on the keyboard above the return (or enter) key. It is usually shift–backslash (\ ).

c. This line occurs after all of the iterations of the `j` for loop but before the next iteration of the `i` for loop. The effect of it is to concatenate a trailing pipe and carriage return (i.e., line break) onto `str`, giving the output the nice formatting of a line for each iteration of the `i` for loop.

The code produces the following output, though you will only see the first couple of lines in the Unity Console pane.

```
The Length of sArray2d is: 16

|A| | |B|
| | |C| |
| | | | |
| |D| | |
```

Just looking at the output in the Console pane of Unity, you will only see the top two lines of the `sArray2d` grid array listed in the output. However, if you click that line in the Console pane, you will see that more data is revealed in the bottom half of the Console pane (see Figure 23.2).

Figure 23.2. Clicking an output message in the Console causes an expanded view to appear below. Note that the first line of the most recent Console message is also shown in the lower-left corner of the Unity window.



As you can see in the figure, the fancy text formatting that we did doesn't line up as well in the Console pane because it uses a non-

monospaced font (that is, a font where an *i* has a different width than an *m;* `in monospaced fonts, i and m have the same width`). You can click any line in the Console pane and choose *Edit* > *Copy* from the menu bar to copy that data and then paste it into another program. This is something that I do often, and I most commonly past into a text editor. (I prefer *TextWrangler*[1] on Mac or *EditPad Pro*[2] on Windows, both of which are quite powerful.)

[1] *TextWrangler* is available for free from BareBones Software: http://www.barebones.com.

[2] *EditPad Pro* has a free trial available from Just Great Software: http://editpadpro.com.

You also should be aware that the Unity Inspector pane does not display multidimensional arrays. In fact, similar to Dictionaries, if the Inspector does not know how to properly display a variable, it will completely ignore it, so not even the name of a public multidimensional array will appear in the Inspector pane.

Stop Unity's execution by clicking the Play button again (so that it is not blue) and then use the Main Camera Inspector to disable the *Array2dEx (Script)* component.

Jagged Arrays

A jagged array is an array of arrays. This is similar to the multidimensional array, but it allows the subarrays to be different lengths. We'll create a jagged array that holds the following data:

```
| A | B | C | D |
| E | F | G |
| H | I |
| J |   |   | K |
```

As you can see, the 0th and 3rd rows each contain four elements, while rows 1 and 2 contain three and two elements respectively.

Note that null elements are allowed as is shown in the 3rd row. In fact, it is also possible for an entire row to be null (though that would cause an error on the particular code we have on line 32 of the following code listing).

Create a new C# script named JaggedArrayEx and attach it to Main Camera. Open JaggedArrayEx in MonoDevelop and enter the following code:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class JaggedArrayEx : MonoBehaviour {
 5     public string[][]       jArray;                          // a
 6
 7     void Start () {
 8         jArray = new string[4][];                            // b
 9
10         jArray[0] = new string[4];                           // c
11         jArray[0][0] = "A";
12         jArray[0][1] = "B";
13         jArray[0][2] = "C";
14         jArray[0][3] = "D";
15
16         // The following lines use single-line Array
initialization      // d
17         jArray[1] = new string[] { "E", "F", "G" };
18         jArray[2] = new string[] { "H", "I" };
19
20         jArray[3] = new string[4];                           // e
21         jArray[3][0] = "J";
22         jArray[3][3] = "K";
23
24         print( "The Length of jArray is: "+jArray.Length
);             // f
25         // Outputs: The Length of jArray is: 4
26
```

```
27        print( "The Length of jArray[1] is: "+jArray[1].Length
);      // g
28        // Outputs: The Length of jArray[1] is: 3
29
30        string str = "";
31        foreach (string[] sArray in jArray) {                    // h
32            foreach( string sTemp in sArray ) {
33                if (sTemp != null) {
34                    str += " | "+sTemp;                          // i
35                } else {
36                    str += " |  ";                               // j
37                }
38            }
39            str += " | \n";
40        }
41
42        print( str );
43    }
44 }
```

a. Line 5 declares `jArray` as a jagged array (that is, an array of arrays). Where a `string[]` is a collection of strings, a `string[][]` is a collection of string arrays (or `string[]`s).

b. Line 8 defines `jArray` as a jagged array with a length of 4. Note that the second set of brackets is still empty, denoting that the subarrays can be of any length.

c. Line 10 defines the 0th element of `jArray` to be an array of strings with a length of 4.

d. Lines 17 and 18 use the single-line form of array definition. Because the elements of the array are defined between the braces, the length of the array does not need to be explicitly stated (hence the empty brackets in new `string[]`).

e. Lines 20–22 define the 3rd element of `jArray` to be a `string[]` with a length of 4 and then fill only the 0th and 3rd elements of that

`string[]`, leaving elements 1 and 2 null.

f. Line 24 outputs `"The Length of jArray is: 4"`. Because `jArray` is an array of arrays (rather than a multidimensional array), `jArray.Length` counts only the number of elements that can be accessed via the first set of brackets.

g. Line 27 outputs `"The Length of jArray[1] is: 3"`. Because `jArray` is an array of arrays, subarray `Length` can also now be easily determined.

h. In jagged arrays, `foreach` works separately on the array and sub-arrays. foreach on `jArray` will iterate through the four `string[]` (string array) elements of `jArray`, and `foreach` on any of those `string[]`s will iterate over the strings within. Note that `sArray` is a `string[]` (string array) and that `sTemp` is a string.

As was mentioned previously, line 32 would throw a *null reference error* if one of the rows of `jArray` were null. In that case, `sArray` would be null, and trying to run the foreach statement in line 32 on a null variable would lead to a *null reference*, the attempt to reference an element of something that is null. The foreach statement would be attempting to access data of `sArray` like `sArray.Length` and `sArray[0]`. Because null data have no elements or value, accessing things like null.Length throws an error.

i. On a keyboard, the string literal in line 34 is typed: space pipe space.

j. On a keyboard, the string literal in line 36 is typed: space pipe space space.

The code outputs the following to the Console pane:

```
The Length of jArray is: 4
The Length of jArray[1] is: 3
 | A | B | C | D |
 | E | F | G |
```

```
| H | I |
| J |   |   | K |
```

Using for Loops Instead of foreach for Jagged Arrays

It is also possible to use for loops based on the `Length` of the array and subarrays. The foreach loop in the preceding code listing could be replaced with this code:

```
31        string str = "";
32        for (int i=0; i<jArray.Length; i++) {
33            for (int j=0; j<jArray[i].Length; j++) {
34                str += " | "+jArray[i][j];
35            }
36            str += " | \n";
37        }
```

This code produces the exact same output as the foreach loops shown earlier. Whether you choose to use for or foreach will depend on the situation.

Jagged Lists

As a final note on jagged collections, it is also possible to create jagged Lists. A jagged two-dimensional list of strings would be declared `List<List<string>> jaggedStringList`. Just as with jagged arrays, the subLists would initially be null, so you would have to initialize them as you added them as shown in the following code. Just like all Lists, jagged Lists do *not* allow empty elements. Create a new C# script named JaggedListTest, attach it to Main Camera, and enter this code:

```
 1 using UnityEngine;
 2 using System.Collections.Generic;                    // a
 3
 4 public class JaggedListTest : MonoBehaviour {
 5
```

```
 6    public List<List<string>> jaggedList;
 7
 8    // Use this for initialization
 9    void Start () {
10        jaggedList = new List<List<string>>();
11
12        // Add a couple List<string>s to jaggedList
13        jaggedList.Add( new List<string>() );
14        jaggedList.Add( new List<string>() );
15
16        // Add two strings to jaggedList[0]
17        jaggedList[0].Add ("Hello");
18        jaggedList[0].Add ("World");
19
20        // Add a third List<string> to jaggedList, including data
21        jaggedList.Add ( new List<string>( new string[]
{"complex",
          Image"initialization"} ) );                          // b
22
23        string str = "";
24        foreach (List<string> sL in jaggedList) {
25            foreach (string sTemp in sL) {
26                if (sTemp != null) {
27                    str += " | "+sTemp;
28                } else {
29                    str += " |  ";
30                }
31            }
32            str += " | \n";
33        }
34        print( str );
35    }
36
37 }
```

a. Though using System.Collections; is included in all Unity C# scripts by default, it's not actually necessary (though

System.Collections.Generic is required for Lists).

b. This is one of the first instances in this book of the Image code continuation character. This is used throughout the book when a single line is longer than can fit the width of the page. You should not try to type the Image character, rather it is there to let you know to continue typing the single line as if there were no line break. With no leading tabs, line 21 would appear as follows:

```
jaggedList.Add( new List<string>( new string[]
{"complex","initialization"} ) );
```

The code outputs the following to the Console pane:

```
 | Hello | World |
 |
 | complex | initialization |
```

Whether to Use Array or List

Arrays and Lists are very similar, so people are often unsure which one to use in any given situation. The primary differences between the array and List collections types are as follows:

• List has flexible length, whereas array length is more difficult to change.

• Array is very slightly faster.

• Array allows multidimensional indices.

• Array allows empty elements in the middle of the collection.

Because they are simpler to implement and take less forethought (due to their flexible length), I personally tend to use Lists much more often than arrays. This is especially true when prototyping games, since prototyping requires a lot of flexibility.

Summary

Now that you have a handle on Lists, Dictionaries, and arrays, it will be possible for you to work easily with large numbers of objects in your games. For example, you could return to the Hello World project from Chapter 19, "Hello World: Your First Program," and add a `List<GameObject>` to the CubeSpawner code that had every new cube added to it at the time the cube was instantiated. This would give you a reference to each cube, allowing you to manipulate the cube after it was created. The exercise below shows you how to do so.

Summary Exercise

In this exercise, we return to the Hello World project from Chapter 19 and write a script that will add each new cube created to a `List<GameObject>` named `gameObjectList`. Every frame that a cube exists, it will be scaled down to 95% of its size in the previous frame. Once a cube has shrunk to a scale of 0.1 or less, it will be deleted from the scene and `gameObjectList`.

However, if we delete an element from `gameObjectList` while the foreach loop is iterating over it, this will cause an error. To avoid this, the cubes that need to be deleted will be temporarily stored in another list named `removeList`, and then the list will be iterated over to remove them from `gameObjectList`. (You'll see what I mean in the code.)

Open your Hello World project and create a new scene (File > Scene from the menu bar). Save the scene as *_Scene_3*. Create a new script named CubeSpawner3 and attach it to the Main Camera in the scene. Then, open CubeSpawner3 in MonoDevelop and enter the following code:

```
1 using UnityEngine;
2 using System.Collections;
3 using System.Collections.Generic;
```

```
 4
 5 public class CubeSpawner3 : MonoBehaviour {
 6     public GameObject       cubePrefabVar;
 7     public List<GameObject>  gameObjectList; // Will hold all
the Cubes
 8     public float            scalingFactor = 0.95f;
 9     // ^ Amount that each cube will shrink each frame
10     public int               numCubes = 0; // Total # of Cubes
instantiated
11
12     // Use this for initialization
13     void Start() {
14         // This initializes the List<GameObject>
15         gameObjectList = new List<GameObject>();
16     }
17
18     // Update is called once per frame
19     void Update () {
20         numCubes++; // Add to the number of
Cubes                    // a
21
22         GameObject gObj = Instantiate( cubePrefabVar ) as
GameObject;    // b
23
24         // These lines set some values on the new Cube
25         gObj.name = "Cube "+numCubes;                          //
c
26         Color c = new Color(Random.value, Random.value,
Random.value);   // d
27         gObj.renderer.material.color = c;
28         // ^ Gives the Cube a random color
29         gObj.transform.position =
Random.insideUnitSphere;              // e
30
31         gameObjectList.Add (gObj); // Add gObj to the List of
Cubes
32
```

```
33        List<GameObject> removeList = new List<GameObject>
();          // f
34        // ^ This removeList will store information on Cubes that
should be
35        //    removed from gameObjectList
36
37        // Iterate through each Cube in gameObjectList
38        foreach (GameObject goTemp in gameObjectList)
{               // g
39
40            // Get the scale of the Cube
41            float scale =
goTemp.transform.localScale.x;                // h
42            scale *= scalingFactor; // Shrink it by the
scalingFactor
43            goTemp.transform.localScale = Vector3.one * scale;
44
45            if (scale <= 0.1f) { // If the scale is less than 0.1f...    // i
46               removeList.Add (goTemp); // ...then add it to the
removeList
47            }
48        }
49
50        foreach (GameObject goTemp in removeList)
{                   // g
51            gameObjectList.Remove (goTemp);                          //
j
52            // ^ Remove the Cube from gameObjectList
53            Destroy (goTemp);  // Destroy the Cube's GameObject
54        }
55
56    }
57 }
```

a. The increment operator (++) is used to increase the total number of cubes that have been created.

b. An instance of `cubePrefabVar` is instantiated. The words "`as GameObject`" are necessary because `Instantiate()` can be used on any kind of object (meaning that C# has no way of knowing what kind of data `Instantiate()` will return). The "`as GameObject`" tells C# that this object should be treated as a GameObject.

c. The `numCubes` variable is used to give unique names to each cube. The first cube will be named *Cube 1*, the second *Cube 2*, and so on.

d. Lines 26 and 27 assign a random color to each cube. Colors are accessed through the material attached to the GameObject's Renderer, as is demonstrated on line 27.

e. `Random.insideUnitSphere` returns a random location that is inside a sphere with a radius of 1 (centered on the point [0,0,0]). This code makes the cubes spawn at a random location near [0,0,0] rather than all at exactly the same point.

f. As is stated in the code comments, `removeList` will be used to store cubes that will need to be removed from `gameObjectList`. This is necessary because C# does not allow you to remove elements from a list in the middle of a `foreach` loop that is iterating over the list. (That is, it is not possible to call `gameObjectList.Remove()` anywhere within the `foreach` loop on lines 38–48 that iterates over `gameObjectList`.)

g. This `foreach` loop iterates over all of the cubes in `gameObjectList`. Note that the temporary variable created for the foreach is `goTemp`. `goTemp` is also used in the `foreach` loop on line 50, so `goTemp` is declared on both lines 38 and 50. Because `goTemp` is locally scoped to the `foreach` loop in each case, there is no conflict caused by declaring the variable twice in the same `Update()` function. See "Variable Scope" in Appendix B, "Useful Concepts," for more information.

h. Lines 41–43 get the current scale of a cube (by getting the x dimension of its `transform.localScale`), multiply that scale by 95%, and then set the `transform.localScale` to this new value. Multiplying

a Vector3 by a float (as is done on line 43) multiplies each individual dimension by that same number, so [2,4,6] * 0.5f would yield [1,2,3].

i. As mentioned in the code comments, if the newly reduced scale is less than 0.1f, the cube will be added to `removeList`.

j. The `foreach` loop from lines 50–54 iterates over `removeList` and removes any cube that is in `removeList` from `gameObjectList`. Because the `foreach` is iterating over `removeList`, it's perfectly fine to remove elements from `gameObjectList`. The removed cube GameObject still exists on screen until the `Destroy()` command is used to destroy it. Even then, it still exists in the computer's memory because it is still an element of `removeList`. However, because `removeList` is a local variable scoped to the `Update()` function, once the `Update()` function is complete, `removeList` will cease to exist, and then any objects that are exclusively stored in `removeList` will also be deleted from memory.

Save your script and then switch back to Unity. You must assign *Cube Prefab* from the Project pane to the cubePrefabVar variable in the *Main Camera:CubeSpawner3 (Script)* component of the Main Camera Inspector if you want to actually instantiate any cubes.

After you have done this, press Play in Unity, and you should see that a number of cubes spawn in as they did in previous versions of Hello World. However, they spawn in different colors, they shrink over time, and they are eventually destroyed (instead of existing indefinitely as they did in earlier versions).

Because the CubeSpawner3 code keeps track of each cube through the `gameObjectList`, it is able to modify each cube's scale every frame and then destroy each cube once it's smaller than a scale of 0.1f. At a `scalingFactor` of 0.95f, it takes each cube 45 frames to shrink to a scale <= 0.1f, so what would be the 0th cube in `gameObjectList` is always removed and destroyed for being too small, and the `Count` of `gameObjectList` stays at 45.

Moving Forward

In the next chapter, you learn how to create and name functions other than `Start()` and `Update()`.

Chapter 24. Functions and parameters

In this chapter, you learn to take advantage of the immense power of functions. You write your own custom functions that can take various kinds of variables as input arguments and can return a single variable as the function's result. We also explore some special cases of parameters for function input like function overloading, optional parameters, and the `params` keyword modifier, all of which will help you to write more effective, modular, reusable, and flexible code.

Set Up the Function Examples Project

In Appendix A, "Standard Project Setup Procedure," detailed instructions show you how to set up Unity projects for the chapters in this book. At the start of each project, you will also see a sidebar like the one here. Please follow the directions in the sidebar to create the project for this chapter.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. For information on the standard project setup procedure, see Appendix A.

- **Project name:** Function Examples
- **Scene name:** _Scene_Functions
- **C# Script names:** CodeExample

Attach the script CodeExample to the Main Camera in the scene _Scene_Functions.

Definition of a Function

You've actually been writing functions since your first Hello World program, but up until now, you've been adding content to built-in Unity MonoBehaviour functions like `Awake()`, `Start()`, and `Update()`. From now on, you'll also be writing custom functions.

The best way to think about a function is as a chunk of code that does something. For instance, to count the number of times that `Update()` has been called, you can create a new C# script with the following code (you will need to add the bold lines):

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CodeExample : MonoBehaviour {
 5
 6    public int       numTimesCalled = 0;                          // a
 7
 8    void Update() {
 9       numTimesCalled++;                                          // b
10        PrintUpdates();                                         // c
11    }
12
13    void PrintUpdates() {                                        // d
14        string outputMessage = "Updates:
"+numTimesCalled;              // e
15        print( outputMessage ); // Output example: "Updates:
1"          // f
16    }
17
18 }
```

a. Declares the public variable `numTimesCalled` and defines it to initially have the value 0. Because `numTimesCalled` is declared as a public variable inside the class `CodeExample` but outside of any

function, it is available to be accessed by any of the functions within the `CodeExample` class.

b. `numTimesCalled` is incremented (1 is added to it).

c. Line 10 *calls* the function `PrintUpdates()`. When your code calls a function, it causes the function to be executed. This will be described in more detail soon.

d. Lines 13 *declares* the function `PrintUpdates()`. Declaring a function is similar to declaring a variable. `void` is the return type of the function, meaning that the function is not expected to return a value (as will be covered in more detail soon). Lines 13–16 collectively *define* the function. All lines of code between the opening brace `{` on line 13 and the closing brace `}` on line 16 are part of the definition of `PrintUpdates()`.

Note that it the order in which your functions are declared in the class does not matter. Whether `PrintUpdates()` or `Update()` is declared first is irrelevant as long as they are both within the braces of the class `CodeExample`. C# will look through all the declarations in a class before running any code. It's perfectly fine for `PrintUpdates()` to be called on line 10 and declared on line 13 because both `PrintUpdates()` and `Update()` are functions declared in the class `CodeExample`.

e. Line 14 defines a *local* string variable named `outputMessage`. Because `outputMessage` is defined within the function `PrintUpdates()` its *scope* is limited to `PrintUpdates()`, meaning that the variable name `outputMessage` has no value outside of the function `PrintUpdates()`. For more information about variable scope, see the "Variable Scope" section of Appendix B, "Useful Concepts."

Line 14 also defines `outputMessage` to be the concatenation of `"Updates: "` and the public integer `numTimesCalled`.

f. The Unity function `print()` is called with the single *argument* `outputMessage`. This prints the value of `outputMessage` to the Unity

Console. Function arguments are covered later in this chapter.

In an actual game, `PrintUpdates()` would not be a terribly useful function, but it does showcase two of the important concepts covered in this chapter.

• **Functions encapsulate actions:** A function can be thought of as a named collection of several lines of code. Every time the function is called, those lines of code are executed. This was demonstrated by both `PrintUpdates()` and the `BuySomeMilk()` example from Chapter 18, "Introducing Our Language: C#."

• **Functions contain their own scope:** As you can read in the "Variable Scope" section of Appendix B, "Useful Concepts," variables declared within a function have their scope limited to that function. Therefore, the variable `outputMessage` (declared on line 14) has a scope limited to just the function `PrintUpdates()`. This can either be stated as "`outputMessage` is scoped to the function `PrintUpdates()`" or "`outputMessage` is *local to* the function `PrintUpdates()`."

Contrast the scope of `outputMessage` with that of the public variable `numTimesCalled`, which has a scope of the entire `CodeExample` class and can be used by any function in `CodeExample`.

If you run this code in Unity, you will see that `numTimesCalled` is incremented every frame and `PrintUpdates()` is called every frame (which outputs the value of `numTimesCalled` to the Console pane). Calling a function causes it to execute, and when the function is done, execution then returns to the point from where it was called. So, in the class `CodeExample`, the following happens every frame:

1. Every frame, the Unity engine calls the `Update()` function (line 8).

2. Then, line 9 increments `numTimesCalled`.

3. Line 10 calls `PrintUpdates()`.

4. Execution then jumps to the beginning of the `PrintUpdates()` function on line 13.

5. Lines 14 and 15 are executed.

6. When Unity reaches the closing brace of `PrintUpdates()` on line 16, execution returns to line 10 (the line from which it was called).

7. Execution continues to line 11, which ends the `Update()` function.

The remainder of this chapter covers both simple and complex uses of functions, and it's an introduction to some rather complicated concepts. As you continue into the tutorials later in this book, you'll get a much better understanding of how functions work and get more ideas for your own functions, so if there's anything that doesn't make sense the first time through this chapter, that's okay. You can return to it once you've read a bit more of the book.

Using Code From This Chapter in Unity

Though the first code listing in this chapter includes all of the lines of the `CodeExample` class, later code examples do not. If you want to actually run the rest of the code from this chapter in Unity, you will need to wrap it inside of a class. Classes are covered in detail in Chapter 26, "Classes," but for now, you can accomplish this by adding the bolded lines that follow around any of the code listed in this chapter:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CodeExample : MonoBehaviour {
 5
     // The code listing would replace this comment
16
17 }
```

For example, without the bold lines listed immediately above, the first code listing in this chapter would have looked like this:

```
 6 public int        numTimesCalled = 0;
 7
 8 void Update() {
 9     PrintUpdates();
10 }
11
12 void PrintUpdates() {
13     numTimesCalled++;
14     print( "Updates: "+numTimesCalled );  // Example: "Updates: 5"
15 }
```

If you wanted to enter this listing of lines 6–15 into a C# script in MonoDevelop, you would need to add the bold lines from the previous listing around them. The final version of code in MonoDevelop would be the code listing on the first page of this chapter.

The remainder of the code listings in this chapter will have numbering that starts on line 6 to indicate that other lines would need to precede and follow the lines shown in the code listings of the book.

Function Parameters and Arguments

Some functions are called with empty parentheses following them (for example, `PrintUpdates()` in the first code listing). Other functions can be passed information in between the parentheses (for example, `Say("Hello")` in the following listing). When a function is designed to receive outside information via the parentheses like this, the type of information is specified by one or more *parameters* that create a local function variable (with a specific type) to hold the information. In line 10 of the following code listing, `void Say( string sayThis )` declares a parameter named `sayThis` that is of the string type.

`sayThis` can then be used as a local variable within the `Say()` function.

When information is sent to a function via its parameters, it is referred to as *passing* information to the function. The information passed is called an *argument*. In line 7 of the following listing, the function `Say()` is called with the argument `"Hello"`. Another way to say this is that `"Hello"` is passed to the function `Say()`. The argument passed to a function must match the parameters of the function, or it will cause an error.

```
 6 void Awake() {
 7     Say("Hello");                                    // a
 8 }
 9
10 void Say( string sayThis ) {                         // b
11     print(sayThis);
12 }
```

a. When `Say()` is called by line 7, the string literal `"Hello"` is passed into the function `Say()` as an argument, and line 10 then sets the value of `sayThis` to `"Hello"`.

b. The string `sayThis` is declared as a parameter variable of the function `Say()`. This makes `sayThis` a local variable that is scoped to the function `Say()`, in other words, the variable `sayThis` does not exist outside of the function `Say()`.

In the function `Say()` in the previous listing, we've added a single parameter named `sayThis`. Just as with any other variable declaration, the first word, `string`, is the variable type and the second word, `sayThis`, is the name of the variable.

Just like other local function variables, the parameter variables of a function disappear from memory as soon as the function is complete; if the parameter `sayThis` were used anywhere in the `Awake()` function, it would cause a compiler error due to `sayThis` being exclusively limited in scope to the function `Say()`.

In line 7 of this (preceding) code listing, the argument passed into the function is the string literal `"Hello"`, but any type of variable or literal can be passed into a function as an argument as long as it matches the type of the parameter(s) of the function (for example, line 7 of the following code listing, passes `this.gameObject` as an argument to the function `PrintGameObjectName()`). If a function has multiple parameters, arguments passed to it must be separated by commas (see line 8 in the following code listing).

```
 6 void Awake() {
 7     PrintGameObjectName( this.gameObject );
 8     SetColor( Color.red, this.gameObject );
 9 }
10
11 void PrintGameObjectName( GameObject go ) {
12     print( go.name );
13 }
14
15 void SetColor( Color c, GameObject go ) {
16     Renderer r = go.renderer;
17     r.material.color = c;
18 }
```

Returning Values

In addition to receiving values as parameters, functions can also return back a single value, known as the *result* of the function as shown on line 13 of the following code listing.

```
 6 void Awake() {
 7     int num = Add( 2, 5 );
 8     print( num ); // Prints the number 7 to the Console
 9 }
10
11 int Add( int numA, int numB ) {
12     int sum = numA + numB;
```

```
13     return( sum );
14 }
```

In this example, the function `Add()` has two parameters, the integers `numA` and `numB`. When called, it will sum the two integers that were passed in and then return the result. The `int` at the beginning of the function definition declares that `Add()` will be returning an integer as its result. Just as you must declare the type of any variable for it to be useful, you must also declare the return type of a function for it to be used elsewhere in code.

Returning `void`

Most of the functions that we've written so far have had a return type of `void`, which means that no value can be returned by the function. Though these functions don't return a specific value, there are still times that you might want to call `return` within them.

Any time `return` is used within a function, it stops execution of the function and returns execution back to the line from which the function was called. (For example, the `return` on line 16 of the following code listing returns execution back to line 9.)

It is sometimes useful to return from a function to avoid the remainder of the function. For example, if you had a list of over 100,000 GameObjects (e.g., `reallyLongList` in the following code listing), and you wanted to move the GameObject named "Phil" to the origin (Vector3.zero), but didn't care about doing anything else, you could write this function:

```
 6 public List<GameObject> reallyLongList; // Defined in the
Unity Editor   // a
 7
 8 void Awake() {
 9     MoveToOrigin("Phil");                                  // b
10 }
11
```

```
12 void MoveToOrigin(string theName) {
13     foreach (GameObject go in reallyLongList)
{                       // c
14         if (go.name == theName) {                                   // d
15             go.transform.position = Vector3.zero;                       // e
16             return;                                         // f
17         }
18     }
19 }
```

a. `List<GameObject> reallyLongList` is a very long list of GameObjects that we are imagining has been predefined in the Unity Inspector. Because we must imagine this predefined List for this example, entering this code into Unity would not work unless you defined `reallyLongList` yourself.

b. The function `MoveToOrigin()` is called with the string literal `"Phil"` as its argument.

c. The `foreach` statement iterates over `reallyLongList`.

d. If a GameObject with the name `"Phil"` (i.e., `theName`) is found...

e. ...then it is moved to the position [0,0,0].

f. Line 16 returns execution to line 9. This avoids iterating over the rest of the List.

In `MoveToOrigin()`, you really don't care about checking the other GameObjects after you've found the one named Phil, so it is better to short circuit the function and return before wasting computing power on doing so. If Phil is the last GameObject in the list, you haven't saved any time, however, if Phil is the first GameObject, you have saved a lot.

Note that when `return` is used in a function with the `void` return type, it does not use parentheses.

Proper Function Names

As you'll recall, variable names should be sufficiently descriptive, start with a lowercase letter, and use camel caps (uppercase letters at each word break). For example:

```
int       numEnemies;
float     radiusOfPlanet;
Color     colorAlert;
string    playerName;
```

Function names are similar; however, function names should all start with a capital letter so that they are easy to differentiate from the variables in your code. Here are some good function names:

```
void ColorAGameObject( GameObject go, Color c ) {...}
void AlignX( GameObject go0, GameObject go1, GameObject go2 ) {...}
void AlignListX( List<GameObject> goList ) {...}
void SetX( GameObject go, float eX ) {...}
```

Why Use Functions?

Functions are a perfect method for encapsulating code and functionality in a reusable form. Generally, any time that you would write the same lines of code more than a couple of times, it's good style to define a function to do so instead. Let's start with a code listing that has some repeated code in it.

The function `AlignX()` in the following code listing takes three GameObjects as parameters, averages their position in the X direction, and sets them all to that average X position:

```
 6 void AlignX( GameObject go0, GameObject go1, GameObject go2 ) {
 7     float avgX = go0.transform.position.x;
 8     avgX += go1.transform.position.x;
```

```
 9    avgX += go2.transform.position.x;
10    avgX = avgX/3.0f;
11
12    Vector3 tempPos;
13    tempPos = go0.transform.position;                    //
a
14    tempPos.x = avgX;                                  // a
15    go0.transform.position = tempPos;                    //
a
16
17    tempPos = go1.transform.position;
18    tempPos.x = avgX;
19    go1.transform.position = tempPos;
20
21    tempPos = go2.transform.position;
22    tempPos.x = avgX;
23    go2.transform.position = tempPos;
24 }
```

a. In lines 13-15, you can see ho we handle the Unity restriction that does not allow you to directly set the position.x of a transform. Instead, you must first copy the current position to another variable (for example, `Vector3 tempPos`), then change the x value, and finally copy the whole Vector3 back onto `transform.position`. This is very tedious to write repeatedly, which led to the `SetX()` function shown in the next code listing. The `SetX()` function in that listing enables you to set the x position of a transform in a single step (e.g., `SetX( this.gameObject, 25.0f )`).

Because of the limitations on directly setting an x, y, or z value of the `transform.position`, there is a lot of repeated code on lines 13 through 23 of the `AlignX()` function. Typing that can be very tedious, and if you needed to change anything later, it would necessitate changing the same thing three times in this `AlignX()` function. This is one of the main reasons for writing functions. In the following code listing, the lines 11-23 from the previous code listing have been replaced by calls to a new function, `SetX()`. The bold lines in the

following code listing have been altered from the previous code listing.

```
 6 void AlignX( GameObject go0, GameObject go1, GameObject go2
) {
 7     float avgX = go0.transform.position.x;
 8     avgX += go1.transform.position.x;
 9     avgX += go2.transform.position.x;
10      avgX = avgX/3.0f;
11
12     SetX ( go0, avgX );
13     SetX ( go1, avgX );
14     SetX ( go2, avgX );
15 }
16
17 void SetX( GameObject go, float eX ) {
18     Vector3 tempPos = go.transform.position;
19     tempPos.x = eX;
20     go.transform.position = tempPos;
21 }
```

In this improved code listing, the removed lines from the previous code have been replaced by the definition of a new function `SetX()` (lines 17–21) and three calls to it (lines 12–14). If anything needed to change about how we were setting the x value, it would only require making a change once to `SetX()` rather than making the change three times in the prior code listing. Though this is a simple example, I hope it serves to demonstrate the power that functions allow us as programmers.

The remainder of this chapter covers some more complex and interesting ways to write functions in C#.

Function Overloading

Function overloading is a fancy term for the capability of functions in C# to act differently based upon the type and number of parameters

that are passed into them. The bold sections of the code below demonstrate function overloading.

```
 6 void Awake() {
 7     print( Add( 1.0f, 2.5f ) );
 8     // ^ Prints: "3.5"
 9     print(  Add( new Vector3(1, 0, 0), new Vector3(0, 1, 0) )  );
10     // ^ Prints "(1.0, 1.0, 0.0)"
11     Color colorA = new Color( 0.5f,  1,     0, 1);
12     Color colorB = new Color( 0.25f, 0.33f, 0, 1);
13     print( Add( colorA, colorB ) );
14     // ^ Prints "RGBA(0.750, 1.000, 0.000, 1.000)"
15 }
16
17 float Add( float f0, float f1 ) {                          // a
18     return( f0 + f1 );
19 }
20
21 Vector3 Add( Vector3 v0, Vector3 v1 ) {                          //
a
22     return( v0 + v1 );
23 }
24
25 Color Add( Color c0, Color c1 ) {                              // a
26     float r, g, b, a;
27     r = Mathf.Min( c0.r + c1.r, 1.0f );                     // b
28     g = Mathf.Min( c0.g + c1.g, 1.0f );                       // b
29     b = Mathf.Min( c0.b + c1.b, 1.0f );                       // b
30     a = Mathf.Min( c0.a + c1.a, 1.0f );                       // b
31     return( new Color( r, g, b, a ) );
32 }
```

a. There are three different `Add()` functions declared and defined in the previous listing, and each is called based on the parameters passed in by various lines of the `Awake()` function. When two floating-point numbers are passed in, the float version of `Add()` is used; when

two Vector3s are passed in, the Vector3 version is used; and when two Colors are passed in, the Color version is used.

b. In the Color version of `Add()`, care is taken to not allow `r`, `g`, `b`, or `a` to exceed 1 because the red, green, blue, and alpha channels of a color are limited to values between 0 and 1. This is done through the use of the `Mathf.Min()` function. `Mathf.Min()` takes any number of arguments as parameters and returns the one with the minimum value. In the previous listing, if the summed reds are equal to 0.75f, then 0.75f will be returned in the red channel; however, if the greens were to sum to any number greater than 1.0f, a green value of 1.0f will be returned instead.

Optional Parameters

There are times when you want a function to have optional parameters that may either be passed in or omitted:

```
 6 void Awake() {
 7     SetX( this.gameObject, 25 );                          // b
 8     print( this.gameObject.transform.position.x ); // Outputs: "25"
 9     SetX( this.gameObject );                              // c
10     print( this.gameObject.transform.position.x ); // Outputs: "0"
11 }
12
13 void SetX( GameObject go, float eX=0.0f ) {                  //
a
14     Vector3 tempPos = go.transform.position;
15     tempPos.x = eX;
16     go.transform.position = tempPos;
17 }
```

a. The float `eX` is defined as an optional parameter with a default value of 0.0f.

b. Because a float can hold any integer value,[1] it is perfectly fine to pass an int into a float. (For example, the integer literal `25` on line 7 is

passed into the float `eX` on line 13.)

[1] To be more precise, a float can hold *most* int values. As was described in Chapter 19, “Variables and Components,” floats get somewhat inaccurate for very big and very small numbers, so a very large int might be rounded to the nearest number that a float can represent. Based on an experiment I ran in Unity, a float seems to be able to represent every whole number up to 16,777,217 after which it will lose accuracy.

c. Because the float `eX` parameter is optional, it is not required, as is shown on line 9.

In this version of the `SetX()` function, float `eX` is an optional parameter. If you give a parameter a default value in the definition of the function, the compiler will interpret that parameter as optional (for example, line 13 in the code listing where the float `eX` is given a default value of 0.0f).

The first time it’s called from `Awake()`, the `eX` parameter is set to 25.0f, which overrides the default of 0.0f. However, the second time it’s called, the `eX` parameter is omitted, leaving `SetX()` to default to a value of 0.0f.

Optional parameters must come after any required parameters in the function definition.

The `params` Keyword

The `params` keyword can be used to make a function accept any number of parameters of the same type. These parameters are converted into an array of that type.

```
6 void Awake() {
7     print( Add( 1 ) );          // Outputs: "1"
8     print( Add( 1, 2 ) );       // Outputs: "3"
9     print( Add( 1, 2, 3 ) );    // Outputs: "6"
```

```
10     print( Add( 1, 2, 3, 4 ) ); // Outputs: "10"
11 }
12
13 int Add( params int[] ints ) {
14     int sum = 0;
15     foreach (int i in ints) {
16         sum += i;
17     }
18     return( sum );
19 }
```

`Add()` can now accept any number of integers and return their sum. As with optional parameters, the `params` list needs to come after any other parameters in your function definition (meaning that you can have other required parameters before the `params` list).

This also allows us to rewrite the `AlignX()` function from before to take any number of possible GameObjects as is demonstrated in the following code listing.

```
 6 void AlignX( params GameObject[] goArray )
{                              // a
 7     float sumX = 0;
 8     foreach (GameObject go in goArray) {                    //
b
 9         sumX += go.transform.position.x;                   // c
10     }
11     float avgX = sumX / goArray.Length;                     //
d
12
13     foreach (GameObject go in goArray) {                    //
e
14         SetX ( go, avgX );
15     }
16 }
17
18 void SetX( GameObject go, float eX ) {
```

```
19      Vector3 tempPos = go.transform.position;
20      tempPos.x = eX;
21      go.transform.position = tempPos;
22  }
```

a. The `params` keyword creates an array of GameObjects from any GameObjects passed in.

b. `foreach` can iterate over every GameObject in `goArray`. The `GameObject go` variable is scoped to the `foreach` loop, so it does not conflict with the `GameObject go` variable in the `foreach` loop on lines 13–15.

c. The X position of the current GameObject is added to `sumX`.

d. The average X position is found by dividing the sum of all X positions by the number of GameObjects.

e. Another `foreach` loop iterates over all the GameObjects in `goArray` and calls `SetX()` with each GameObject as a parameter.

Recursive Functions

Sometimes a function is designed to call itself repeatedly, this is known as a *recursive function*. One simple example of this is calculating the factorial of a number.

In math, 5! (5 factorial) is the multiplication of that number and every other natural number below it. (Natural numbers are the integers greater than 0.)

5! = 5 * 4 * 3 * 2 * 1 = 120

It is a special case in math that 0! is equal to 1.

0! = 1

For our purposes, we will return a 0 any time a negative number is passed into the factorial function:

-5! = 0

We can write a recursive function `Fac()` to calculate the factorial of any integer:

```
 6 void Awake() {
 7     print( Fac(5)  );    // Outputs: "120"                    // a
 8     print( Fac(0)  );    // Outputs: "1"
 9     print( Fac(-5) );    // Outputs: "0"
10 }
11
12 int Fac( int n ) {                                          // b, d
13     if (n < 0) {     // This handles the case if n<0
14         return( 0 );
15     }
16     if (n == 0) {    // This is the "terminal case"              // e
17         return( 1 );
18     }
19     int result = n * Fac( n-1 );                          // c, f
20     return( result );                                     // g
21 }
```

a. When Fac() is called with the integer parameter 5.

b. This enters the first iteration of Fac() with n = 5.

c. On line 19, n (as 5) is then multiplied by the result of calling Fac() with a value of 4. This process of a function calling itself is called recursion.

d. Which enters the second iteration of Fac() with n = 4. The process continues until the sixth iteration of Fac(), where n = 0.

e. Because n is 0, a 1 is returned back up to the fifth iteration.

f. ...which then multiplies 1 * 1

g. ...and passes a 1 back up to the fourth iteration and so on until all the recursions of Fac() have completed, and the first iteration of Fac() returns a value of 120 to line 7.

The chain of all these recursive `Fac()` calls works something like this:

Fac(5) // 1st Iteration
5 * **Fac(4)** // 2nd Iteration
5 * 4 * **Fac(3)** // 3rd Iteration
5 * 4 * 3 * **Fac(2)** // 4th Iteration
5 * 4 * 3 * 2 * **Fac(1)** // 5th Iteration
5 * 4 * 3 * 2 * 1 * **Fac(0)** // 6th Iteration
5 * 4 * 3 * 2 * **1 * 1** // 5th Iteration
5 * 4 * 3 * **2 * 1** // 4th Iteration
5 * 4 * **3 * 2** // 3rd Iteration
5 * **4 * 6** // 2nd Iteration
5 * 24 // 1st Iteration
120 // Final Return Value

The best way to really understand what's happening in this recursive function is to explore it using the debugger, a feature in MonoDevelop that enables you to watch each step of the execution of your programs and see how different variables are affected by your code. The process of debugging is the topic of the next chapter.

Summary

In this chapter, you have seen the power of functions and many different ways that you can use them. Functions are a cornerstone of any modern programming language, and the more programming you do, the more you will see how powerful and necessary they are.

The upcoming Chapter 25, "Debugging," shows you how to use the debugging tools in Unity. These tools are meant to help you find problems with your code, but they are also very useful for

understanding how your code works. After you have learned about debugging from the next chapter, I recommend returning to this chapter and examining the `Fac()` function in more detail. And, of course, feel free to explore any of the functions in this chapter or others using the debugger to better understand them.

Chapter 30. Prototype 3: Space shmup

The SHMUP (or shoot ‘em up) game genre includes such classic games as Galaga and Galaxian from the 1980s and the modern masterpiece Ikaruga.

In this chapter, you create a SHMUP using several programming techniques that will serve you well throughout your programming and prototyping careers, including class inheritance, enums, static fields and methods, and the singleton pattern. Though you’ve seen many of these techniques before, they will be used more extensively in this prototype.

Getting Started: Prototype 3

In this project, you make a prototype for a classic space-based SHMUP. This chapter will get you to the same basic prototype level as the previous two chapters, and the next chapter will show you how to implement several additional features. Figure 30.1 shows an image of what the finished prototype will look like after both chapters. These images show the player ship at the bottom surrounded by a green shield as well as several enemy types and upgrades (the power-up cubes marked B, O, and S).

Figure 30.1. Two views of the Space SHMUP game prototype. The player is using the blaster weapon in the left image and the spread weapon in the right.



Importing a Unity Asset Package

One new thing in the setup for this prototype is that you will be asked to download and import a custom Unity asset package. The creation of complex art and imagery for games is beyond the scope of this book, but I've created a package of some simple assets for you that will allow you to create all the visual effects required for this game. Of course, as mentioned several times throughout this book, when you're making a prototype, how it plays and feels are much more important than how it looks, but it's still important to have an understanding of how to work with art assets.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on this procedure, see Appendix A, "Standard Project Setup Procedure."

• **Project name:** Space SHMUP Prototype.

• **Scene name:** _Scene_0.

• **Project folders:** __Scripts (2 underscores before "Scripts"), _Materials, _Prefabs.

• **Download and import package:** Find Chapter 30 at http://book.prototools.net.

• **C# script names**: (none yet).

• **Rename:** Change Main Camera to _MainCamera.

To download and install the package mentioned in the sidebar "Set Up the Project for This Chapter," first follow the URL listed (http://book.prototools.net) and search for this chapter. Download C30_Space_SHMUP_Starter.unitypackage to your machine, which will usually place it in your *Downloads* folder. Open your project in Unity and select *Assets* > *Import Package* > *Custom Package* from

the menu bar. Navigate to and select *C30_Space_SHMUP_Starter.unitypackage* from your Downloads folder. This will open the import dialog box shown in Figure 30.2.

Figure 30.2. The .unitypackage import dialog box



Select all the files as shown in Figure 30.2 (by clicking the *All* button), and click *Import*. This will place four new *textures* and one new *shader* into the _Materials folder. Textures are usually just image files. The creation of textures is beyond the scope of this book, but many books and online tutorials cover texture creation. *Adobe Photoshop* is probably the most commonly used image editing tool, but it is very expensive. A common open source alternative is *Gimp* (http://www.gimp.org).

The creation of shaders is also far beyond the scope of this book. Shaders are programs that tell your computer how to render a texture on a GameObject. They can make a scene look realistic, cartoony, or however else you like, and they are an important part of the graphics of any modern game. Unity uses its own unique shader language called ShaderLab. If you want to learn more about it, a good place to start is the Unity Shader Reference documentation (http://docs.unity3d.com/Documentation/Components/SL-Reference.html).

The included shader is a simple one that bypasses most of the things a shader can do to simply render a colored, non-lit shape on the screen. For on-screen elements that you want to be a specific bright color, this custom UnlitAlpha.shader is perfect. UnlitAlpha also allows for alpha blending and transparency, which will be very useful for the power-up cubes in this game.

Setting the Scene

Follow these steps to set the scene:

1. Select *Directional Light* in the Hierarchy and set its transform to P:[0,20,0] R:[50,330,0] S:[1,1,1].

2. Make sure that you renamed *Main Camera* (as you were instructed in the project setup sidebar). Select *_MainCamera* and set its transform to P:[0,0,-10] R:[0,0,0] S:[1,1,1].

3. In the Camera component, set the following and then save your scene.

• *Clear Flags* to Solid Color

• *Background* to black (with 255 alpha)

• *Projection* to Orthographic

• *Size* to 40 (after setting *Projection*)

• *Near Clipping Plane* to 0.3

• *Far Clipping Plane* to 100

4. Because this game will be a vertical, top-down shooter, we need to set an aspect ratio for the Game pane that is in portrait orientation. In the Game pane, click the pop-up menu list of aspect ratios that currently shows *Free Aspect* (See Figure 30.0). At the bottom of the list, you will see a + symbol. Click this to add a new aspect ratio preset. Set the values to those shown in Figure 30.3, and then click *Add*. Set the Game pane to this new *Portrait (3:4)* aspect ratio.

Figure 30.3. Adding a new aspect ratio preset to the Game pane



Making the Hero Ship

In this chapter, we interleave the construction of artwork and code rather than building all the art first. To make the player's spaceship,

complete these steps:

1. Create an empty GameObject and name it *_Hero* (*GameObject* > *Create Empty*). Set its transform to P:[0,0,0] R:[0,0,0] S:[1,1,1].

2. Create a cube (*GameObject* > *3D Object* > *Cube*) and drag it onto _Hero, making it a child thereof. Name the cube *Wing* and set its transform to P:[0,-1,0] R:[0,0,45] S:[3,3,0.5].

3. Create an empty GameObject, name it *Cockpit*, and make it a child of _Hero.

4. Create a cube and make it a child of Cockpit (you can do this by right-clicking on Cockpit and choosing *3D Object* > *Cube*). Set the Cube's transform to P:[0,0,0] R:[315,0,45] S:[1,1,1].

5. Make Cockpit's transform P:[0,0,0] R:[0,0,180] S:[1,3,1]. This uses the same trick as was used in Chapter 26, "Object-Oriented Thinking," to make a quick, angular ship.

6. Select _Hero in the Hierarchy and click the *Add Component* button in the Inspector. Choose *New Script* from the pop-up menu. Name the script Hero, double-check that *Language* is C Sharp, and click *Create and Add*. This is another way to make a new script and attach it to a GameObject. In the Project pane, move the Hero script into the __Scripts folder.

7. Add a Rigidbody component to _Hero by selecting _Hero in the Hierarchy and then choosing *Add Component* > *Physics* > *Rigidbody* from the *Add Component* button in the Inspector. Set the following on the Rigidbody component of _Hero.

• *Use Gravity* to false (unchecked)

• *isKinematic* to true (checked)

• *Constraints*: freeze Z position and X, Y, and Z rotation (by checking them)

You'll add more to _Hero later, but this will suffice for now.

8. Save your scene! Remember that you should be saving your scene every time you make a change to it. I'll quiz you later.

Hero.Update()

In the code listing that follows, the `Update()` method first reads the horizontal and vertical axes from the InputManager (see the "Input.GetAxis() and The InputManager" sidebar), placing values between –1 and 1 into the floats `xAxis` and `yAxis`. The second chunk of `Update()` code moves the ship in a time-based way, taking into account the `speed` setting.

The last line (marked `// c`) rotates the ship based on the input. Although we earlier froze the rotation in _Hero's Rigidbody component, it is still possible for us to manually set the rotation on a Rigidbody with isKinematic set to true. (As discussed in an earlier chapter, isKinematic = true means that the Rigidbody will be tracked by the physics system but that it will not move automatically due to Rigidbody.velocity.) This rotation will make the movement of the ship feel more dynamic and expressive, or "juicy."[1]

[1] *Juiciness,* as a term that relates to gameplay, was coined in 2005 by Kyle Gabler and the other members of the Experimental Gameplay Project at Carnegie Mellon University's Entertainment Technology Center. To them, a juicy element had "constant and bountiful user feedback." You can read about it more in their Gamasutra article "How to Prototype a Game in Under 7 Days." http://www.gamasutra.com/view/feature/130848/how_to_prototype_a_game_in_under_7_.php

Open the Hero C# script in MonoDevelop and enter the following code:

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

```
public class Hero : MonoBehaviour {
   static public Hero      S; // Singleton                              // a

   [Header("Set in the Unity Inspector")]
   // These fields control the movement of the ship
   public float          speed = 30;
   public float          rollMult = -45;
   public float          pitchMult = 30;

   [Header("These fields are set dynamically")]
   public float          shieldLevel = 1;

   void Awake() {
      S = this;  // Set the Singleton                                  // a
   }

   void Update () {
     // Pull in information from the Input class
      float xAxis = Input.GetAxis("Horizontal");                        //
b
      float yAxis = Input.GetAxis("Vertical");                         // b

      // Change transform.position basec on the axes
      Vector3 pos = transform.position;
      pos.x += xAxis * speed * Time.deltaTime;
      pos.y += yAxis * speed * Time.deltaTime;
      transform.position = pos;

      // Rotate the ship to make it feel more
dynamic                    // c
      transform.rotation =
Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
   }
}
```

a. The singleton for the Hero class (see “Software Design Patterns” in Appendix B).

b. These two lines use Unity’s `Input` class to pull information from the Unity InputManager. See the sidebar for more information.

c. The `transform.rotation...` line below this comment is used to give the ship a bit of rotation based on the speed at which it is moving, which can make the ship feel more reactive and juicy.

`Input.GetAxis()` and The InputManager

Much of the code in the `Hero.Update()` code listing should look familiar to you, though this is the first time in the book that we’ve used the `Input.GetAxis()` method. Various axes are configured in Unity’s InputManager, and `Input.GetAxis()` allows them to be read. To view the default Input axes, choose *Edit > Project Settings > Input* from the menu bar.

One thing to note about the settings in Figure 30.4 is that there are several that are listed twice (for example, Horizontal, Vertical, Jump). As you can see in the expanded view of the horizontal axes in the figure, this allows the horizontal axis to be controlled by either presses on the keyboard (shown in the left image of Figure 30.4) or a joystick axis (shown in the right image). This is one of the great strengths of the input axes; several different types of input can control a single axis. As a result, your games only need one line to read the value of an axis rather than a line to handle joystick input, a line for each arrow key, and a line each for the a and d keys.

Figure 30.4. Unity’s InputManager showing default settings (shown in two halves)



Every call to `Input.GetAxis()` will return a float between -1 and 1 in value (with a default of 0). Each axis in the InputManager also

includes values for *Sensitivity* and *Gravity*, though these are only used for *Key or Mouse Button* input (see the left image of Figure 30.4). Sensitivity and gravity cause the axis value to interpolate smoothly when a key is pressed or released. (That is, instead of immediately jumping to the final value, the axis value will blend from the original value to the final value over time.) In the horizontal axis shown, a sensitivity of 3 means that when the right-arrow key is pressed, it will take 1/3 of a second for the value to interpolate from 0 to 1. A gravity of 3 means that when the right-arrow key is released, it will take 1/3 of a second for the axis value to interpolate back to 0. The higher the sensitivity or gravity, the faster the interpolation will take place.

As with almost anything in Unity, you can find out a lot more about the InputManager by clicking the Help button (that looks like a blue book with a question mark and is between the name InputManager and the gear at the top of the Inspector).

Try playing the game and see how the ship feels to you. The settings for `speed`, `rollMult`, and `pitchMult` work for me, but this is your game, and you should have settings that feel right to you. Make changes as necessary in the Unity Inspector for _Hero.

Part of what makes this feel nice is the apparent inertia that the ship carries. When you release the movement key, it takes the ship a little while to slow down. Similarly, upon pressing a movement key, it takes the ship a little while to get up to speed. This apparent movement inertia is caused by the *sensitivity* and *gravity* axis settings that are described in the sidebar. Changing these settings in the InputManager will affect the movement and maneuverability of _Hero.

The Hero Shield

The shield for _Hero will be a combination of a transparent, textured quad (to provide the visuals) and a Sphere Collider (for collision

handling).

1. Create a new quad (*GameObject* > *3D Object* > *Quad*). Rename the quad *Shield* and make it a child of _Hero. Set the transform of Shield to P:[0,0,0] R:[0,0,0], S:[8,8,8].

2. Select Shield in the Hierarchy and delete the existing Mesh Collider component by clicking the tiny gear to the right of the Mesh Collider name in the Inspector and choosing *Remove Component* from the pop-up menu. Then add a Sphere Collider component (*Component* > *Physics* > *Sphere Collider*).

3. Create a new material (*Assets* > *Create* > *Material*), name it *Mat_Shield*, and place it in the _Materials folder in the Project pane. Drag Mat_Shield onto the Shield under _Hero in the Hierarchy to assign it to the Shield quad.

4. Select Shield in the Hierarchy, and you will now see Mat_Shield in the Inspector for Shield. Set the Shader of Mat_Shield to *ProtoTools* > *UnlitAlpha*. Below the shader selection pop-up for Mat_Shield, there should be an area that allows you to choose the main color for the material as well as the texture. (If you don't see this, click once on the name *Mat_Shield* in the Inspector, and it should appear.)

5. Click *Select* in the bottom-right corner of the texture square and select the texture named *Shields*. Click the color swatch next to *Main Color* and choose a bright green (RGBA:[0,255,0,255]). Then set:

• *Tiling.x* to 0.2

• *Offset.x* to 0.4

• Tiling.y should remain 1.0

• Offset.y should remain 0

The Shield texture was designed to be split into five sections horizontally. The x Tiling of 0.2 causes Mat_Shield to only use 1/5 of

the total Shield texture in the x direction, and the x Offset chooses which fifth. Try x Offsets of 0, 0.2, 0.4, 0.6, and 0.8 to see the different levels of shield strength.

6. Create a new C# script named *Shield* (*Asset* > *Create* > *C# Script*). Drop it into the __Scripts folder in the Project pane and then drag it onto Shield in the Hierarchy to assign it as a component of the Shield GameObject.

7. Open the Shield script in MonoDevelop and enter the following code:

```
using UnityEngine;
using System.Collections;

public class Shield : MonoBehaviour {
   [Header("Set in the Unity Inspector")]
   public float    rotationsPerSecond = 0.1f;

   [Header("These fields are set dynamically")]
   public int      levelShown = 0;

   // This non-public variable will not appear in the Inspector
   Material       mat;                                    // a

   void Start() {
      mat = GetComponent<Renderer>().material;                    //
b
   }

   void Update () {
      // Read the current shield level from the Hero Singleton
      int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel );       //
c
      // If this is different from levelShown...
      if (levelShown != currLevel) {
         levelShown = currLevel;
```

```
            // Adjust the texture offset to show different shield level
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0
); // d
        }
        // Rotate the shield a bit every frame in a time-based way
        float rZ = -(rotationsPerSecond*Time.time*360) %
360f;        // e
        transform.rotation = Quaternion.Euler( 0, 0, rZ );
    }

}
```

a. The Material field `mat` is *not* declared public, so it will not be visible in the Inspector, and it will not be able to be accessed outside of this Shield class.

b. In `Start()`, `mat` is defined as the material of the Renderer component on the this GameObject (Shield in the Hierarchy). This allows us to quickly set the texture offset in the line marked `// d`.

c. `currLevel` is set to the floor of the current `Hero.S.shieldLevel` float. By flooring the `shieldLevel`, we make sure that the shield jumps to the new x Offset rather than showing an Offset between two shield icons.

d. This line adjusts the x Offset of Mat_Shield to show the proper shield level.

e. This line and the next cause the Shield GameObject to rotate slowly about the z axis.

Keeping _Hero On Screen[2]

[2] The first edition of this book had a much more complex system for keeping GameObjects on screen that was more than was needed for this chapter and somewhat confusing. I've replaced it with this

version in the second edition to both streamline the chapter and to reinforce the concept of components.

The motion of your _Hero ship should feel pretty good now, and the rotating shield looks pretty nice, but at this point, you can easily drive the ship off the screen. In order to resolve this, we're going to make a reusable component script. You can read more about the component software design pattern in Chapter XX, "Object-Oriented Thinking" and under "Software Design Patterns" in Appendix B, "Useful Concepts." In brief, a component is a small piece of code that is meant to work alongside others to add functionality to a GameObject without conflicting with other code on the object. Unity's components that you've worked with in the Inspector (e.g., Renderer, Transform, etc.) all follow this pattern. Now, we'll do the same with a small script to keep _Hero on screen. Note that this script will only work with orthographic cameras.

1. Select _Hero in the Hierarchy and using the *Add Component* button in the Inspector, choose *Add Component > New Script*. Name the script BoundsCheck and click *Create and Add*. Then drag the BoundsCheck script in the Project pane into the __Scripts folder.

2. Open the BoundsCheck script and add the following code:

```
using UnityEngine;
using System.Collections;

// To type the next 4 lines, start by typing /// and then Tab.
/// <summary>
/// Keeps a GameObject on screen.
/// Note that this ONLY works for an orthographic Main Camera.
/// </summary>
public class BoundsCheck : MonoBehaviour {                          // a
    [Header("Set in the Unity Inspector")]
    public float    radius = 1f;

    [Header("These fields are set dynamically")]
    public float    camWidth;
```

```
    public float    camHeight;

    void Start() {
        camHeight = Camera.main.orthographicSize;                    //
b
        camWidth = camHeight *
Camera.main.aspect;                    // c
    }

    void LateUpdate () {                                             // d
        Vector3 pos = transform.position;

        if (pos.x > camWidth - radius) {
            pos.x  = camWidth - radius;
        }
        if (pos.x < -camWidth + radius) {
            pos.x  = -camWidth + radius;
        }

        if (pos.y > camHeight - radius) {
            pos.y  = camHeight - radius;
        }
        if (pos.y < -camHeight + radius) {
            pos.y  = -camHeight + radius;
        }

        transform.position = pos;
    }
}
```

a. Because this is intended to be a reusable piece of code, it's useful for us to add some internal documentation to it. The lines above the class declaration that are all begin with /// are part of MonoDevelop's internal documentation system. Once you've typed this, it interprets the text between the `<summary>` tags as a summary of what the class is used for. After typing it, hover your mouse over the name of the

class on the line marked `// a`, and you should see a pop-up with this class summary.

b. `Camera.main` gives us access to the first camera with the tag *MainCamera* in our scene. Then, if the camera is orthographic, `.orthographicSize` gives us the Size number from the Camera Inspector (which is 40 in this case). This makes `camHeight` the distance from the origin of the world (position 0,0,0) to the top or bottom edge of the screen in world coordinates.

c. `Camera.main.aspect` is the aspect ratio of the camera in width/height as defined by the aspect ratio of the Game pane (currently set to *Portrait (3:4)*). By multiplying `camHeight` by `.aspect`, we can get the distance from the origin to the left or right edge of the screen.

d. `LateUpdate()` is called every frame after `Update()` has been called on all GameObjects. If this code was in the `Update()` function, it might happen before the `Update()` call on the Hero script, or it might happen after. By putting this code in `LateUpdate()`, we avoid a *race condition* between the two `Update()` functions and ensure that `Hero.Update()` moves the _Hero GameObject to the new position each frame before this function keeps it on screen.

A *race condition* is an instance where the order in which two pieces of code execute (i.e., A before B or B before A) matters, but we don't have control over that order. For example, in this code, if `BoundsCheck.LateUpdate()` were to execute before `Hero.Update()`, the _Hero GameObject would potentially be moved out of bounds (because it would first limit the bounds and *then* move the ship). Using `LateUpdate()` in `BoundsCheck` enforces the execution order of the two scripts.

3. Press Play and try flying your ship around. Based on the default setting for `radius`, you should see that the ship stops while it is still 1m on screen. If you set `BoundsCheck.radius` to 4 in the _Hero Inspector, the ship will be kept entirely on screen. If you set radius to -4, the ship will be allowed to exit the edge of the screen but will be

locked there, ready to come right back on. Stop playback and set `radius` to 4.

Adding Some Enemies

The enemies for a game like this were covered a bit in Chapter 25, “Classes.” There you learned about setting up a superclass for all enemies that can be extended by subclasses. For this game, we extend that further, but first, let’s create the artwork.

Enemy Artwork

Because the hero ship has such an angular aesthetic, all the enemies will be constructed of spheres as shown in Figure 30.5.

Figure 30.5. Each of the five enemy ship types



Enemy_0-4

To create the artwork for Enemy_0 do the following:

1. Create an empty GameObject and name it *Enemy_0*., and set its transform to P:[-20,0,0] R:[0,0,0] S:[1,1,1]. This position is to make sure it doesn’t overlap with _Hero as we build it.

2. Create a sphere named *Cockpit*, make it a child of Enemy_0, and set its transform to P:[0,0,0] R:[0,0,0] S:[2,2,1].

3. Create a second sphere named *Wing*, make it a child of Enemy_0, and set its transform to P:[0,0,0] R:[0,0,0] S:[5,5,0.5].

Another way of writing the preceding three steps would be:



4. Follow this formatting to make the remaining four enemies. When finished, they should look like the enemies in Figure 30.5.

Enemy_1



Enemy_2



Enemy_3



Enemy_4



5. You must add a Rigidbody component to each of the enemy GameObjects (that is, Enemy_0, Enemy_1, Enemy_2, Enemy_3, and Enemy_4). To add a Rigidbody, complete these steps:

a. Select each enemy in the Hierarchy and choose *Component* > *Physics* > *Rigidbody* from the menu bar to add the Rigidbody component.

b. In the Rigidbody component for the enemy, set *Use Gravity* to false

c. Set *isKinematic* to true.

d. Open the disclosure triangle for *Constraints* and freeze z position and x, y, and z rotation.

Be sure to do this for all five enemies. If a moving GameObject doesn't have a Rigidbody component, the GameObject's collider location will not move with the GameObject, but if a moving GameObject does have a Rigidbody, the colliders of both it and all of

its children are updated every frame (which is why you don't need to add a Rigidbody component to any of the children of the enemy GameObjects).

6. Drag each of these enemies to the _Prefabs folder of the Project pane to create a prefab for each.

7. Delete all of the enemy instances from the Hierarchy except for Enemy_0.

The Enemy C# Script

To create the Enemy script, follow these steps:

1. Create a new C# script named *Enemy* and place it into the __Scripts folder.

2. Drag the Enemy script onto Enemy_0 in the Project pane (not in the Hierarchy). When you click on Enemy_0 in either the Project or Hierarchy panes, you should see an Enemy (Script) component attached.

3. Open the Enemy script in MonoDevelop and enter the following code:

```
using UnityEngine;                // Required for Unity
using System.Collections;         // Required for Arrays & other Collections

public class Enemy : MonoBehaviour {
   [Header("Set in the Unity Inspector")]
   public float      speed = 10f;      // The speed in m/s
   public float      fireRate = 0.3f;  // Seconds/shot (Unused)
   public float      health = 10;
   public int        score = 100;      // Points earned for destroying this

   // This is a Property: A method that acts like a field
```

```
    public Vector3 pos {                                    // a
        get {
            return( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }

    void Update() {
        Move();
    }

    public virtual void Move() {                            // b
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }

}
```

a. As was discussed in Chapter 25, “Classes,” a *property* is a function masquerading as a field. This means that you can get and set the value of `pos` as if it were a class variable of Enemy.

b. The `Move()` method gets the current position of this Enemy_0, moves it in the downward Y direction, and assigns it back to `pos` (setting the position of the GameObject).

4. Press Play, and the instance of Enemy_0 in the scene should move toward the bottom of the screen. However, with the current code, this instance will continue off screen and exist until you stop your game. We need to have the enemy destroy itself once it has moved entirely off screen. This is a great place to reuse our BoundsCheck component.

5. To attach the BoundsCheck script to the Enemy_0 prefab, select the Enemy_0 prefab in the Hierachy (not the Project pane this time).

In the Inspector, click *Add Component* and choose *Add Component* > *Scripts* > *BoundsCheck*. This attaches the script to the instance of Enemy_0 in the Hierarchy, but it has not yet attached it to the Enemy_0 prefab in the Project pane.

6. To apply this change made to the Enemy_0 instance back to its prefab, click *Apply* at the top of the Inspector for the Enemy_0 instance in the Hierarchy. Now check the Enemy_0 prefab in the Project pane to see that the script is attached.

7. Select the Enemy_0 instance in the Hierarchy and set the radius value of in the BoundsCheck Inspector to -2.5. Note that this value is bolded because it is different from the value on the prefab. Click Apply at the top of the Inspector again, and the radius value will no longer be bolded, showing you that it is the same value as the one on the prefab.

8. Press Play, and you'll see that the Enemy_0 instance stops right after it has gone off screen. However, instead of forcing Enemy_0 to remain on screen, we really want to be able to check whether it has gone off screen and then destroy it when it has.

9. To do so, make the following bolded modifications to the BoundsCheck script.

```
/// <summary>
/// Checks whether a GameObject is on screen and can force it to stay on screen.
/// Note that this ONLY works for an orthographic Main Camera.
/// </summary>
public class BoundsCheck : MonoBehaviour {
    [Header("Set in the Unity Inspector")]
    public float    radius = 1f;
    public bool     keepOnScreen = true;                    // a

    [Header("These fields are set dynamically")]
    public bool     isOnScreen = true;                      // b
```

```
    public float    camWidth;
    public float    camHeight;

    void Start() { ... }       // Remember, ellipses mean to not alter this
method.

    void LateUpdate () {
        Vector3 pos = transform.position;                       // c
        isOnScreen = true;                                      // d

        if ( pos.x > camWidth - radius ) {
            pos.x  = camWidth - radius;
            isOnScreen = false;                                 // e
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x  = -camWidth + radius;
            isOnScreen = false;                                 // e
        }

        if ( pos.y > camHeight - radius ) {
            pos.y  = camHeight - radius;
            isOnScreen = false;                                 // e
        }
        if ( pos.y < -camHeight + radius ) {
            pos.y  = -camHeight + radius;
            isOnScreen = false;                                 // e
        }

        if ( keepOnScreen && !isOnScreen ) {                    // f
            transform.position = pos;                           // g
            isOnScreen = true;
        }
    }
}
```

a. `keepOnScreen` allows us to choose whether BoundsCheck forces a GameObject to stay on screen (`true`) or allows it to exit the screen

and notifies us that it has done so (`false`).

b. `isOnScreen` will turn false if the GameObject exits the screen. More accurately it will turn false if the GameObject goes past the edge of the screen minus `radius`. This is why `radius` is set to -2.5 for Enemy_0, so that it is completely off screen before `isOnScreen` is set to `false`.

c. Remember that ellipses in code mean you should *not* modify the `Start()` method.

d. `isOnScreen` is set to true until proven false. This allows the value of `isOnScreen` to return to true if the GameObject was off screen in the last frame but has come back on in this frame.

e. If any of these four if statements are true, then the GameObject is outside of the area it is supposed to be in. `isOnScreen` is set to `false`, and `pos` is adjusted to a position that would bring the GameObject back "on screen."

f. If `keepOnScreen` is `true`, then we are trying to force the GameObject to stay on screen. If `keepOnScreen` is `true` and `isOnScreen` is `false`, then the GameObject has gone out of bounds and needs to be brought back in. In this case, `transform.position` is set to the updated `pos` that is on screen, and `isOnScreen` is set to `true` because this position assignment has now moved the GameObject back on screen.

If `keepOnScreen` is `false`, then `pos` is *not* assigned back to `transform.position`, the GameObject is allowed to go off screen, and `isOnScreen` is allowed to remain `false`. The other possibility is that the GameObject was on screen the whole time, in which case, `isOnScreen` would still be `true` from when it was set on line `// d`.

g. Note that this line is now indented and inside the if statement on line `// f`.

Happily, all of these modifications to the code do not negatively impact the way it was used for _Hero, and everything still works fine. This has created a reusable component that can be applied to both _Hero and the Enemy GameObjects.

Deleting the Enemy When It Goes Off Screen

Now that BoundsCheck can tell us when Enemy_0 goes off screen, we need to set it to properly do so.

1. First, set `keepOnScreen` to `false` in the BoundsCheck (Script) component of the Enemy_0 prefab in the Project.

To ensure that this propagates to the Enemy_0 instance in the Hierarchy, select the instance in the Hierarchy and click the gear to the right of the BoundsCheck (Script) component heading in the Inspector. From the gear pop-up menu, choose Revert to Prefab to set the values of the instance in the Hierarchy to those on the prefab.

When you've done this, the BoundsCheck (Script) component on both the Enemy_0 prefab in the Project pane and the Enemy_0 instance in the Hierarchy should look like Figure 30.6.

Figure 30.6. The BoundsCheck (Script) component settings for both the prefab and instance of Enemy_0



2. Add the following bold code to the Enemy script:

```
public class Enemy : MonoBehaviour {
    ...
    public int        score = 100; // Points earned for destroying this

    private BoundsCheck bndCheck;                                    // a

    void Awake() {                                                   // b
        bndCheck = GetComponent<BoundsCheck>();
```

```
    }

    ...

    void Update() {
        Move();

        if ( bndCheck != null && !bndCheck.isOnScreen ) {           //
c
            // Check to make sure it's gone off the bottom of the
screen
            if ( pos.y < bndCheck.camHeight - bndCheck.radius )
{      // d
                // We're off the bottom, so destroy this GameObject
                Destroy( gameObject );
            }
        }
    }

    ...
}
```

a. This private variable will allow this Enemy script to store a reference to the BoundsCheck script on the same GameObject.

b. In this `Awake()` method, we search for the BoundsCheck script component attached to this same GameObject. If there is not one, `bndCheck` will be set to `null`. Things like this are often placed in the `Awake()` method so that they are ready immediately once this GameObject is instantiated.

c. First checks to make sure `bndCheck` is not `null`. If we attached the Enemy script to a GameObject without attaching a BoundsCheck script as well, this could be the case. Only if `bndCheck != null` does the script check to see if the GameObject is not on screen (according to BoundsCheck).

d. If `isOnScreen` is `false`, this line checks to see whether it is off screen because it has a `pos.y` that is too negative (i.e., it has gone off the bottom of the screen). If this is the case, the GameObject is destroyed.

This works and does what we want, but it seems a bit messy to be doing the same comparison of `pos.y` versus the `camHeight` and `radius` both here and in BoundsCheck. It is generally considered good programming style to let each C# class (or component) handle the job it is meant to do and not have crossover like this. As such, let's alter BoundsCheck to be able to tell us in which direction the GameObject has gone off screen.

3. Modify the BoundsCheck script by adding the bolded code that follows:

```
public class BoundsCheck : MonoBehaviour {
    ...
    public float    camHeight;
    [HideInInspector]
    public bool     offRight, offLeft, offUp, offDown;              // a

    void Start() { ... }

    void LateUpdate () {
        Vector3 pos = transform.position;
        isOnScreen = true;
        offRight = offLeft = offUp = offDown = false;              // b

        if ( pos.x > camWidth - radius ) {
            pos.x  = camWidth - radius;
            offRight = true;                                       // c
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x  = -camWidth + radius;
            offLeft = true;                                        // c
        }
```

```
        if ( pos.y > camHeight - radius ) {
            pos.y  = camHeight - radius;
            offUp = true;                                   // c
        }
        if ( pos.y < -camHeight + radius ) {
            pos.y  = -camHeight + radius;
            offDown = true;                                  // c
        }

        isOnScreen = !(offRight || offLeft || offUp || offDown);       //
d
        if ( keepOnScreen && !isOnScreen ) {
            transform.position = pos;
            isOnScreen = true;
        }
    }
}
```

a. Here we declare four private variables, one for each direction in which the GameObject could go off screen. As bools, they all will default to `false`. The `[HideInInspector]` line preceding this causes these four public fields to not appear in the Inspector, though they are still public variables and can still be read (or set) by other classes. It applies to all four *`off__` bools* (i.e., `offRight`, `offLeft`, etc.) because they are all declared on the line beneath it. If the `off__` bools were declared on four separate lines, a `[HideInInspector]` line would need to precede each one to achieve the same effect.

b. At the beginning of each `LateUpdate()` we set all four `off__` bools to `false`. In this line, `offDown` is first set to `false`, then `offUp` is set to the value of `offDown` (i.e., `false`), and so on until all `off__` bools hold the value `false`. This takes the place of the old line that set `isOnScreen` to `true`.

c. Each instance of `isOnScreen = false;` has now been replaced with an `off__ = true;` so that we know in which direction the

GameObject has exited the screen. It is possible for two of these `off__` bools to both be true, for example when the GameObject has exited the bottom-right corner of the screen.

d. Here, `isOnScreen` is set based on the values of all the `off__` bools. First, inside the parentheses, we take the logical OR (`||`) of all the `off__` bools. If one or more of them are `true`, this will evaluate to `true`. Then, we take the NOT (`!`) of that and assign it to `isOnScreen`. So, if one or more of the `off__` bools are `true`, `isOnScreen` will be `false`.

4. Now, make the following bolded changes to the Enemy script to take advantage of the improvements to BoundsCheck.

```
public class Enemy : MonoBehaviour {
    ...

    void Update() {
        Move();

        if ( bndCheck != null && bndCheck.offDown ) {                    // a
            // We're off the bottom, so destroy this
GameObject        // b
            Destroy( gameObject );                                       // b
        }
    }

    ...
}
```

a. Now, we just need to check against bndCheck.offDown to determine whether the Enemy instance has gone off the bottom of the screen.

b. These two lines have lost one tab of indentation because the re is now only one if clause instead of two.

This is a much simpler implementation from the viewpoint of the Enemy class, and it makes good use of the BoundsCheck component, allowing it to do its job without needlessly duplicating its functionality in the Enemy class.

Now, when you play the scene, you should see that the Enemy_0 ship moves down the screen and is destroyed as soon as it exits the bottom of the screen.

Spawning Enemies at Random

With all of this in place, it's now possible to instantiate a number of Enemy_0s randomly.

1. Attach a BoundsCheck script to _MainCamera and set its keepOnScreen field to false.

2. Create a new C# script called *Main* inside the __Scripts folder. Attach it to _MainCamera, and then enter the following code:

```
using UnityEngine;                // Required for Unity
using System.Collections;         // Required for Arrays & other Collections
using System.Collections.Generic;   // Required to use Lists or Dictionaries
using UnityEngine.SceneManagement;  // Allows the loading & reloading of
                                    // scenes

public class Main : MonoBehaviour {
   static public Main S;                        // A singleton for Main

   [Header("Set in the Unity Inspector")]
   public GameObject[]     prefabEnemies;       // Array of Enemy prefabs
   public float            enemySpawnPerSecond = 0.5f; // # Enemies/second
```

```
    public float          enemyDefaultPadding = 1.5f; // Padding for
                                                      // position

    private BoundsCheck     bndCheck;


    void Awake() {
        S = this;
        // Set bndCheck to reference the BoundsCheck component
on this
        // GameObject
        bndCheck = GetComponent<BoundsCheck>();

        // Invoke SpawnEnemy() once (in 2 seconds, based on
default values)
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond
);              // a
    }

    public void SpawnEnemy() {
        // Pick a random Enemy prefab to instantiate
        int ndx = Random.Range(0,
prefabEnemies.Length);              // b
        GameObject go = Instantiate<GameObject>(
prefabEnemies[ ndx ] );// c

        // Position the Enemy above the screen with a random x
position
        float enemyPadding =
enemyDefaultPadding;                    // d
        if (go.GetComponent<BoundsCheck>() != null)
{               // e
            enemyPadding = Mathf.Abs(
go.GetComponent<BoundsCheck>().radius );
        }

        // Set the initial position for the spawned Enemy          // f
```

```
        Vector3 pos = Vector3.zero;
        float xMin = -bndCheck.camWidth + enemyPadding;
        float xMax =  bndCheck.camWidth - enemyPadding;
        pos.x = Random.Range( xMin, xMax );
        pos.y = bndCheck.camHeight + enemyPadding;
        go.transform.position = pos;

        // Invoke SpawnEnemy() again
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond
);              // g
    }
}
```

a. This `Invoke()` function will call the `SpawnEnemy()` method in 1/0.5 seconds (i.e., 2 seconds) based on the default values.

b. Based on the length of the array `prefabEnemies`, this will choose a random number between 0 and one less than `prefabEnemies.Length`, so if there are 4 prefabs in the `prefabEnemies` array, it will return 0, 1, 2, or 3. The `int` version of `Random.Range()` will never return a number as high as the max (i.e., second) integer passed in. The float version is able to return the max number.

c. The random `ndx` generated is used to select a GameObject prefab from `prefabEnemies`.

d. The enemyPadding is initially set to the enemyDefaultPadding set in the Inspector.

e. However, if the selected enemy prefab has a BoundsCheck component, we instead read the radius from that. The absolute value of the radius is taken because sometimes the radius is set to a negative value so that the the GameObject must be entirely off screen before registering as isOnScreen = false, as is the case for Enemy_0.

f. This section of the code sets an initial position for the enemy that was instantiated. It uses the BoundsCheck on this _MainCamera

GameObject to get the camWidth and camHeight and chooses an X position where the spawned enemy is entirely on screen horizontally. It then chooses a Y position where the enemy is just above the screen.

g. Invoke is called again. The reason that `Invoke()` is used instead of `InvokeRepeating()` is that we want to be able to dynamically adjust the amount of time between each enemy spawn. Once `InvokeRepeating()` is called, the invoked function is always called at the rate specified. Adding an `Invoke()` call at the end of `SpawnEnemy()` allows the game to adjust `enemySpawnPerSecond` on the fly and have it affect how frequently `SpawnEnemy()` is called.

3. Once you've typed this code and saved the file, switch back to Unity and follow these instructions:

a. Delete the instance of Enemy_0 from the Hierarchy (leaving the prefab in the Project pane alone, of course).

b. Select _MainCamera in the Hierarchy.

c. Open the disclosure triangle next to prefabEnemies in the *Main (Script)* component of _MainCamera and set the *Size* of prefabEnemies to 1.

d. Drag Enemy_0 from the Project pane into *Element 0* of the prefabEnemies array.

e. *Save your scene!* Have you been remembering?

If you didn't save your scene after creating all of those enemies, you really should have. There are all sorts of things beyond your control that could cause Unity to crash, and you really don't want to have to redo work. Getting into a habit of saving your scene frequently can save you a ton of wasted time and sorrow as a developer.

4. Play your scene. You should now see an Enemy_0 spawn about once every 2 seconds, travel down to the bottom of the screen, and

then disappear once it exits the bottom of the screen.

However, right now, when the _Hero collides with an enemy, nothing happens. This needs to be fixed, and to do so, we're going to have to look at layers.

Setting Tags, Layers, and Physics

As was presented in Chapter 28, "Prototype 1: Apple Picker," one of the things that layers control in Unity is which objects may or may not collide with each other. First, let's think about the Space SHMUP prototype. In this game, several different types of GameObjects could be placed on different layers and interact with each other in different ways:

• **Hero:** The _Hero ship should collide with enemies, enemy projectiles, and power-ups but should not collide with hero projectiles.

• **ProjectileHero:** Projectiles fired by _Hero should only collide with enemies.

• **Enemy:** Enemies should collide with _Hero & hero projectiles but not with power-ups.

• **ProjectileEnemy:** Projectiles fired by enemies should only collide with _Hero.

• **PowerUp:** Power-ups should only collide with _Hero.

To create these layers as well as some tags that will be useful later, complete these steps:

1. Open the *Tags & Layers* manager in the Inspector pane (*Edit* > *Project Settings* > *Tags and Layers*). Tags and physics layers are different from each other, but both are set here.

2. Open the disclosure triangle next to Tags. Click the + below the Tags list and enter the tag name for each of the tags shown in the left image of Figure 30.7. Note that in the middle of your typing the name of Tags Element 5, *PowerUpBox*, you may receive a console message ("Default GameObject Tag: PowerUp already registered"), which you can safely ignore.

Figure 30.7. TagManager showing tags and layer names for this prototype



In case it's difficult to see, the Tag names are: Hero, Enemy, ProjectileHero, ProjectileEnemy, PowerUp, and PowerUpBox.

3. Open the disclosure triangle next to *Layers*. Starting with User Layer 8, enter the layer names shown in the right image of Figure 30.7. Builtin Layers 0–7 are reserved by Unity, but you can set the names of User Layers 8–31.

The Layer names are: Hero, Enemy, ProjectileHero, ProjectileEnemy, and PowerUp.

4. Open the PhysicsManager (*Edit* > *Project Settings* > *Physics*) and set the *Layer Collision Matrix* as shown in Figure 30.8.

Figure 30.8. PhysicsManager with proper settings for this prototype



Note

In Unity, there are settings for both Physics and Physics2D. In this chapter, you should be setting Physics (the standard 3D PhysX physics library), not Physics2D.

As you experienced in Chapter 28, “Prototype 1: Apple Picker,” the grid at the bottom of the PhysicsManager sets which layers collide with each other. If there is a check, objects in the two layers are able to collide, if there is no check, they won’t. Removing checks can speed the execution of your game because it will test fewer objects versus each other for collision. As you can see in Figure 30.8, the Layer Collision Matrix we’ve chosen achieve the collision behavior we specified earlier.

Assign the Proper Layers to GameObjects

Now that the layers have been defined, you must assign the GameObjects you’ve created to the correct layer, as follows:

1. Select _Hero in the Hierarchy and choose *Hero* from the *Layer* pop-up menu in the Inspector. When Unity asks if you’d like to also assign the children of _Hero to this new layer, choose *Yes, change children*.

2. Set the tag of _Hero to *Hero* using the *Tag* pop-up menu in the Inspector. You do not need to change the tags of the children of _Hero.

3. Select all five of the Enemy prefabs in the Project pane and set them to the *Enemy* layer. When asked, elect to change the layer of their children as well.

4. Also set the tag of all Enemy prefabs to *Enemy*. You do not need to set the tags of the children of each enemy.

Making the Enemies Damage the Player

Now that the enemies and hero have colliding layers, we need to make them react to the collisions.

1. Open the disclosure triangle next to _Hero in the Hierarchy and select its child *Shield*. In the Inspector, set the Sphere Collider of

Shield to be a trigger (check the box next to *Is Trigger*). We don't need things to bounce off of Shield; we just need to know when they've hit.

2. Add the following bolded method to the end of the Hero C# script:

```
public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        print("Triggered: "+other.gameObject.name);
    }
}
```

3. Play the scene and try running into some enemies. You will see that you get a separate trigger event for each of the child GameObjects of the enemy (i.e., Cockpit and Wing) but not for the enemy itself. We need to be able to get the Enemy_0 GameObject that is the parent of Cockpit and Wing, and if we had even more deeply nested child GameObjects, we would need to still find this topmost or root parent.

Luckily, this is a pretty common thing to need to do, so it's part of the Transform component of any GameObject. If you call transform.root on any GameObject, it will give you the transform of the root GameObject, from which it is easy to get the GameObject itself.

4. Replace the OnTriggerEnter() code of the Hero C# script with these bolded lines:

```
public class Hero : MonoBehaviour {
    ...
    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
```

```
        print("Triggered: "+go.name);
    }
}
```

Now when you play the scene and run the ship into enemies, you should see that `OnTriggerEnter()` announces it has hit *Enemy_0(Clone)*, an instance of Enemy_0.

Tip

Iterative Code Development When prototyping on your own, this kind of console announcement test is something that you will do often to test whether the code you've written is working properly. I find that it is much better to do small tests along the way like this than to work on code for hours only to find at the end that something is causing a bug. Testing incrementally makes things *a lot* easier to debug because you know that you've only made slight changes since the last test that worked, so it's easier to find the place where you added a bug.

Another key element of this approach is using the debugger. Throughout the authoring of this book, any time I ran into something that worked a little differently than I expected, I used the debugger to understand what was happening. If you don't remember how to use the MonoDevelop debugger, I highly recommend rereading Chapter 24, "Debugging."

Using the debugger effectively is often the difference between solving your code problems and just staring at pages of code blankly for several hours. Try putting a debug breakpoint into the `OnTriggerEnter()` method you just modified and watching how code is called and variables change.

Iterative code development also has the same strengths as the iterative process of design, and it is the key to the agile development methodology discussed in Chapter 27, "The Agile Mentality."

---

5. Next, modify the `OnTriggerEnter()` method of the `Hero` class to make a collision with an enemy decrease the player's shield by 1 and destroy the enemy that was hit. It's also very important to make sure that the same parent GameObject doesn't trigger the collider twice (which can happen with very fast-moving objects where two child colliders of one object hit the Shield trigger in the same frame).

```
public class Hero : MonoBehaviour {
    ...
    public float            shieldLevel = 1;
    // This variable holds a reference to the last triggering
GameObject
    private GameObject      lastTriggerGo = null;                   // a

    ...

    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
        //print("Triggered: "+go.name);                             // b

        // Make sure it's not the same triggering go as last time
        if (go == lastTriggerGo) {                                  // c
            return;
        }
        lastTriggerGo = go;                                         // d

        if (go.tag == "Enemy") {  // If the shield was triggered by an
enemy
            shieldLevel--;       // Decrease the level of the shield by 1
            Destroy(go);         // ... and Destroy the enemy          // e
        } else {
            print("Triggered by non-Enemy: "+go.name);              //
f
        }
    }
}
```

a. This private field will hold a reference to the last GameObject that triggered _Hero's collider. It is initially set to `null`.

b. Comment out this line here.

c. If `lastTriggerGo` is the same as `go` (the current triggering GameObject), this collision is ignored as a duplicate, and the function returns (exits). This can happen if two child GameObjects of the same Enemy trigger the hero collider at the same time (that is, in the same single frame).

d. Assign `go` to `lastTriggerGo` so that it is updated before the next time `OnTriggerEnter()` is called.

e. `go`, the enemy GameObject, is destroyed by hitting the shield. Because the actual GameObject `go` that we're testing is the Enemy GameObject found by `transform.root`, this will delete the entire enemy (and by extension, all of its children), and not just one of the enemy's child GameObjects.

f. If _Hero collides with something that is not tagged Enemy, then this will print to the Console and let us know what it is.

6. Play the scene and try running into some ships. After running into more than a few, you may notice a strange shield behavior. The shield will loop back around to full strength after being completely drained. What do you think is causing this? Try selecting _Hero in the Hierarchy while playing the scene to see what's happening to the `shieldLevel` field.

Because there is no bottom limit to `shieldLevel`, it continues past 0 into negative territory. The Shield C# script then translates this into negative X offset values for Mat_Shield, and because the material's texture is set to loop, it looks like the shield is returning to full strength.

To fix this, we will convert `shieldLevel` to a property that protects and limits a new private field named `_shieldLevel`. The `shieldLevel`

property will watch the value of the `_shieldLevel` field and make sure that `_shieldLevel` never gets above 4 and that the ship is destroyed if `_shieldLevel` ever drops below 0. A protected field like `_shieldLevel` should be set to private because it does not need to be accessed by other classes; however, in Unity, private fields are not normally viewable in the Inspector. To remedy this, the line `[SerializeField]` is added above the declaration of _shieldLevel to instruct Unity to show it in the Inspector even though it is a private field. Properties are never visible in the Inspector, even if they're public.

7. In the Hero class, change the name of the public variable shieldLevel to _shieldLevel near the top of the class, set it to private, and add the `[SerializeField]` line:

```
    [Header("These fields are set dynamically")]
    [SerializeField]
    private float       _shieldLevel = 1; // Remember to add the
                                          // underscore
    // This variable holds a reference to the last triggering
GameObject
```

8. Add the `shieldLevel` property to the end of the Hero class.

```
public class Hero : MonoBehaviour {

    ...

    void OnTriggerEnter(Collider other) {
        ...
    }

    public float shieldLevel {
        get {
            return( _shieldLevel );                    // a
        }
        set {
            _shieldLevel = Mathf.Min( value, 4 );              // b
```

```
            // If the shield is going to be set to less than zero
            if (value < 0) {                                    // c
                Destroy(this.gameObject);
            }
        }
    }
}
```

a. The `get` clause just returns the value of `_shieldLevel`.

b. `Mathf.Min()` ensures that `_shieldLevel` is never set to a number higher than 4.

c. If the value passed into the set clause is less than 0, _Hero is destroyed.

The `shieldLevel--`; line in `OnTriggerEnter()` uses both the get and set clauses of the `shieldLevel` property. First, it uses the get clause to determine the current value of `shieldLevel`, then it subtracts 1 from that value and calls the set clause to assign that value back.

Restarting the Game

From your testing, you can see that the game gets exceedingly boring once _Hero has been destroyed. We'll now modify both the Hero and Main classes to call a method when _Hero is destroyed that waits for 2 seconds and then restarts the game.

1. Add a `gameRestartDelay` field near the top of the Hero class:

```
    public float          pitchMult = 30;
    public float          gameRestartDelay = 2f;

    [Header("These fields are set dynamically")]
```

2. Then add the following lines to the `shieldLevel` property definition in the Hero class:

```
        if (value < 0) {
            Destroy(this.gameObject);
            // Tell Main.S to restart the game after a delay
            Main.S.DelayedRestart( gameRestartDelay );
        }
```

When you initially type this into MonoDevelop, it will appear red because the DelayedRestart() function does not yet exist in the Main class.

3. Finally, add the following methods to the Main class to make this work.

```
public class Main : MonoBehaviour {
    ...

    public void SpawnEnemy() { ... }

    public void DelayedRestart( float delay ) {
        // Invoke the Restart() method in delay seconds
        Invoke( "Restart", delay );
    }

    public void Restart() {
        // Reload _Scene_0 to restart the game
        SceneManager.LoadScene( "_Scene_0" );
    }

}
```

4. Press Play to test the game. Now, once the player ship has been destroyed, the game waits a couple seconds and then restarts by reloading the scene.

If your lighting looks strange after you've reloaded the scene (e.g., your ship and enemy ships look a bit darker), then you may be experiencing a known bug with Unity's lighting system. Unity may

have now resolved this, but if you are seeing issues, there is an interim fix. Follow these directions to resolve it for this project:

1. From the menu bar, choose Window > Lighting.

2. Click the Scene button at the top of the Lighting pane.

3. Uncheck the Auto selection at the bottom of the Lighting pane (next to the Build button). This will stop Unity from constantly recalculating the global illumination settings.

4. To make sure the lighting is built properly, click the Build button at the bottom of the Lighting pane to manually calculate the global illumination.

5. Press Play to test, and you should see that the lighting is consistent even after reloading the scene.

Shooting (Finally)

Now that the enemy ships can hurt the player, it's time to give _Hero a way to fight back. This will be simple in this chapter and greatly expanded in the next one.

ProjectileHero, the Hero's Bullet

Follow these steps to create the Hero's bullet:

1. Create a cube named *ProjectileHero* in the Hierarchy with the following transform values:

ProjectileHero (Cube) P:[10,0,0] R:[0,0,0] S:[0.25,1,0.5]

2. Set both the tag and layer of ProjectileHero to *ProjectileHero*.

3. Create a new material named *Mat_Projectile*, place it in the _Materials folder of the Project pane, give it the *ProtoTools* >

UnlitAlpha shader, and assign it to the ProjectileHero GameObject.

4. Add a Rigidbody component to the ProjectileHero GameObject with these settings:

• *Use Gravity* to false (unchecked)

• *isKinematic* to false (unchecked)

• *Constraints*: freeze Z position and X, Y, and Z rotation (by checking them)

5. In the Box Collider component of the ProjectileHero GameObject, set Size.Z to 10. This will ensure that the projectile is able to hit anything that is slightly off of the XY (i.e., Z=0) plane.

6. Create a new C# script named *Projectile* and attach it to ProjectileHero. We'll edit the script later.

When finished with these steps, your settings should match those shown in Figure 30.9.

Figure 30.9. ProjectileHero with the proper settings showing the large Size.z of the Box Collider



7. Save your scene.

8. Attach a BoundsCheck script component to ProjectileHero as well. Set `keepOnScreen` to `false` and `radius` to -1. The BoundsCheck `radius` will not affect collisions with other GameObjects, it only affects when the ProjectileHero thinks it has gone off screen.

9. Make ProjectileHero into a prefab by dragging it from the Hierarchy into the _Prefabs folder in the Project pane. Then delete the instance remaining in the Hierarchy.

10. Save your scene—yes, save it again. As I've said, you want to save as often as you can.

Giving _Hero the Ability to Shoot

Now we add the capability for the Hero to shoot the bullet.

1. Open the Hero C# script and make the following bolded changes:

```
public class Hero : MonoBehaviour {
    ...
    public float            gameRestartDelay = 2f;
    public GameObject       projectilePrefab;
    public float            projectileSpeed = 40;

    ...

    void Update () {
        ...
        transform.rotation =
Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);

        // Allow the ship to fire
        if ( Input.GetKeyDown( KeyCode.Space ) ) {                    // a
            TempFire();
        }
    }

    void TempFire() {                                                 // b
        GameObject projGO = Instantiate<GameObject>(
projectilePrefab );
        projGO.transform.position = transform.position;
        Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
        rigidB.velocity = Vector3.up * projectileSpeed;
    }

    void OnTriggerEnter(Collider other) { ... }
```

```
    ...
}
```

a. This causes the ship to fire every time the space bar is pressed.

b. This method is named `TempFire()` because we will be replacing it in the next chapter.

2. In Unity, select _Hero in the Hierarchy and assign ProjectileHero from the Project pane to the projectilePrefab of the Hero script.

3. Save and click Play. Now, you can fire projectiles by pressing the space bar, but they don't yet destroy enemy ships, and they continue forever when they go off screen.

Scripting the Projectile

To script the projective follow these steps:

1. Open the Projectile C# script and make the following bolded changes. All we need the Projectile to do is destroy itself when it goes off screen. We'll add more in the next chapter.

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {

    private BoundsCheck     bndCheck;

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
    }

    void Update () {
        if (bndCheck.offUp) {                                   // a
            Destroy( gameObject );
        }
```

```
    }
}
```

a. If the Projectile goes off the top of the screen, destroy it.

2. And of course, remember to save your scene.

Allowing Projectiles to Destroy Enemies

You also need the capability of destroying enemies with the bullets.

1. Open the Enemy C# script and make the following bolded changes:

```
public class Enemy : MonoBehaviour {
    ...
    public virtual void Move() { ... }

    void OnCollisionEnter( Collision coll ) {
        GameObject otherRootGO =
coll.transform.root.gameObject;       // a
        if ( otherRootGO.tag == "ProjectileHero" ) {                  // b
            Destroy( otherRootGO );  // Destroy the Projectile
            Destroy( gameObject );   // Destroy this Enemy
GameObject
        } else {
            print( "Enemy hit by non-ProjectileHero: " +
otherRootGO.name );  Image// c
        }
    }
}
```

a. Get the root GameObject of whatever other transform was hit in the Collision.

b. If otherRootGO has the ProjectileHero tag, then destroy it and this Enemy instance.

c. If otherRootGO doesn't have the ProjectileHero tag, print the name of what was hit to the Console so that we can debug what happened. If you wish to test this, you can temporarily remove the ProjectileHero tag from the ProjectileHero prefab and shoot an Enemy.

Now, when you press Play, Enemy_0s will come down the screen, and you can shoot them with projectiles. That's it for this chapter—you've got a nice, simple prototype—but in the next chapter, we'll expand on this considerably by adding additional enemies, three kinds of power-ups, two kinds of guns, and a lot more interesting coding tricks.

Summary

In most chapters, I include a next steps chapter to give you ideas of what you can do to extend the project and push yourself. However, for the Space SHMUP prototype, we're going to do some of those things in the next chapter and demonstrate some new coding concepts in the process. Take a break now so you can approach the next chapter fresh and congratulate yourself on a prototype well done.

Chapter 31. Prototype 3.5: Space shmup Plus

Most of these prototype chapters end with a "Next Steps" section that suggests things you may want to add to the game. This chapter shows you the steps to do just that for the Space SHMUP game that you built in the previous chapter.

In this chapter, you add power-ups, multiple enemies, and different weapon types to the Space SHMUP game. In doing so, you learn about class

inheritance, enums, function delegates, and several other important topics. And, you'll make the game a lot more fun!

Getting Started: Prototype 3.5

At the end of the previous chapter, we had a pretty basic version of a space shooter. In this chapter, we'll make it more fun and expansive. In case you had any issues with the game as built in the previous chapter, you can download it from the website for the book.

Set Up the Project for this Chapter

Rather than following the standard project setup procedure, you have two options for this chapter:

1. Make a duplicate of your project folder from the previous chapter.

2. Download a finished version of the last chapter from the website for the book. To do so, find Chapter 31 at http://book.prototools.net.

Once you've acquired a project folder, open _Scene_0 in Unity, and we'll get started.

Programming Other Enemies

Let's start by expanding the kinds of enemies that our hero will face. Later, we'll give our hero the chance to fight back against these more dire enemies.

1. Create new C# scripts named *Enemy_1*, *Enemy_2*, *Enemy_3*, and *Enemy_4*.

2. Place these scripts into the *__Scripts* folder in the Project pane.

3. Assign each of these scripts to their respective Enemy_# prefab in the Project pane.

We'll work on the script for each enemy in turn.

Enemy_1

Enemy_1 will move down the screen in a sine wave. It extends the Enemy class, which means that it inherits all of the fields, functions, and properties of Enemy (as long as they are public or protected; private elements are not inherited). For more information on classes and class inheritance (including method overriding), check out Chapter 25, "Classes."

1. Open the Enemy_1 script in MonoDevelop and enter the following bolded code:

```
using UnityEngine;
using System.Collections;

// Enemy_1 extends the Enemy class
public class Enemy_1 : Enemy {                                  // a
    [Header("Enemy_1 Fields")]
```

```
    // # seconds for a full sine wave
    public float    waveFrequency = 2;
    // sine wave width in meters
    public float    waveWidth = 4;
    public float    waveRotY = 45;

    private float    x0; // The initial x value of pos
    private float    birthTime;

    void Start() {
        // Set x0 to the initial x position of Enemy_1
        x0 = pos.x;                                          // b

        birthTime = Time.time;
    }

    // Override the Move function on Enemy
    public override void Move() {                                  // c
        // Because pos is a property, you can't directly set pos.x
        //   so get the pos as an editable Vector3
        Vector3 tempPos = pos;
        // theta adjusts based on time
        float age = Time.time - birthTime;
        float theta = Mathf.PI * 2 * age / waveFrequency;
        float sin = Mathf.Sin(theta);
        tempPos.x = x0 + waveWidth * sin;
        pos = tempPos;

        // rotate a bit about y
        Vector3 rot = new Vector3(0, sin*waveRotY, 0);
        this.transform.rotation = Quaternion.Euler(rot);

        // base.Move() still handles the movement down in y
        base.Move();                                          // d
    }
}
```

a. As an extension of the Enemy class, Enemy_1 inherits the public speed, fireRate, health, and score fields as well as the public pos property and the public Move() method. However, it does not inherit the private bndCheck field, which we'll discuss more in the next step.

b. Setting x0 to the initial x of this Enemy_1 works fine here in Start() because the position will have already been set by the time Start() is called. Had this line been put in an Awake() method, it would have been incorrect because Awake() is called in the instant that a GameObject is instantiated (i.e., before the position is set by the Main:SpawnEnemy() method (in Main.cs).

Another reason to avoid adding an Awake() method to Enemy_1 is that it would override the Awake() method on Enemy. Awake(), Start(), Update(), and other built-in MonoBehaviour methods are scripted in a special manner so that you don't need to write virtual or override on them to have them be overridden. However, this is different from standard C# style.

c. In normal C# methods like the `Move()` method that we wrote in Enemy, you must declare the method `virtual` in the super class and `override` in the subclass for it to properly override the function in the subclass. Because `Move()` is marked as a virtual method in the `Enemy` superclass, we are able to override it here and replace it with another function (also named `Move()`).

d. `base.Move()` calls the `Move()` function on the superclass `Enemy`.

2. Back in Unity, select _MainCamera in the Hierarchy and change *Element 0* of `prefabEnemies` from Enemy_0 to Enemy_1 (which is the Enemy_1 prefab in the Project pane) in the Main (Script) component. This will allow us to test with Enemy_1 instead of Enemy.

3. Press Play. You'll see that the Enemy_1 ship will now appear instead of Enemy_0, and it will move in a wave. However, you'll see in the Scene pane that Enemy_1 instances don't disappear when they go off the bottom of the screen. This is because Enemy_1 doesn't have a BoundsCheck component attached to it.

4. We want to attach BoundsCheck to the Enemy_1 prefab, and we also want to keep all the same values that it has set on the Enemy_0 prefab. To do this, follow these steps to learn another way to attach a script to a GameObject:

a. Select Enemy_0 in the _Prefabs folder of the Project pane.

b. In the Inspector for Enemy_0, click the gear icon in the top-right corner of the *BoundsCheck (Script)* component and choose *Copy Component*.

c. Select Enemy_1 in the _Prefabs folder of the Project pane.

d. In the Inspector for Enemy_1, click the gear icon in the top-right corner of the Transform component and choose *Paste Component as New*. This will attach a new *BoundsCheck (Script)* component to the Enemy_1 prefab that comes with all of the same settings as those on the BoundsCheck component we copied from the Enemy_0 prefab.

Making bndCheck Protected Instead of Private

This is a somewhat subtle yet important point. The field declaration of bndCheck in the Enemy class is currently private.

```
private BoundsCheck bndCheck;
```

This means that it can be seen within the Enemy class but not in any other classes, including Enemy_1, even though Enemy_1 is a subclass of Enemy. This means that the Awake() and Move() methods on Enemy can see and interact with bndCheck, yet the override Move() method on Enemy_1 doesn't know it exists. To test this:

1. Open the Enemy_1 script and add the bolded line to the end of the Move() method:

```
public override void Move() {
    ...
    base.Move();

    print( bndCheck.isOnScreen );
}
```

Because `bndCheck` is a private variable of the Enemy class, it appears red here in Enemy_1 and cannot be read. To fix this, we need to make `bndCheck` *protected* instead of public. Like private variables, protected variables can't be seen by other classes, but unlike private variables, protected variables can be seen by and inherited by subclasses.



2. Open the Enemy script and change `bndCheck` from `private` to `protected`.

```
protected BoundsCheck bndCheck;
```

Now, if you check the Enemy_1 script, `bndCheck.isOnScreen` will no longer be red, and your code will compile properly.

3. Return to the Enemy_1 script and comment out the print() line.

```
// print( bndCheck.isOnScreen ); // This line is now commented out.
```

4. Press play, and you should see the Enemy_1s now disappear after they exit the bottom of the screen.

Tip

Sphere Colliders Only Scale Uniformly You might have noticed that the collision with Enemy_1 actually occurs slightly before the projectile (or _Hero) touches the wing. If you select Enemy_1 in the

Project pane and drag an instance into the scene, you will see that the green collider sphere around Enemy_1 doesn't scale to match the flat ellipse of the wing. This isn't a huge problem, but it is something to be aware of. A Sphere Collider will scale with the largest single component of scale in the transform. (In this case, because wing has a Scale.x of 6, the Sphere Collider scales up to that.)

If you want, you can try other types of colliders to see whether one of them will scale to match the wing more accurately. A Box Collider will scale nonuniformly. You can also approximate one direction being much longer than the others with a Capsule Collider. A Mesh Collider will match the scaling most exactly, but Mesh Colliders are much slower than other types. This shouldn't be a problem on a modern high-performance PC, but Mesh Colliders are often much too slow for mobile platforms like iOS or Android.

If you choose to give Enemy_1 a Mesh Collider, then when it rotates about the y axis, it will move the edges of the wing out of the XY (that is, z=0) plane. This is why the ProjectileHero prefab has a Box Collider Size.z of 10 (to make sure that it can hit the wingtips of Enemy_1 even if they are not in the XY plane).

Preparing for the Other Enemies

The remaining enemies make use of linear interpolation, an important development concept that is described in Appendix B. You saw a very simple interpolation in Chapter 29, "Prototype 2: Mission Demolition," but these will be a bit more interesting. Take a moment to read the "Interpolation" section of Appendix B, "Useful Concepts," before tackling the remaining enemies.

Enemy_2

Enemy_2 will move via a linear interpolation that is heavily eased by a sine wave. It will rush in from the side of the screen, slow, reverse

direction for a bit, slow, and then fly off the screen along its initial velocity. Only two points will be used in this interpolation, but the u value will be drastically curved by a sine wave. The easing function for the u value of Enemy_2 will be along the lines of

$u = u + 0.6 * \text{Sin}(2\pi * u)$

This is one of the easing functions explained in the “Interpolation” section of Appendix B.

1. Attach a BoundsCheck script to the Enemy_2 prefab in the _Prefabs folder of the Project pane. The BoundsCheck component will be used extensively for Enemy_2.

2. In the Enemy_2 prefab BoundsCheck Inspector, set the radius = 3 and keepOnScreen = false.

3. Open the Enemy_2 C# script and enter the following code. After you have the code working, you’re welcome to adjust the easing curve and see how it affects the motion.

```
using UnityEngine;
using System.Collections;

public class Enemy_2 : Enemy {                                  // a
   // Enemy_2 uses a Sin wave to modify a 2-point linear
interpolation
   public Vector3        p0, p1;
   public float          birthTime;
   public float          lifeTime = 10;
   // Determines how much the Sine wave will affect movement
   public float          sinEccentricity = 0.6f;

   void Start () {
      // Pick any point on the left side of the screen
      p0 = Vector3.zero;                                        // b
      p0.x = -bndCheck.camWidth - bndCheck.radius;
      p0.y = Random.Range( -bndCheck.camHeight,
```

```
bndCheck.camHeight );

        // Pick any point on the right side of the screen
        p1 = Vector3.zero;
        p1.x = bndCheck.camWidth + bndCheck.radius;
        p1.y = Random.Range( -bndCheck.camHeight,
bndCheck.camHeight );

        // Possibly swap sides
        if (Random.value > 0.5f) {
            // Setting the .x of each point to its negative will move it
to
            // the other side of the screen
            p0.x *= -1;
            p1.x *= -1;
        }

        // Set the birthTime to the current time
        birthTime = Time.time;                                  // c
    }

    public override void Move() {
        // Bézier curves work based on a u value between 0 & 1
        float u = (Time.time - birthTime) / lifeTime;

        // If u>1, then it has been longer than lifeTime since
birthTime
        if (u > 1) {
            // This Enemy_2 has finished its life
            Destroy( this.gameObject );                         // d
            return;
        }

        // Adjust u by adding a U Curve based on a Sine wave
        u = u + sinEccentricity*(Mathf.Sin(u*Mathf.PI*2));

        // Interpolate the two linear interpolation points
```

```
        pos = (1-u)*p0 + u*p1;
    }
}
```

a. Enemy_2 also extends the Enemy superclass.

b. This section will choose a random point on the left side of the screen. It initially chooses an x position that is just off the left side of the screen: -bndCheck.camWidth is the left side of the screen, and -bndCheck.radius makes sure that the Enemy_2 is entirely off screen.

Then, a random Y position is chosen that is between the bottom of the screen (-bndCheck.camHeight) and the top of the screen (bndCheck.camHeight).

c. birthTime is used by the interpolation in the Move() function.

d. If it has been longer than lifeTime since the birthTime, then u will be greater than 1, and this Enemy_2 will be destroyed.

4. Swap the Enemy_2 prefab into the Element 0 slot of Main.S.prefabEnemies using the _MainCamera Inspector and press Play.

As you can see the easing function causes each Enemy_2 to have very smooth movement that goes forth, back, and forth between the points it has selected on either side of the screen.

Enemy_3

Enemy_3 will use a Bézier curve to swoop down from above, slow, and fly back up off the top of the screen. For this example, we will use a simple version of the three-point Bézier curve function. In the “Recursive Functions” section of Appendix B you can find a recursive version of the Bézier curve function that can use any number of points (not just three).

1. Attach BoundsCheck to the Enemy_3 prefab in the _Prefabs folder of the Project pane.

2. In the Enemy_3 prefab BoundsCheck Inspector, set radius = 2.5 and keepOnScreen = false.

3. Open the Enemy_3 script and enter the following code:

```
using UnityEngine;
using System.Collections;

public class Enemy_3 : Enemy {                    // Enemy_3 extends Enemy

    // Enemy_3 will move following a Bezier curve, which is a linear
    //   interpolation between more than two points.
    public Vector3[]       points;
    public float           birthTime;
    public float           lifeTime = 5;

    // Again, Start works well because it is not used by Enemy
    void Start () {
        points = new Vector3[3]; // Initialize points

        // The start position has already been set by Main.SpawnEnemy()
        points[0] = pos;

        // Set xMin and xMax the same way that Main.SpawnEnemy() does
        float xMin = -bndCheck.camWidth + bndCheck.radius;
        float xMax =  bndCheck.camWidth - bndCheck.radius;

        Vector3 v;
        // Pick a random middle position in the bottom half of the screen
```

```
        v = Vector3.zero;
        v.x = Random.Range( xMin, xMax );
        v.y = -bndCheck.camHeight *  Random.Range( 2.75f, 2 );
        points[1] = v;

        // Pick a random final position above the top of the screen
        v = Vector3.zero;
        v.y = pos.y;
        v.x = Random.Range( xMin, xMax );
        points[2] = v;

        // Set the birthTime to the current time
        birthTime = Time.time;
    }

    public override void Move() {
        // Bezier curves work based on a u value between 0 & 1
        float u = (Time.time - birthTime) / lifeTime;

        if (u > 1) {
            // This Enemy_3 has finished its life
            Destroy( this.gameObject );
            return;
        }

        // Interpolate the three Bezier curve points
        Vector3 p01, p12;
        p01 = (1-u)*points[0] + u*points[1];
        p12 = (1-u)*points[1] + u*points[2];
        pos = (1-u)*p01 + u*p12;
    }
}
```

4. Now try swapping Enemy_3 into the Element 0 of `prefabEnemies` on _MainCamera.

5. Press play to see the movement of these new enemies. After playing for a bit, you'll notice a couple things about Bézier curves:

a. Even though the midpoint is at or below the bottom of the screen, no Enemy_3 ever gets that far down. That is because a Bézier curve touches both the start and end points but is only influenced by the midpoint.

b. Enemy_3 slows down a lot in the middle of the curve. This is also a feature of the math that makes Bézier curves work.

6. To improve the motion along the Bézier curve and reduce the slowdown at the bottom f the curve, add the following bold line to the Enemy_3 `Move()` method. This will add easing to the Enemy_3 movement that will speed up the middle of the curve:

```
public override void Move() {
    ...
    // Interpolate the three Bezier curve points
    Vector3 p01, p12;
    u = u - 0.2f*Mathf.Sin(u*Mathf.PI*2);
    p01 = (1-u)*points[0] + u*points[1];
    ...
}
```

Saving Enemy_4 for Later

For now, we're going to wait before implementing Enemy_4. This is because we first need to make some changes to projectiles and how they work. At this point, players can destroy any enemy with a single shot. In the next section, we'll change that and add the ability to have different kinds of weapons on the ship.

Shooting Revisited

The way that we managed firing Projectiles in the previous chapter was fine for a rough prototype, but we need to add additional

capabilities to get this game to the next level. In this prototype, we will build two different kinds of weapons with the ability to expand that in the future. To enable this, we will create a WeponDefinition class that will allow us to define the behavior of each type of weapon.

The WeaponType Enum

As described in Chapter 29, “Prototype 2 - Mission Demolition,” an enum—short for enumeration—is a way to associate various options together into a new kind of variable. In the example for this chapter, we create an enum called WeaponType that includes all possible options for both weapon and shield types. This will be used for both the type of any weapon and the type of any power-up.

1. Right click on the __Scripts folder in the Project pane and choose Create > C# Script. This will create a *NewBehaviourScript* in the __Scripts folder.

2. Rename NewBehaviourScript to *Weapon*.

3. Open the Weapon script in MonoDevelop and enter the following code. The public enum WeaponType declaration should go between `using System.Collections;` and `public class Weapon : MonoBehaviour {`.

```
using UnityEngine;
using System.Collections;

/// <summary>
/// This is an enum of the various possible weapon types.
/// It also includes a "shield" type to allow a shield power-up.
/// Items marked [NI] below are Not Implemented in the IGDPD book.
/// </summary>
public enum WeaponType {
    none,       // The default / no weapon
```

```
    blaster,    // A simple blaster
    spread,     // Two shots simultaneously
    phaser,     // [NI] Shots that move in waves
    missile,    // [NI] Homing missiles
    laser,      // [NI]Damage over time
    shield      // Raise shieldLevel
}

public class Weapon : MonoBehaviour {
    ...             // The Weapon class will be filled in later in the chapter.
}
```

As a public enum outside of the Weapon class, WeaponType can be seen by and used by any other script in the project. We'll make use of this extensively throughout the rest of this chapter, and the weapons will actually be defined via WeaponType in the Main C# script rather than the Weapon script.

For more information on enums, look at the Appendix B, "Useful Concepts," subsection titled "Enum" under "C# and Unity coding concepts."

The Serializable WeaponDefinition Class

We now need to create a class to define the various types of weapons. Unlike most of the other classes that you've created in this book, this will not be a subclass of MonoBehaviour, and it will not be attached individually to a GameObject. Instead, this will be a very simple, separate, public class that is defined within the Weapon C# script, just as the public enum WeaponType was.

Another important aspect of this class is that it is *serializable*, meaning that it can be seen and edited within the Unity Inspector!

Open the Weapon script and enter the following bolded code between the `public enum WeaponType` definition and the `public class Weapon` definition.

```
public enum WeaponType {
    ...
}

 /// <summary>
/// The WeaponDefinition class allows you to set the properties
/// of a specific weapon in the Inspector. The Main class has
/// an array of WeaponDefinitions that makes this possible.
/// </summary>
[System.Serializable]                                          // a
public class WeaponDefinition {                                    // b
    public WeaponType   type = WeaponType.none;
    public string       letter;                 // Letter to show on the
power-up
    public Color        color = Color.white;    // Color of Collar &
power-up
    public GameObject   projectilePrefab;       // Prefab for
projectiles
    public Color        projectileColor = Color.white;
    public float        damageOnHit = 0;        // Amount of damage
caused
    public float        continuousDamage = 0;   // Damage per
second (Laser)
    public float        delayBetweenShots = 0;
    public float        velocity = 20;          // Speed of projectiles
}

public class Weapon : MonoBehaviour {
    ...                 // The Weapon class will be filled in later in the
chapter.
}
```

a. The [System.Serializable] attribute causes the class defined immediately after it to be serializable and editable within the Unity Inspector. Some classes are too complex to be serializable, but WeaponDefinition is simple enough to where it will work.

b. Each of the fields of WeaponDefinition can be altered to change an aspect of the bullets fired by our ship.

As described in the code comments, the enum `WeaponType` defines all the possible weapon types and power-up types. `WeaponDefinition` is a class that combines a `WeaponType` with several variables that will be useful for defining each weapon.

Modifying Main to use WeaponDefinition and WeaponType

Now we need to use the new WeaponType enum and WeaponDefinition class in Main. We are doing this in the Main class because it is responsible for spawning enemies and eventually power-ups.

1. Add the following WeaponDefinitions declaration to the `Main` class and save.

```
public class Main : MonoBehaviour {
    ...
    public float            enemyDefaultPadding = 1.5f; // Padding for
                                                        // position
    public WeaponDefinition[]   weaponDefinitions;

    private BoundsCheck     bndCheck;

    ...void Awake() {...}
}
```

2. Select _MainCamera in the Hierarchy. You should now see a `weaponDefinitions` array in the Main (Script) component Inspector.

3. Click the disclosure triangle next to weaponDefinitions in the Inspector and set the Size of the array to 3.

4. Enter settings for the three WeaponDefinitions as shown in Figure 31.1. You can see that the WeaponType enum appears in the Inspector as a pop-up menu (though like other things in the

Inspector, the enum types have been capitalized). The colors you pick don’t have to be exactly right, but it is important that the alpha value of each color is set to a fully-opaque value of 255, which appears as a white bar beneath the color swatch.

Figure 31.1. Settings for the WeaponDefinitions of blaster, spread, and shield on Main



Warning

COLORS SOMETIMES DEFAULT TO AN INVISIBLE ALPHA When you create a serializable class like `WeaponDefinition` that includes color fields, the alpha values of those colors can default to 0 (that is, invisible). To fix this, make sure that the white lines under each of your color definitions are actually white (and not black). If you click on the color itself, you will be presented with four values to set (R, G, B, and A). Make sure that A is set to 255 (that is, fully opaque) or your shots will be invisible.

If you are using OS X and have chosen to use the OS X color picker in Unity instead of the default one, the A value is set by the Opacity slider at the bottom of the color picker window (which should be set to 100% for these colors).

A Generic Dictionary for WeaponDefinitions

In order to make it easier to access the WeaponDefinitions, we’re going to copy them at runtime from the `WeaponDefinitions` array to a private *Dictionary* field named `weaponDict`. A Dictionary is a type of generic collection like List. However, where a List is an ordered (linear) collection, Dictionaries have a *key* type and *value* type, with the key used to retrieve the value. Dictionaries can be a good way to store a very large number of things because accessing any one of those things is a *constant time* operation, meaning that it takes the

same small amount of time regardless of where that thing is in the data structure. Contrast this with a List or array; if you were searching through the `WeaponDefinitions` array for the WeaponDefinition with the type `blaster`, you would encounter it immediately, while searching for the WeaponDefinition with the type `shield` would take three times as long.

Here, the `weaponDict` Dictionary has the enum `WeaponType` as the key and the class `WeaponDefinition` as the value. Unfortunately, Dictionaries do not appear in the Unity Inspector, or we would have just used one from the start. Instead, the `weaponDict` Dictionary is defined in the `Awake()` method of the Main class and then used by the static function `Main.GetWeaponDefinition()`.

1. Open the Main script in MonoDevelop and enter the following bold code.

```
public class Main : MonoBehaviour {
    static public Main S;
    static Dictionary<WeaponType, WeaponDefinition>
WEAP_DICT;          // a
    ...

    void Awake() {
        ...
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond );

        // A generic Dictionary with WeaponType as the key
        WEAP_DICT = new Dictionary<WeaponType,
WeaponDefinition>();     // a
        foreach( WeaponDefinition def in weaponDefinitions )
{          // b
            WEAP_DICT[def.type] = def;
        }
    }
}
```

a. Dictionaries are declared and defined with both a key type and value type. Making `WEAP_DICT` static but protected means that any instance of `Main` can access it and any static method of `Main` can access it, which we'll take advantage of later.

b. This loop iterates through each element of the `weaponDefinitions` array and creates an entry in the `WEAP_DICT` dictionary that matches it.

Next, we need to create a static function to allow other classes to get access to the data in `WEAP_DICT`. Because `WEAP_DICT` is static as well, any static method of the `Main` class can access it. Making the new `GetWeaponDefinition()` method public and static allows any other code within the project to call it as `Main.GetWeaponDefinition()`.

2. Add the following bolded code to the end of the Main C# script:

```
public class Main : MonoBehaviour {
    ...
    public void Restart() {
        // Reload _Scene_0 to restart the game
        SceneManager.LoadScene( "_Scene_0" );
    }

    /// <summary>
    /// Static function that gets a WeaponDefinition from the
WEAP_DICT
    /// static protected field of the Main class.
    /// </summary>
    /// <returns>The WeaponDefinition or, if there is no
WeaponDefinition
    /// with the WeaponType passed in, returns a new
WeaponDefinition
    /// with a WeaponType of none..</returns>
    /// <param name="wt">The WeaponType of the desired
WeaponDefinition
    /// </param>
    static public WeaponDefinition GetWeaponDefinition(
```

```
WeaponType wt ) { Ê// a
      if (WEAP_DICT.ContainsKey(wt)) {                    // b
         return( WEAP_DICT [wt]);
      }

      return( new WeaponDefinition() );                   // c
   }
}
```

a. The documentation in the code above this function now includes not only a summary but also descriptions of what is returned and the parameter passed in.

b. This if statement checks to make sure that `WEAP_DICT` has an entry with the key that was passed in as `wt`. If you try to retrieve an entry that isn't there (for example, if you try `WEAP_DICT[WeaponType.phaser]`), it will throw an error.

In the case that there *is* an element with the proper WeaponType, that WeaponDefinition is returned.

c. If there is no `WeaponDefinition` in `WEAP_DICT` with the proper `WeaponType` key, a new `WeaponDefinition` with a type of `WeaponType.none` is returned.

Modifying the Projectile Class to Use WeaponDefinitions

Now, we need to alter the Projectile class considerably to make it use our new WeaponDefinitions.

1. Open the `Projectile` class in MonoDevelop.

2. IMPORTANT: Delete the OnCollisionEnter() method entirely. We will be moving collision detection to the Enemy class.

3. Make your code for the `Projectile` class match the following code listing.

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {
    private BoundsCheck     bndCheck;
    private Renderer        rend;

    [Header("These fields are set dynamically")]
    public Rigidbody        rigid;
    [SerializeField]                                    // a
    public WeaponType       _type;                          // b

    // This public property masks the field _type and takes action
when it
    // is set
    public WeaponType       type {                          // c
        get {
            return( _type );
        }
        set {
            SetType( value );                           // c
        }
    }

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
        rend = GetComponent<Renderer>();                    // d
        rigid = GetComponent<Rigidbody>();
    }

    void Update () {
        if (bndCheck.offUp) {
            Destroy( gameObject );
        }
    }

    /// <summary>
```

```
    /// Sets the _type private field and colors this projectile to match
the
    ///   WeaponDefinition.
    /// </summary>
    /// <param name="eType">The WeaponType to use.</param>
    public void SetType( WeaponType eType ) {                    // e
        // Set the _type
        _type = eType;
        WeaponDefinition def = Main.GetWeaponDefinition( _type );
        rend.material.color = def.projectileColor;
    }
}
```

a. The `[SerializeField]` attribute above the `_type` declaration forces `_type` to be visible and settable in the Unity Inspector even though it is private. However, you should *not* set this field in the Inspector.

b. It is common practice throughout this book to name a private field accessed through a property with an underscore and the property name (e.g., the private field `_type` is accessed via the property `type`).

c. The type property gets `_type` normally, but it calls the `SetType()` method as its setter, allowing us to do more than just set `_type`.

d. We need to use the Renderer component attached to this GameObject in the `SetType()` method, so it is cached here.

e. `SetType()` not only sets the `_type` private field but also colors the projectile to match the color based on the `weaponDefinitions` in `Main`.

Using a Function Delegate to Fire

In this game prototype, the `Hero` class will have a function delegate `fireDelegate` that is called to fire all weapons, and each Weapon attached to it will add its individual `Fire()` target method to `fireDelegate`.

1. Before continuing, please read the “Function Delegates” section of Appendix B, “Useful Concepts.”

2. Add the following bold code to the `Hero` class:

```
public class Hero : MonoBehaviour {
    ...
    private GameObject      lastTriggerGo = null;

    // Declare a new delegate type WeaponFireDelegate
    public delegate void WeaponFireDelegate();                    // a
    // Create a WeaponFireDelegate field named fireDelegate.
    public WeaponFireDelegate fireDelegate;


    void Awake() {
        S = this;  // Set the Singleton
        fireDelegate += TempFire;                                 // b
    }


    void Awake() { ... }

    void Update () {
        ...
        transform.rotation =
Quaternion.Euler(yAxis*pitchMult,xAxis*rollMultÊ,0);

        // Allow the ship to fire
//        if ( Input.GetKeyDown(KeyCode.Space) ) {                // c
//            TempFire();
//        }

        // Use the fireDelegate to fire Weapons
        // First, make sure the button is pressed Axis("Jump")
        // Then ensure that fireDelegate isn't null to avoid an error
        if (Input.GetAxis("Jump") == 1 && fireDelegate != null)
```

```
{      // d
          fireDelegate();                                  // e
       }
   }

   void TempFire() {                                       // f
      GameObject projGO = Instantiate<GameObject>(
projectilePrefab );
      projGO.transform.position = transform.position;
      Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
//       rigidB.velocity = Vector3.up * projectileSpeed;          // g

      Projectile proj = projGO.GetComponent<Projectile>
();          // h
      proj.type = WeaponType.blaster;
      float tSpeed = Main.GetWeaponDefinition( proj.type
).velocity;
      rigidB.velocity = Vector3.up * tSpeed;
   }

   void OnTriggerEnter(Collider other) { ... }
   ...
}
```

a. Though both are public, neither the `WeaponFireDelegate()` delegate type nor the `fireDelegate` field will appear in the Unity Inspector.

b. Adding `TempFire` to the `fireDelegate` causes `TempFire` to be called any time `fireDelegate` is called like a function (see `// e`).

Note that when we add `TempFire` to the `fireDelegate`, we don't follow the method name `TempFire` with parentheses. This is because we are adding the method itself rather than calling the method and adding the result it returns (which is what would happen if we put parentheses after the name of the method).

c. Be sure that you comment out (or delete) the entire section that was inside the `if ( Input.GetKeyDown(KeyCode.Space) ) { ... }` statement.

d. `Input.GetAxis("Jump")` is equal to 1 when the space bar or jump button on a controller is pressed.

If `fireDelegate` is called when it has no methods assigned to it, it will throw an error. To avoid this, `fireDelegate != null` is tested to see whether it is null before calling it.

e. `fireDelegate` is called here as if it were a function. This, in turn, calls all of the functions that have been added to the `fireDelegate` delegate (at this point, this means that it will call `TempFire()`).

f. `TempFire()` is now used by the `fireDelegate` to fire a standard blaster shot. We will replace `TempFire()` later when we make the Weapon class.

g. You need to either comment out or delete this line.

h. This new section pulls information from the WeaponType of the Projectile class and uses it to set the velocity of the `projGO` GameObject.

3. Press Play in Unity and try firing. You should fire a lot of blaster shots very rapidly. In the next section, we'll add a `Weapon` class that will better manage firing and will replace `TempFire()` with the `Fire()` function on that `Weapon` class.

Moving the Damage Function to Enemy

Now that we've removed the simplistic OnCollisionEnter function from the Projectile, we need to write a more robust one to be used by each Enemy.

1. IMPORTANT: If you haven't yet, delete the OnCollisionEnter() method entirely. We will be moving collision detection to the Enemy

class.

2. Open the Enemy C# script and add the following bolded code:

```
using UnityEngine;              // Required for Unity

using System.Collections;       // Required for Arrays & other
Collections

public class Enemy : MonoBehaviour {
    [Header("Set in the Unity Inspector")]

    public float      speed = 10f;     // The speed in m/s

    public float      fireRate = 0.3f; // Seconds/shot (Unused)
    public float      health = 10;

    public int        score = 100; // Points earned for destroying this

    private BoundsCheck bndCheck;

    void Awake() {
        bndCheck = GetComponent<BoundsCheck>();
    }

    // This is a Property: A method that acts like a field
    public Vector3 pos {                                    // b
        get {
            return( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }

    void Update() {
        Move();
```

```
        if ( bndCheck != null && bndCheck.offDown ) {
            // We're off the bottom, so destroy this GameObject
            Destroy( gameObject );
        }
    }

    public virtual void Move() {
        Vector3 tempPos = pos;                                  // a
        tempPos.y -= speed * Time.deltaTime;                    // a
        pos = tempPos;                                          // a
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject otherGO = coll.gameObject;                   // a
        if ( otherGO.tag == "ProjectileHero" ) {                // b
            Destroy( otherGO );  // Destroy the Projectile
            Destroy( gameObject );   // Destroy this Enemy GameObject
        } else {
            print( "Enemy hit by non-ProjectileHero: " + otherGO.name ); Ê// c
        }
    }
}
```

a. Get the root GameObject of whatever other transform was hit in the Collision.

b. If otherGO has the ProjectileHero tag, then destroy it and this Enemy instance.

c. If otherGO doesn't have the ProjectileHero tag, print the name of what was hit to the Console so that we can debug what happened. If you wish to test this, you can temporarily remove the ProjectileHero tag from the ProjectileHero prefab and shoot an Enemy.

This new OnCollisionEnter method on Enemy will propagate to each of the Enemy_# subclasses.

Creating a Weapon Object to Fire Projectiles

We'll start with the artwork for the new Weapon GameObject. The benefit of the Weapon is that we can attach as many as we want to _Hero, and each one can add itself to the `fireDelegate` on the Hero class and then be fired in concert when `fireDelegate` is called like a function.

1. In the Hierarchy, create an empty GameObject, name it *Weapon*, and give it the following structure and children:



2. Remove the Collider component from both Barrel and Collar by selecting them individually and then right-clicking on the name of the Box Collider component and choosing *Remove Component* from the pop-up menu. You can also click the gear to the right of the Box Collider name to get the same menu.

3. Now, create a new material named *Mat_Collar* inside of the _Materials folder in the Project pane.

4. Drag this material on to Collar to assign it. In the Inspector, choose *ProtoTools* > *UnlitAlpha* from the Shader pop-up menu (see Figure 31.2).

Figure 31.2. Weapon with the Collar selected and proper material and shader selected



5. Attach the Weapon C# script to the Weapon GameObject in the Hierarchy.

6. Drag the Weapon GameObject into the _Prefabs folder in the Project pane to make it a prefab.

7. Make the Weapon instance in the Hierarchy a child of _Hero and check that its position is [0,2,0]. This should place the Weapon on the nose of the _Hero ship as is shown in Figure 31.2.

8. Save your scene! Are you remembering to save constantly?

```
public class Projectile : MonoBehaviour {
    private BoundsCheck     bndCheck;
    private Renderer        rend;

    [Header("These fields are set dynamically")]
    public Rigidbody        rigid;
    [SerializeField]
    public WeaponType       _type;

    ...

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
        rend = GetComponent<Renderer>();
        rigid = GetComponent<Rigidbody>();
    }
    ...
}
```

Adding Firing to the Weapon C# Script

To add firing to the weapon script, do the following:

1. Start by disabling the TempFire() script in Hero. Open the Hero C# script in MonoDevelop and comment out the following bolded line:

```
public class Hero : MonoBehaviour {
    ...
```

```
    void Awake() {
        S = this;  // Set the Singleton
//      fireDelegate += TempFire;
    }
    ...
}
```

Commenting out this line will make `fireDelegate` no longer call `TempFire()`. You may now delete the `TempFire()` method from the Hero class if you wish. If you press Play now, the Hero ship will not fire.

2. Open the Weapon C# script in MonoDevelop add the following bold code:

```
public class Weapon : MonoBehaviour {
    static public Transform   PROJECTILE_ANCHOR;

    [Header("These fields are set dynamically")]
    public WeaponType         _type = WeaponType.none;
    public WeaponDefinition   def;
    public GameObject         collar;
    public float              lastShotTime; // Time last shot was fired

    private Renderer          collarRend;

    void Start() {
        collar = transform.Find("Collar").gameObject;
        collarRend = collar.GetComponent<Renderer>();

        // Call SetType() for the default _type of WeaponType.none
        SetType( _type );                                   // a

        if (PROJECTILE_ANCHOR == null) {                    // b
            GameObject go = new GameObject("_ProjectileAnchor");
            PROJECTILE_ANCHOR = go.transform;
        }
```

```
        // Find the fireDelegate of the root GameObject
        GameObject rootGO =
transform.root.gameObject;                  // c
        if ( rootGO.GetComponent<Hero>() != null ) {                  // d
            rootGO.GetComponent<Hero>().fireDelegate += Fire;
        }
    }

    public WeaponType type {
        get {    return( _type );    }
        set {    SetType( value );   }
    }

    public void SetType( WeaponType wt ) {
        _type = wt;
        if (type == WeaponType.none) {                              // e
            this.gameObject.SetActive(false);
            return;
        } else {
            this.gameObject.SetActive(true);
        }
        def = Main.GetWeaponDefinition(_type);                      // f
        collarRend.material.color = def.color;
        lastShotTime = 0; // You can fire immediately after _type is
set. Ê// g
    }

    public void Fire() {
        // If this.gameObject is inactive, return
        if (!gameObject.activeInHierarchy) return;                  // h
        // If it hasn't been enough time between shots, return
        if (Time.time - lastShotTime < def.delayBetweenShots)
{       // i
            return;
        }
        Projectile p;
        Vector3 vel = Vector3.up * def.velocity;                    // j
```

```
        if (transform.up.y < 0) {
            vel.y = -vel.y;
        }
        switch (type) {                                    // k
            case WeaponType.blaster:
                p = MakeProjectile();
                p.rigid.velocity = vel;
                break;

            case WeaponType.spread:                                // l
                p = MakeProjectile();
                p.rigid.velocity = vel;
                p = MakeProjectile();
                p.transform.rotation = Quaternion.AngleAxis( 10,
Vector3.back );
                p.rigid.velocity = p.transform.rotation * vel;
                p = MakeProjectile();
                p.transform.rotation = Quaternion.AngleAxis(-
10, Vector3.back );
                p.rigid.velocity = p.transform.rotation * vel;
                break;

        }
    }

    public Projectile MakeProjectile() {                          // m
        GameObject go = Instantiate( def.projectilePrefab ) as
GameObject;
        if ( transform.parent.gameObject.tag == "Hero" ) {            //
n
            go.tag = "ProjectileHero";
            go.layer = LayerMask.NameToLayer("ProjectileHero");
        } else {
            go.tag = "ProjectileEnemy";
            go.layer = LayerMask.NameToLayer("ProjectileEnemy");
        }
        go.transform.position = collar.transform.position;
```

```
        go.transform.parent =
PROJECTILE_ANCHOR;                    // o
        Projectile p = go.GetComponent<Projectile>();
        p.type = type;
        lastShotTime = Time.time;                          // p
        return( p );

    }
}
```

a. When the Weapon GameObject starts, it calls `SetType()` with whatever WeaponType `_type` is set to. This ensures that either the Weapon disappears (if `_type` is `WeaponType.none`) or shows the correct collar color (if `_type` is `WeaponType.blaster` or `WeaponType.spread`).

b. `PROJECTILE_ANCHOR` is a static Transform created to act as a parent in the Hierarchy to all the Projectiles created by Weapon scripts. If `PROJECTILE_ANCHOR` is `null` (because it has not yet been created), this script creates a new GameObject named ***_ProjectileAnchor*** and assigns its transform to `PROJECTILE_ANCHOR`.

c. Weapons are always attached to other GameObjects (like _Hero). This finds the root GameObject of which this Weapon is a child.

d. If this root GameObject has a Hero script attached to it, then the `Fire` method of this Weapon is added to the `fireDelegate` delegate of that Hero class instance. If you wanted to add Weapons to Enemies, you could add a similar if statement her to check for an attached Enemy script. Even if a subclass of Enemy (e.g., Enemy_1, Enemy_2, etc.) were attached to `rootGO`, it would still return when `rootGO` was asked for the Enemy script component because that class would be a subclass of Enemy.

e. If `type` is `WeaponType.none`, this GameObject is disabled. When a GameObject is not active, it doesn't receive any MonoBehaviour method calls (e.g., Update(), LateUpdate(), FixedUpdate(), OnCollisionEnter(), etc.), it is not part of the physics simulation, and

it visually disappears from the scene. However, it is still possible to call functions and set variables on an inactive GameObject, so if something calls `SetType()` or sets the `type` property to `WeaponType.blaster` or `WeaponType.spread`, the GameObject will become active again.

f. Not only does `SetType()` set whether or not the GameObject is active, it also pulls the proper `WeaponDefinition` from Main, sets the color of the Collar, and resets `lastShotTime`.

g. Resetting `lastShotTime` to 0 allows this Weapon to be fired immediately (See `// i`).

h. `gameObject.activeInHierarchy` will be `false` if either this Weapon is inactive or if the _Hero GameObject (the root parent of this Weapon) is inactive or destroyed. In any case where `gameObject.activeInHierarchy` is `false`, this function will return, and the Weapon will not fire.

i. If the difference between the current time and the last time this Weapon was fired is less than the `delayBetweenShots` defined in the WeaponDefinition, this Weapon will not fire.

j. An initial velocity in the up direction is set, but if `transform.up.y` is `< 0` (which would be true for Enemy Weapons that are facing downward), the y component of `vel` is set to face downward as well.

k. This switch statement has options for each of the two WeaponTypes implemented in this chapter. The `WeaponType.blaster` generates a single Projectile by calling `MakeProjectile()`(which returns a reference to the Projectile class instance attached to the new projectile GameObject) and then assigns an velocity to its Rigidbody in the direction of `vel`.

l. If the `_type` is `WeaponType.spread`, then three different Projectiles are created. Two of them have their direction rotated 10 degrees around the `Vector3.back` axis (i.e., the -z axis that extends out of the screen towards you). Then, their Rigidbody.velocity is set to the

multiplication of that rotation by `vel`. When a Quaternion is multiplied by a Vector3, it rotates that Vector3, causing the resultant velocity to point in the direction that the Projectile is angled.

m. The `MakeProjectile()` method instantiates a clone of the prefab stored in the WeaponDefinition and returns a reference to the attached Projectile class instance.

n. Based on whether this was fired by the _Hero or an Enemy, the Projectile is given the proper tag and physics layer.

o. The Projectile GameObject's parent is set to be `PROJECTILE_ANCHOR`. This places it under _ProjectileAnchor in the Hierarchy pane, keeping the Hierarchy relatively clean to look at and avoiding the issue that we have with Enemy clones cluttering the pane. The `true` argument passed in tells go to maintain its current world position through the transition.

p. `lastShotTime` is set to the current time, preventing this Weapon from shooting for `def.delayBetweenShots` seconds.

3. Press Play, and you will see that the Weapon attached to _Hero disappears. This is because its WeaponType is WeaponType.none.

4. Select the Weapon attached to _Hero in the Hierarchy and set the Type of its Weapon (Script) component to *Blaster*. Then press Play, and you'll see that you can hold the space bar to fire blaster shots every 0.2 seconds (as defined in the `weaponDefinitions` array of the Main (Script) component of _MainCamera).

5. Select the Weapon attached to _Hero in the Hierarchy and set the Type of its Weapon (Script) component to *Spread*. Press Play, and you will see that the Weapon Collar is now blue, and three shots are fired every 0.4 seconds.

Revising the Enemy OnCollisionEnter Method

Now that our Weapons are firing shots that have the potential to do different amounts of damage (though they are currently set to do the same amount), we need to improve the `OnCollisionEnter()` method of the Enemy class.

1. Open the Enemy C# script in MonoDevelop and delete the `OnCollisionEnter()` method.

2. Replace the old `OnCollisionEnter()` method with this code.

```
public class Enemy : MonoBehaviour {
   ...
   public virtual void Move() { ... }

   void OnCollisionEnter( Collision coll ) {                    // a
      GameObject otherGO = coll.gameObject;
      switch (otherGO.tag) {
         case "ProjectileHero":                                 // b
            Projectile p = otherGO.GetComponent<Projectile>();
            // If this Enemy is off screen, don't damage it.
            if ( !bndCheck.isOnScreen ) {                       // c
               Destroy( otherGO );
               break;
            }

            // Hurt this Enemy
            // Get the damage amount from the Main WEAP_DICT.
            health -=
Main.GetWeaponDefinition(p.type).damageOnHit;
            if (health <= 0) {                                  // d
               // Destroy this Enemy
               Destroy(this.gameObject);
            }
            Destroy( otherGO );                                 // e
            break;

         default:
```

```
            print( "Enemy hit by non-ProjectileHero: " +
otherGO.name ); Image// f
            break;

        }
    }
}
```

a. Make sure you're replacing the old `OnCollisionEnter()` method entirely.

b. If the GameObject that hit this Enemy has the ProjectileHero tag, it should damage this Enemy. If it has any other tag, it will be handled by the `default` case (`// f`).

c. If this Enemy is not on screen, the Projectile GameObject that hit it is destroyed, and break; is called, which exits the switch statement without completing any of the remaining code in the case "ProjectileHero".

d. If the Enemy's health is decreased to below 0, then this Enemy is destroyed. With a default Enemy health of 10 and blaster damageOnHit of 1, this will take 10 shots.

e. The Projectile GameObject is destroyed.

f. If somehow a GameObject tagged something other than a ProjectileHero hits this Enemy, this will post a message about it to the Console pane.

3. Before pressing Play on the scene, it will be helpful to switch from Enemy_3s being spawned back to spawning regular Enemies. Select *_MainCamera* in the Hierarchy and set *Element 0* of the `prefabEnemies` array of the *Main (Script)* component to be the Enemy_0 prefab.

Now when you play the scene, it is possible to destroy the enemy, but each one takes 10 shots to take down, and it's difficult to tell that

they’re being damaged.

Showing Enemy Damage

In order to show that an Enemy is being damaged, we will add code that makes the Enemy blink red for a couple of frames every time it is hit. However, to do so, we’re going to need to have access to all the materials of all the children of each enemy. This seems like something that might be useful in several different games, so we’ll make this part of a new *Utils* C# class that we will fill with reusable game code.

Creating the Reusable Utils Script

We will be using the Utils class throughout the rest of this book. The Utils class is going to be almost entirely composed of static functions so that the functions can easily be called from anywhere in your code.

Create a new C# script named *Utils* and place it in the __Scripts folder. Open Utils in MonoDevelop and enter the following code:

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

public class Utils : MonoBehaviour {

//============================= Materials Functions
============================\\

    // Returns a list of all Materials on this GameObject and its
children
    static public Material[] GetAllMaterials( GameObject go )
{              // a
        List<Material> mats = new List<Material>();
```

```
        Renderer rend = go.GetComponent<Renderer>();
        if (rend != null) {
            mats.Add(rend.material);
        }
        // Recursively iterates over all children of go.transform
        foreach( Transform t in go.transform ) {                    // b
            mats.AddRange( GetAllMaterials( t.gameObject ) );
        }
        return( mats.ToArray() );                                   // c
    }

}
```

a. `GetAllMaterials()` is a recursive function, so it can call itself multiple times. It will grab the material of the Renderer of `go` (the GameObject passed in) and then call `GetAllMaterials()` on all the children of `go`. As a static public function, this can be called anywhere in this project via `Utils.GetAllMaterials()`. For more information on recursive functions, see Chapter 24, "Functions and Parameters."

b. The transform of a GameObject can be iterated over using a foreach loop. This will return each direct child of the transform one at-a-time. This loop then recursively calls `GetAllMaterials()` on the GameObject of each child transform and adds the array of Materials returned to the `mats` array.

c. When finished, the `mats` array is returned, either passing it upward to be included in the `mats` array of the parent GameObject or passing it back to the original calling function.

Using GetAllMaterials to Make the Enemy Blink Red

Now we need to utilize the GetAllMaterials static method of Utils

1. Add the following bold code to the `Enemy` class:

```
public class Enemy : MonoBehaviour {
    ...
    public int        score = 100; // Points earned for destroying this
    public float      showDamageDuration = 0.1f; // # seconds to
show damage Image// a

    [Header("These fields are set dynamically")]
    public Color[]    originalColors;
    public Material[] materials;// All the Materials of this & its
children
    public bool       showingDamage = false;
    public float      damageDoneTime; // Time to stop showing
damage
    public bool       notifiedOfDestruction = false; // Will be used
later


    protected BoundsCheck bndCheck;

    void Awake() {
        bndCheck = GetComponent<BoundsCheck>();
        // Get materials and colors for this GameObject and its
children
        materials = Utils.GetAllMaterials( gameObject );              //
b
        originalColors = new Color[materials.Length];
        for (int i=0; i<materials.Length; i++) {
            originalColors[i] = materials[i].color;
        }
    }

    ...

    void Update() {
        Move();

        if ( showingDamage && Time.time > damageDoneTime )
```

```
{          // c
           UnShowDamage();
       }

       if ( bndCheck != null && bndCheck.offDown ) {
          // We're off the bottom, so destroy this GameObject
          Destroy( gameObject );
       }
    }

    ...

    void OnCollisionEnter( Collision coll ) {
       GameObject otherGO = coll.gameObject;
       switch (otherGO.tag) {
          case "ProjectileHero":
             ...
             // Hurt this Enemy
             ShowDamage();                                    // d
             // Get the damage amount from the Main WEAP_DICT.
             ...
       }
    }

    void ShowDamage() {                                       // e
       foreach (Material m in materials) {
          m.color = Color.red;
       }
       showingDamage = true;
       damageDoneTime = Time.time + showDamageDuration;
    }

    void UnShowDamage() {                                     // f
       for ( int i=0; i<materials.Length; i++ ) {
          materials[i].color = originalColors[i];
       }
       showingDamage = false;
```

```
    }
}
```

a. Add all the new bolded fields at the top.

b. The materials array is filled using our new `Utils.GetAllMaterials()` method. Then, code here iterates through all the materials and stores their original color. Though all of our Enemy GameObjects are currently white, this allows us to set whatever color we want on them, color each one red when the Enemy is damaged, and then return them to their original color.

Importantly, this call to `Utils.GetAllMaterials()` is made in the `Awake()` method, and the result is cached in `materials`. This ensures that it only happens once for each Enemy. `Utils.GetAllMaterials()` includes several calls to `GetComponent<>()`, which is a somewhat slow function that can take processing time and decrease performance.

c. If the Enemy is currently showing damage (i.e., it's red) and the current time is later than `damageDoneTime`, `UnShowDamage()` is called.

d. A call to `ShowDamage()` is added to the section of `OnCollisionEnter()` that damages the Enemy.

e. `ShowDamage()` turns all materials in the `materials` array red, sets `showingDamage` to `true`, and sets the time that it should stop showing damage.

f. `UnShowDamage()` turns all materials in the `materials` array back to their original color and sets `showingDamage` to `false`.

Now, when the enemy is struck by a projectile from the hero, it will turn entirely red for `damageDoneTime` seconds by setting the color of all materials to red. After that time, the Enemy ship and its children revert to their original colors.

2. Press Play and test your game. Now it's possible to see that the player is damaging the ship, but it still takes many hits to destroy one. Let's make some power-ups that will increase the power and number of the player's weapons.

3. Did you remember to save your project? Always save your project frequently.

Adding Power-Ups and Boosting Weapons

In this section, we will create three power-ups for the game:

• **blaster [B]:** If the player weapon type is not blaster, switches to blaster and resets to 1 gun. If the player weapon type is already blaster, increases the number of guns.

• **spread [S]:** If the player weapon type is not spread, switches to spread and resets to 1 gun. If the player weapon type is already spread, increases the number of guns.

• **shield [O]:** Increases the player's shieldLevel by 1.

Artwork for Power-Ups

The power-ups will be made of a letter rendered as 3D text with a spinning cube behind it. (You can see some of them in Figure 30.1 at the beginning of the previous chapter.) To make the power-ups, complete these steps:

1. Create a new 3D text (*GameObject* > *3D Object* > *3D Text* from the menu bar). Name it *PowerUp* and give it a *Cube* child and assign both the following settings:



2. Select the PowerUp.

3. Set the Text Mesh component properties of PowerUp to those shown in Figure 31.3.

Figure 31.3. Settings for PowerUp and its child Cube prior to attaching any scripts



4. Add a Rigidbody component to PowerUp (*Component* > *Physics* > *Rigidbody*) and set it as shown in the Figure 31.3.

5. Set both the tag and the physics layer of PowerUp to *PowerUp*. When asked, click *Yes, change children*.

6. Create a custom material for the PowerUp cube, as follows:

a. Create a new Material named *Mat_PowerUp* inside the _Materials folder.

b. Drag it onto the Cube that is a child of PowerUp.

c. Select the Cube that is a child of PowerUp.

d. Set the Shader of Mat PowerUp to *ProtoTools* > *UnlitAlpha*.

e. Click the *Select* button at the bottom right of the texture box for Mat_PowerUp and choose the texture named *PowerUp* from the Assets tab. You will probably need to open the disclosure triangle in the bottom-left corner of the Mat_PowerUp component in the Inspector to see the texture for Mat_PowerUp.

f. Set the main color of Mat_PowerUp to cyan (a light blue that is RGBA:[0,255,255,255]), and you can see how the PowerUp will look when colored.

g. Set the Box Collider of Cube to be a trigger (check the box next to *Is Trigger*).

Double-check that all the settings for PowerUp and its child Cube match those in Figure 31.3 and save your scene.

PowerUp Code

The power-up code is next:

1. Attach a BoundsCheck script to the PowerUp GameObject in the Hierarchy. Set `radius` to 1 and `keepOnScreen` to `false` (unchecked).

2. Create a new C# script named *PowerUp* in the __Scripts folder.

3. Attach the PowerUp script to the PowerUp GameObject in the Hierarchy.

4. Open the PowerUp script in MonoDevelop and enter the following code:

```
using UnityEngine;
using System.Collections;

public class PowerUp : MonoBehaviour {
    [Header("Set in the Unity Inspector")]
    // This is an unusual but handy use of Vector2s. x holds a min value
    //   and y a max value for a Random.Range() that will be called later
    public Vector2          rotMinMax = new Vector2(15,90);
    public Vector2          driftMinMax = new Vector2(.25f,2);
    public float            lifeTime = 6f;   // Seconds the PowerUp exist
    public float            fadeTime = 4f;   // Seconds it will then fade

    [Header("These fields are set dynamically")]
    public WeaponType       type;           // The type of the PowerUp
    public GameObject       cube;           // Reference to the Cube
```

```
child
   public TextMesh         letter;         // Reference to the
TextMesh
   public Vector3          rotPerSecond;    // Euler rotation speed
   public float            birthTime;

   private Rigidbody       rigid;
   private BoundsCheck     bndCheck;
   private Renderer        cubeRend;

   void Awake() {
      // Find the Cube reference
      cube = transform.Find("Cube").gameObject;
      // Find the TextMesh and other components
      letter = GetComponent<TextMesh>();
      rigid = GetComponent<Rigidbody>();
      bndCheck = GetComponent<BoundsCheck>();
      cubeRend = cube.GetComponent<Renderer>();

      // Set a random velocity
      Vector3 vel = Random.onUnitSphere; // Get Random XYZ
velocity
      // Random.onUnitSphere gives you a vector point that is
somewhere on
      //   the surface of the sphere with a radius of 1m around the
origin
      vel.z = 0; // Flatten the vel to the XY plane
      vel.Normalize(); // Normalizing a Vector3 makes it length 1m

      vel *= Random.Range(driftMinMax.x,
driftMinMax.y);            // a
      rigid.velocity = vel;

      // Set the rotation of this GameObject to R:[0,0,0]
      transform.rotation = Quaternion.identity;
      // Quaternion.identity is equal to no rotation.
```

```
        // Set up the rotPerSecond for the Cube child using
rotMinMax x & y
        rotPerSecond = new Vector3(
Random.Range(rotMinMax.x,rotMinMax.y),
            Random.Range(rotMinMax.x,rotMinMax.y),
            Random.Range(rotMinMax.x,rotMinMax.y) );

        birthTime = Time.time;
    }

    void Update () {
        cube.transform.rotation = Quaternion.Euler(
rotPerSecond*Time.time ) Image; // b

        // Fade out the PowerUp over time
        // Given the default values, a PowerUp will exist for 10
seconds
        //   and then fade out over 4 seconds.
        float u = (Time.time - (birthTime+lifeTime)) / fadeTime;
        // For lifeTime seconds, u will be <= 0. Then it will transition
to
        // 1 over fadeTime seconds.
        // If u >= 1, destroy this PowerUp
        if (u >= 1) {
            Destroy( this.gameObject );
            return;
        }
        // Use u to determine the alpha value of the Cube & Letter
        if (u>0) {
            Color c = cubeRend.material.color;
            c.a = 1f-u;
            cubeRend.material.color = c;
            // Fade the Letter too, just not as much
            c = letter.color;
            c.a = 1f - (u*0.5f);
            letter.color = c;
        }
```

```
        if (!bndCheck.isOnScreen) {
            // If the PowerUp has drifted entirely off screen, destroy it
            Destroy( gameObject );
        }
    }

    public void SetType( WeaponType wt ) {
        // Grab the WeaponDefinition from Main
        WeaponDefinition def = Main.GetWeaponDefinition( wt );
        // Set the color of the Cube child
        cubeRend.material.color = def.color;
        //letter.color = def.color; // We could colorize the letter too
        letter.text = def.letter; // Set the letter that is shown
        type = wt; // Finally actually set the type
    }

    public void AbsorbedBy( GameObject target ) {
        // This function is called by the Hero class when a PowerUp
is
        // collected
        // We could tween into the target and shrink in size,
        //   but for now, just destroy this.gameObject
        Destroy( this.gameObject );
    }

}
```

a. Sets the velocity length to something between the x and y values of `driftMinMax`.

b. Manually rotate the Cube child every Update(). Multiplying `rotPerSecond` by Time.time causes the rotation to be time-based.

5. Press Play, you should see the power-up drifting and rotating. If you fly the hero into the power-up, you will get the console message

“Triggered by: PowerUp,” letting you know that the Trigger Collider on the PowerUp cube is working properly.

6. Drag the PowerUp GameObject from the Hierarchy into the _Prefabs folder in the Project pane to make it into a prefab.

Enabling the Hero to Collect PowerUps

Next, we need to enable the Hero to collect PowerUps. First we’ll just manage the collection, then we’ll modify Hero to upgrade and change weapons in response to PowerUps.

1. Make the following changes to the Hero C# script to enable the hero to collide with and collect power-ups:

```
public class Hero : MonoBehaviour {
    ...
    void OnTriggerEnter(Collider other) {
        ...

        if (go.tag == "Enemy") {
            // If the shield was triggered by an enemy
            // Decrease the level of the shield by 1
            shieldLevel--;
            // Destroy the enemy
            Destroy(go);                                    // 4
        } else if (go.tag == "PowerUp") {
            // If the shield was triggerd by a PowerUp
            AbsorbPowerUp(go);
        } else {
            print("Triggered by: "+go.name);                    // g
        }
    }

    public void AbsorbPowerUp( GameObject go ) {
        PowerUp pu = go.GetComponent<PowerUp>();
        switch (pu.type) {
```

```
            // Leave this switch block empty for now.

        }
        pu.AbsorbedBy( this.gameObject );
    }

    public float shieldLevel { ... }
}
```

2. Now, when you press play, you'll see that the Hero can run into the PowerUp and absorb.

Before we can make absorbing a PowerUp actually do something, we need to do a little more Weapons set up.

3. Add the weapons array to the top of the Hero script as shown in bold code here.

```
public class Hero : MonoBehaviour {
    ...
    public float           projectileSpeed = 40;
    public Weapon[]        weapons;                          // a

    [Header("These fields are set dynamically")]
    ...
}
```

a. The `weapons` array will store a reference to each of the five Weapon GameObjects that we will make children of _Hero in the next section.

Expanding Weapon Options

Now that the code is set up, you need to make a couple changes to _Hero in Unity.

1. Open the disclosure triangle next to the GameObject _Hero in the Hierarchy.

2. Select the Weapon child of _Hero. Press Command-D (or Control+D on Windows) four times to make four duplicates of Weapon.[1] The duplicates should all still be children of _Hero.

[1] If the keyboard command doesn't work, you can choose *Edit* > *Duplicate* from the Menu Bar or right-click the original Weapon in the Hierarchy and choose Duplicate.

3. Rename the four weapons *Weapon_0* through *Weapon_4* and configure their transforms as follows:



4. Select _Hero and open the disclosure triangle for the `weapons` field in the Hero (Script) component Inspector.

5. Set the Size of `weapons` to 5 and assign Weapon_0 through Weapon_4 to the five Weapon slots in order (either by dragging them in from the Hierarchy or by clicking the target to the right of the Weapon slot and selecting each Weapon_# from the Scene tab). Figure 31.4 shows the final setup.

Figure 31.4. The _Hero ship showing five Weapons as children and assigned to the weapons field



6. Let's make absorbing a PowerUp actually do something. To do so, we'll need to make some changes to the Hero script.

7. Open the Hero script and add the GetEmptyWeaponSlot() and ClearWeapons() methods to the bottom of the Hero class.

```
public class Hero : MonoBehaviour {
    ...
```

```
    public float shieldLevel {
        ...
    }

    Weapon GetEmptyWeaponSlot() {
        for (int i=0; i<weapons.Length; i++) {
            if ( weapons[i].type == WeaponType.none ) {
                return( weapons[i] );
            }
        }
        return( null );
    }

    void ClearWeapons() {
        foreach (Weapon w in weapons) {
            w.SetType(WeaponType.none);
        }
    }
}
```

8. Fill the switch block of the AbsorbPowerUp() method that we left empty before with the bold code below.

```
public class Hero : MonoBehaviour {
    ...
    public void AbsorbPowerUp( GameObject go ) {
        PowerUp pu = go.GetComponent<PowerUp>();
        switch (pu.type) {
            case WeaponType.shield:                                  // a
                shieldLevel++;
                break;

            default:                                                 // b
                if (pu.type == weapons[0].type) { // If it is the same
type  Image// c
                    Weapon w = GetEmptyWeaponSlot();
                    if (w != null) {
```

```
                // Set it to pu.type
                w.SetType(pu.type);
            }
        } else { // If this is a different weapon type        // d
            ClearWeapons();
            weapons[0].SetType(pu.type);
        }
        break;
    }
    pu.AbsorbedBy( this.gameObject );
  }
  ...
}
```

a. If the PowerUp has the WeaponType shield, it increases the shield level by 1.

b. Any other PowerUp WeaponType will be a weapon, so that is the default state.

c. If the PowerUp is the same WeaponType as the existing weapons, we search for an unused weapon slot and try to set that empty slot to the same weapon type. If all five slots are already in use, nothing different happens.

d. If the PowerUp is a different WeaponType, then all weapon slots are cleared, and Weapon_0 is set to the new WeaponType that was picked up.

9. To test this, select the PowerUp in the Hierarchy, and inside the PowerUp (Script) component of the Inspector, set the `type` (under the heading “These fields are set dynamically”) to `Spread`. Normally, the type is set dynamically, but we can set it ourselves now for testing.

10. Press Play, and you’ll start with five blasters. When you run into the PowerUp, it will swap those for a single spread gun. Because we set the type of the PowerUp manually, it doesn’t show the right letter.

You can also try testing it with a type of shield, and you'll see that the shield level increases when you run over the PowerUp.

Managing Race Conditions

Now, we're going to break something to make a point. Bear with me; this is important, and you'll probably run into something like it in the future.

1. Add the following bolded lines to the Awake() method of the Hero script:

```
public class Hero : MonoBehaviour {
    ...
    void Awake() {
        S = this;  // Set the Singleton
//        fireDelegate += TempFire;          // c

        // Reset the weapons to start _Hero with 1 blaster
        ClearWeapons();
        weapons[0].SetType(WeaponType.blaster);
    }
    ...
}
```

2. Press Play, and you should see something like the following error message:

NullReferenceException: Object reference not set to an instance of an object

Weapon.SetType (WeaponType wt) (at Assets/__Scripts/Weapon.cs:82)

Hero.Awake () (at Assets/__Scripts/Hero.cs:36)

This tells us that something we are trying to use is null and that it encountered this error on this line of Weapon.cs (line 82 in my

implementation, but your line number may be different).

```
collarRend.material.color = def.color;
```

The error message also tells us that this line of Weapon.cs was reached via a function call on this line of Hero.cs (line 36 for me, but yours may vary).

```
weapons[0].SetType(WeaponType.blaster);
```

So, tracing this back, it looks like the Awake() method of Hero is calling the SetType() method of the 0^{th} Weapon in the weapons array. The SetType() method of Weapon is trying to set the color of collarRend, but collarRend is null, so this is throwing a null reference error.

Look at the Start() method in Weapon. That is where collarRend is set. However, the Awake() method of Hero is going to happen before the Start() method on Warrior meaning that we are trying to use collarRend before it is set! Fixing this is going to involve some additional steps.

3. In the Hero script, change the name of the Awake() method to Start().

If you press play, it may look like this fixed everything, but that's not actually the case. What we've done is possibly fix things, because it's possible that Weapon.Start() will execute before Hero.Start(), but it's also still possible that Hero.Start() will execute first. We need to be sure.

4. Open the Script Execution Order Inspector by choosing *Edit* > *Project Settings* > *Script Execution Order* from the Menu Bar.

a. Click the + button that is circled in image 1 of Figure 31.5 and choose the *Weapon* script from the pop-up menu. This will create a row in the table for Weapon with a value of 100. The 100 number represents the execution order of Weapon versus the other scripts,

which all run at *Default Time*. If the number is higher (as it is now), all Weapon scripts will execute after other scripts, meaning that the Weapon.Start() method will execute after the Hero.Start() or any other script.

Figure 31.5. The Script Execution Order Inspector showing manipulation of the Weapon script's execution order.



b. Click the *Apply* button (his will lock-in the execution order) and press Play.

c. You will see that we still get a null reference exception. Instead of Weapon happening last, we need it to happen first.

d. Open the Script Execution Order Inspector again, and use the thumb on the left side of the Weapon row (that the cursor is touching in image 2 of Figure 31.5) to drag the Weapon row above Default Time. This will also change the number on that row from 100 to -100 (as shown in image 2 of Figure 31.5).

e. Click Apply and press Play again, and this time, you should not get any errors.

Race conditions and script execution order are important but subtle things to keep in mind when you're working on your own projects.

Making Enemies Drop Power-Ups

Getting back to the power-ups. Let's make enemies have the potential to drop a random power-up when they are destroyed. This gives the player a lot more incentive to try to destroy enemies rather than just avoid them, and it gives the player a path to improving her ship.

When an Enemy is destroyed, it will notify the Main singleton, and then the Main singleton will instantiate a new PowerUp. This may

seem like a somewhat roundabout way to do this, but in general, it is best to limit the number of different classes that can instantiate new GameObjects into your scene. If fewer scripts are responsible an element (like instantiation), then it'll be easier to debug if you see that something is going wrong with that element.

1. We'll start by making the Main class able to instantiate new PowerUps. Add the following bolded code to the Main script.

```
public class Main : MonoBehaviour {
    ...
    public WeaponDefinition[]    weaponDefinitions;
    public GameObject            prefabPowerUp;                    // a
    public WeaponType[]          powerUpFrequency = new
WeaponType[] {  // b
                                 WeaponType.blaster, WeaponType.blaster,
                                 WeaponType.spread, WeaponType.shield
};

    private BoundsCheck     bndCheck;

    public void ShipDestroyed( Enemy e ) {                         // c
        // Potentially generate a PowerUp
        if (Random.value <= e.powerUpDropChance)
{                       // d
            // Random.value generates a value between 0 & 1
(though never ==
            // 1)

            // If the e.powerUpDropChance is 0.50f, a PowerUp will
be
            // generated

            //   50% of the time. For testing, it's now set to 1f.

            // Choose which PowerUp to pick
            // Pick one from the possibilities in powerUpFrequency
```

```
        int ndx =
Random.Range(0,powerUpFrequency.Length);        // e
        WeaponType puType = powerUpFrequency[ndx];

        // Spawn a PowerUp
        GameObject go = Instantiate( prefabPowerUp ) as
GameObject;
        PowerUp pu = go.GetComponent<PowerUp>();
        // Set it to the proper WeaponType
        pu.SetType( puType );                          // f

        // Set it to the position of the destroyed ship
        pu.transform.position = e.transform.position;
      }
   }

   void Awake() { ... }
   ...
}
```

a. This will hold the prefab for all PowerUps.

b. This `powerUpFrequency` array of WeaponTypes determines how often each type of PowerUp will be created. By default, it has two blasters, one spread, and one shield, so the blaster power-up will be twice as common.

c. The ShipDestroyed() method it called by an Enemy ship whenever it is destroyed. It will sometimes create a power-up in place of the destroyed ship.

d. Each type of ship has a powerUpDropChance, which is a number between 0 and 1. Random.value generates a random float between 0 (inclusive) and 1 (inclusive). (Because Random.value is inclusive of both 0 and 1, the number could also be either 0 or 1). If that number is less than or equal to the powerUpDropChance, a PowerUp is instantiated. The drop chance is part of the Enemy class so that various enemies can have higher or lower chances of

dropping a PowerUp (e.g., Enemy_0 cold rarely drop one, while Enemy_4 could always drop one).

e. This line makes use of the powerUpFrequency array. When Random.Range() is called with two integer values, it will choose a number between the first number (inclusive) and the second number (exclusive), for example, Random.Range(0,4) would generate an int with a value of 0, 1, 2, or 3. This is very useful for choosing a random entry in an array, as we're doing on this line.

f. Once a power-up type has been selected, the SetType() method is called on the instantiated PowerUp, and the PowerUp then handles coloring itself, setting its _type, and setting the correct letter.

2. Now, add the bolded code that follows to the Enemy script:

```
public class Enemy : MonoBehaviour {
    ...
    public float      showDamageDuration = 0.1f; // # seconds to show damage
    public float      powerUpDropChance = 1f;  // Chance to drop a power-up  Image// a

    [Header("These fields are set dynamically")]
    ...

    void OnCollisionEnter( Collision coll ) {
        GameObject otherGO = coll.gameObject;
        switch (otherGO.tag) {
            case "ProjectileHero":
                ...
                // Hurt this Enemy
                ...
                if (health <= 0) {
                    // Tell the Main singleton that this ship was destroyed  Image// b
                    if (!notifiedOfDestruction) Main.S.ShipDestroyed( this );
```

```
                notifiedOfDestruction = true;
                // Destroy this Enemy
                Destroy(this.gameObject);
            }
            ...
            break;
        ...
    }
  }
}
```

a. `powerUpDropChance` determines how likely this Enemy is to drop a PowerUp when it is destroyed. 0 is never, while 1 is always.

b. Immediately before this Enemy is destroyed, it notifies the Main singleton by calling ShipDestroyed(). This only happens once for each ship, which is enforced by the `notifiedOfDestruction` bool.

3. Select _MainCamera in the Hierarchy and assign the PowerUp prefab from the _Prefabs folder in the Project pane to the `prefabPowerUp` field of the *Main (Script)* component of _MainCamera.

3. Before this code will work, you need to select _ MainCamera in the Hierarchy and assign the PowerUp prefab from the _Prefabs folder in the Project pane to the `prefabPowerUp` field of the *Main (Script)* component of _MainCamera.

4. Select the PowerUp instance in the Hierarchy and delete it (you don't need it because we have a PowerUp prefab).

5. `powerUpFrequency` should already be set in the Inspector, but just in case, Figure 31.6 shows the correct settings. Note that enums appear in the Unity Inspector as pop-up menus.

Figure 31.6. prefabPowerUp and powerUpFrequency on the Main (Script) component of _MainCamera



6. Now play the scene and destroy some enemies. They should drop power-ups that will now improve your ship!

You should notice over time that the blaster [B] power-up is more common than spread [S] or shield [O]. This is because there are two occurrences of blaster in `powerUpFrequency` and only one each of spread and shield. By adjusting the relative numbers of occurrences of each of these in `powerUpFrequency`, you can determine the chance that each will be chosen relative to the others. This same trick could also be used to set the frequency of different types of enemies spawning by assigning some enemies to the `prefabEnemies` array more times than other enemy types.

Enemy_4—A More Complex Enemy

As somewhat of a boss type, Enemy_4 will have more health than other enemy types and will have destructible parts (rather than all the parts being destroyed at the same time). It will also stay on screen, moving from one position to another, until the player destroys it completely.

Collider Modifications

Before getting into code issues, you need to make a few adjustments to the colliders of Enemy_4.

1. First, drag an instance of Enemy_4 into the Hierarchy and make sure that it's positioned away from other GameObjects in the scene (it should default to the P:[20,0,0] that you set it at earlier).

2. Open the disclosure triangle next to Enemy_4 in the Hierarchy and select the Fuselage child.

3. Remove the Sphere Collider component from Fuselage by clicking the gear in the top-right corner of the Sphere Collider component in the Inspector and selecting Remove Component.

4. Add a Capsule Collider to the Fuselage by selecting *Component* > *Physics* > *Capsule Collider* from the menu bar. Set the Capsule Collider as follows in the Fuselage Inspector:



Feel free to play with the values somewhat to see how they affect things. As you can see, the Capsule Collider is a much better approximation of Fuselage than the Sphere Collider was.

5. Now, select the Wing_L child of Enemy_4 in the Hierarchy and replace its Sphere Collider with a Capsule Collider as well. The settings for this Capsule Collider are as follows:



The Direction setting chooses which is the long axis of the capsule. This is determined in local coordinates, so the height of 1 along the X-axis is multiplied by the scale of 5 in the X dimension. The radius of 0.5 is multiplied by the maximum of either the Y or Z scales, so the actual radius of the capsule is 0.5 due to the Y scale of 1. You can see that the capsule does not perfectly match the wing, but again it is a much better approximation than a sphere.

6. Select Wing_R, replace its collider with a Capsule Collider, and give that collider the same settings as used on Wing_L.

7. Once all of these changes have been made, select Enemy_4 in the Hierarchy and add a BoundsCheck (Script) component to Enemy_4 by clicking the Add Component button in the Inspector and choosing Add Component > Scripts > BoundsCheck.

8. In the BoundsCheck (Script) component, set `radius` = 3.5 and `keepOnScreen` = `false`.

9. Click the *Apply* button to the right of the word *Prefab* at the top of the Inspector pane. This will apply these changes to the Enemy_4 prefab in the Project pane.

10. To double-check that this worked successfully, drag a second instance of the Enemy_4 prefab into the Hierarchy pane and check to make sure that the colliders all look correct.

11. Once this is done, delete both instances of Enemy_4 from the Hierarchy pane.

This same Capsule Collider strategy could also be applied to Enemy_3 if you want.

12. Save your scene! Have you been remembering?

Movement of Enemy_4

Enemy_4 will start in the standard position off the top of the screen, pick a random point on screen and move to it over time using a linear interpolation. Once it has reached the point, it will rest for a moment and then move to another.

1. Open the Enemy_4 script and input this code:

```
using UnityEngine;
using System.Collections;

/// <summary>
/// Enemy_4 will start offscreen and then pick a random point on screen to
///   move to. Once it has arrived, it will pick another random point and
///   continue until the player has shot it down.
/// </summary>
public class Enemy_4 : Enemy {

    private Vector3       p0, p1;    // The two points to interpolate
    private float         timeStart;  // Birth time for this Enemy_4
    private float         duration = 4;  // Duration of movement
```

```
   void Start () {
      // There is already an initial position chosen by
Main.SpawnEnemy()
      //   so add it to points as the initial p0 & p1
      p0 = p1 = pos;                                    // a

      InitMovement();
   }

   void InitMovement() {                                 // b
      p0 = p1;   // Set p0 to the old p1
      // Assign a new on-screen location to p1
      float widMinRad = bndCheck.camWidth - bndCheck.radius;
      float hgtMinRad = bndCheck.camHeight - bndCheck.radius;
      p1.x = Random.Range( -widMinRad, widMinRad );
      p1.y = Random.Range( -hgtMinRad, hgtMinRad );

      // Reset the time
      timeStart = Time.time;
   }

   public override void Move () {                        // c
      // This completely overrides Enemy.Move() with a linear
      // interpolation
      float u = (Time.time-timeStart)/duration;

      if (u>=1) {
         InitMovement();
         u=0;
      }

      u = 1 - Mathf.Pow( 1-u, 2 );    // Apply Ease Out easing to
u   // d

      pos = (1-u)*p0 + u*p1; // Simple linear interpolation         //
e
```

```
    }
}
```

a. Enemy_4 interpolates from `p0` to `p1` (i.e., moves smoothly from `p0` to `p1`). The `Main.SpawnEnemy()` script gives this instance a position just above the top of the screen which is assigned here to both `p0` and `p1`. Then `InitMovement()` is called.

b. `InitMovement()` first stores the current `p1` location in `p0` (because Enemy_4 should be at location `p1` any time `InitMovement()` is called). Next, a new `p1` location is chosen that uses information from the BoundsCheck component to guarantee it is on screen.

c. This `Move()` method completely overrides the inherited `Enemy.Move()` method. It interpolates from `p0` to `p1` in `duration` seconds (4 seconds by default). The float `u` increases from 0 to 1 with time as this interpolation happens, and when `u` is >= 1, `InitMovement()` is called to set up a new interpolation.

d. This line applies easing to the `u` value, causing the ship to move in a non-linear fashion. With this “Ease Out” easing, the ship will begin its movement quickly and then slow as it approaches `p1`.

e. This line performs a simple linear interpolation from `p0` to `p1`.

To learn a lot more about both interpolation and easing, read the section on Interpolation in Appendix B, “Useful Concepts.”

2. Select _MainCamera in the Hierarchy. Assign the Enemy_4 prefab in the _Prefabs folder of the Project pane to Element 0 of the prefabEnemies array in the Main (Script) Inspector.

3. Delete the Enemy_4 instance from the Hierarchy.

4. Press Play. You can see that the spawned Enemy_4s stay on screen until you destroy them. However, they’re currently just as simple to take down as any of the other enemies.

Splitting Enemy_4 into Multiple Parts

Now we'll break the Enemy_4 ship into four different parts with the central Cockpit protected by the others.

1. Open the Enemy_4 C# script and start by adding a new serializable class named `Part` to the top of Enemy_4.cs. Also be sure to add a `Part[]` array to the Enemy_4 class named `parts`. Also add the bolded lines that follow to the Start() script of Enemy_4.

```
using UnityEngine;
using System.Collections;

/// <summary>
/// Part is another serializable data storage class just like
WeaponDefinition
/// </summary>
[System.Serializable]
public class Part {
   // These three fields need to be defined in the Inspector pane
   public string      name;        // The name of this part
   public float       health;      // The amount of health this part
has
   public string[]    protectedBy;  // The other parts that protect
this

   // These two fields are set automatically in Start().
   // Caching like this makes it faster and easier to find these
later
   [HideInInspector]    // Makes the field below not appear in the
                        // Inspector
   public GameObject   go;          // The GameObject of this part
   [HideInInspector]
   public Material     mat;         // The Material to show damage
}

...
```

```
public class Enemy_4 : Enemy {
   [Header("Set in the Unity Inspector")]
   public Part[]         parts;   // The array of ship Parts

   private Vector3       p0, p1;    // The two points to interpolate
   private float         timeStart;        // Birth time for this Enemy_4
   private float         duration = 4;     // Duration of movement

   void Start () {
      // There is already an initial position chosen by
Main.SpawnEnemy()
      //   so add it to points as the initial p0 & p1
      p0 = p1 = pos;                                    // a

      InitMovement();

      // Cache GameObject & Material of each Part in parts
      Transform t;
      foreach(Part prt in parts) {
         t = transform.Find(prt.name);
         if (t != null) {
            prt.go = t.gameObject;
            prt.mat = prt.go.GetComponent<Renderer>().material;
         }
      }
   }
   ...
}
```

The serializable `Part` class will store individual information about the four parts of Enemy_4: Cockpit, Fuselage, Wing_L, and Wing_R.

2. Switch back to Unity and do the following:

a. Select the Enemy_4 prefab in the Project pane.

b. Expand the disclosure triangle next to `parts` in the *Enemy_4 (Script)* Inspector.

c. Enter the settings shown in Figure 31.7.

Figure 31.7. The settings for the `parts` array of Enemy_4



As you can see above, each part has 10 health, and there is a tree of protection. Cockpit is protected by Fuselage, and Fuselage is protected by both Wing_L and Wing_R.

3. Now, switch back to MonoDevelop and add the following methods to the end of the Enemy_4 class to make this protection work:

```
public class Enemy_4 : Enemy {
    ...

    public override void Move () {
        ...
    }

    // These two functions find a Part in parts based on name or
GameObject
    Part FindPart(string n) {                                   // a
        foreach( Part prt in parts ) {
            if (prt.name == n) {
                return( prt );
            }
        }
        return( null );
    }
    Part FindPart(GameObject go) {                                  // b
        foreach( Part prt in parts ) {
            if (prt.go == go) {
                return( prt );
            }
```

```
        }
        return( null );
    }

    // These functions return true if the Part has been destroyed
    bool Destroyed(GameObject go) {                              // c
        return( Destroyed( FindPart(go) ) );
    }
    bool Destroyed(string n) {
        return( Destroyed( FindPart(n) ) );
    }
    bool Destroyed(Part prt) {
        if (prt == null) { // If no real ph was passed in
            return(true); // Return true (meaning yes, it was
destroyed)
        }
        // Returns the result of the comparison: prt.health <= 0
        // If prt.health is 0 or less, returns true (yes, it was
destroyed)
        return (prt.health <= 0);
    }

    // This changes the color of just one Part to red instead of the
whole
    // ship
    void ShowLocalizedDamage(Material m) {                       //
d
        m.color = Color.red;
        damageDoneTime = Time.time + showDamageDuration;
        showingDamage = true;
    }


    // This will override the OnCollisionEnter that is part of
Enemy.cs.
    void OnCollisionEnter( Collision coll ) {                    // e
        GameObject other = coll.gameObject;
```

```
        switch (other.tag) {
            case "ProjectileHero":
                Projectile p = other.GetComponent<Projectile>();
                // If this Enemy is off screen, don't damage it.
                if ( !bndCheck.isOnScreen ) {
                    Destroy( other );
                    break;
                }

                // Hurt this Enemy
                GameObject goHit =
coll.contacts[0].thisCollider.gameObject; Image// f
                Part prtHit = FindPart(goHit);
                if (prtHit == null) { // If prtHit wasn't found...       // g
                    goHit = coll.contacts[0].otherCollider.gameObject;
                    prtHit = FindPart(goHit);
                }
                // Check whether this part is still protected
                if (prtHit.protectedBy != null) {                    // h
                    foreach( string s in prtHit.protectedBy ) {
                        // If one of the protecting parts hasn't been
                        // destroyed...
                        if (!Destroyed(s)) {
                            // ...then don't damage this part yet
                            Destroy(other);  // Destroy the ProjectileHero
                            return; // return before causing damage
                        }
                    }
                }
                // It's not protected, so make it take damage
                // Get the damage amount from the Projectile.type and
                // Main.W_DEFS
                prtHit.health -= Main.GetWeaponDefinition
Image ( p.type ).damageOnHit;
                // Show damage on the part
                ShowLocalizedDamage(prtHit.mat);
                if (prtHit.health <= 0) {                        // i
```

```
            // Instead of destroying this enemy, disable the
damaged
            // part
            prtHit.go.SetActive(false);
         }
         // Check to see if the whole ship is destroyed
         bool allDestroyed = true; // Assume it is destroyed
         foreach( Part prt in parts ) {
            if (!Destroyed(prt)) { // If a part still exists
               allDestroyed = false;// change allDestroyed to
false
               break;        // and break out of the foreach loop
            }
         }
         if (allDestroyed) { // If it IS completely destroyed    // j
            // Tell the Main singleton that this ship has been
            // destroyed
            Main.S.ShipDestroyed( this );
            // Destroy this Enemy
            Destroy(this.gameObject);
         }
         Destroy(other);  // Destroy the ProjectileHero
         break;
      }
   }
}
```

a. The `FindPart()` methods at `// a` and `// b` are overloads of each other, meaning that they are two methods with the same name but different parameters (one takes a string, and the other takes a GameObject). Based on what type of variable is passed in, the correct overload of the `FindPart()` function is executed. In either case, `FindPart()` searches through the `parts` array to find which part the string or GameObject are associated with.

b. A GameObject overload of `FindPart()`. Another overloaded function that we've used before is `Random.range()`; it has different

behavior based on whether floats or ints are passed into it.

c. Three overloads of the `Destroyed()` method that checks to see whether a certain part has been destroyed or still has health.

d. `ShowLocalizedDamage()` is a more specialized version of the inherited `Enemy.ShowDamage()` method. This only turns one part red, not the whole ship.

e. This `OnCollisionEnter()` method completely overrides the inherited `Enemy.OnCollisionEnter()` method. Because of the way that MonoBehaviour declares common Unity functions like `OnCollisionEnter()`, the override keyword is not necessary.

f. This line finds the GameObject that was hit. The Collision `coll` includes a field `contacts[]`, which is an array of ContactPoints. Because there was a collision, we're guaranteed that there is at least one ContactPoint (i.e., `contacts[0]`), and each ContactPoint has a field named `thisCollider`, which is the collider for the part of the Enemy_4 that was hit.

g. If the `prtHit` we searched for wasn't found (and therefore `prtHit == null`), then it's usually because—very rarely—`thisCollider` on `contacts[0]` will refer to the ProjectileHero instead of the ship part. In that case, just look at `contacts[0].otherCollider` instead.

h. If this part is still protected by another part that has not yet been destroyed, apply damage to the protecting part instead.

i. If a single part's health reaches 0, then set it to inactive, which makes it disappear and stop colliding with things.

j. If the whole ship has been destroyed, notify `Main.S.ShipDestroyed()` just like the Enemy script would have (if we hadn't overridden `OnCollisionEnter()`).

4. Play the scene. You should eventually be overwhelmed by many Enemy_4s, each of which has two wings that protect the fuselage

and a fuselage that protects the cockpit. If you want more of a chance against these, you can change the value of Main (Script).enemySpawnPerSecond on the _MainCamera to something lower, which will give you more time between Enemy_4 spawns (though it will also delay the initial spawn).

5. The last thing you should do before continuing on is to set the prefabEnemies array on the Main (Script) of _MainCamera to spawn various enemies with reasonable frequency.

a. Select _MainCamera in the Hierarchy.

b. Set the size of prefabEnemies on the Main (Script) Inspector to 10.

c. Set Elements 0, 1, and 2 to Enemy_0 (from the _Prefabs folder of the Project pane).

d. Set Elements 3 & 4 to Enemy_1.

e. Set Elements 5 & 6 to Enemy_2.

f. Set Elements 7 & 8 to Enemy_3.

g. Set Element 9 to Enemy_4.

Doing all this should give you Enemy_0s pretty frequently and Enemy_4s pretty rarely.

6. Set the powerUpDrop chance of each enemy type.

a. Select Enemy_0 in the _Prefabs folder of the Project pane and set powerUpDropChance in the Enemy (Script) Inspector to 0.25 (meaning that an Enemy_0 will drop a PowerUp 25% of the time).

b. Set the powerUpDropChance on Enemy_1 to 0.5.

c. Set the powerUpDropChance on Enemy_2 to 0.5.

d. Set the powerUpDropChance on Enemy_3 to 0.75.

e. Set the powerUpDropChance on Enemy_4 to 1.

7. Save your scene and press Play to try out your game!

Adding Particle Effects and Background

After all of that coding, here are a couple things you can do just for fun to make the game look a little better.

Starfield Background

Create a two-layer starfield background to make things look more like outer space.

Create a quad in the Hierarchy (*GameObject* > *Create Other* > *Quad*). Name it *StarfieldBG*.



This will place StarfieldBG in the center of the camera's view and fill the view entirely. Now, create a new material named *Mat Starfield* and set its shader to *ProtoTools* > *UnlitAlpha*. Set the texture of Mat Starfield to the *Space Texture2D* that you imported at the beginning of this tutorial. Now drag Mat Starfield onto StarfieldBG, and you should see a starfield behind your _Hero ship.

Select Mat Starfield in the Project pane and duplicate it (Command-D on Mac or Control+D on PC). Name the new material *Mat Starfield Transparent*. Select *Space_Transparent* as the texture for this new material.

Select StarfieldBG in the Hierarchy and duplicate it. Name the duplicate *StarfieldFG_0*. Drag the Mat Starfield Transparent material onto StarfieldFG_0 and set its transform.



Now if you drag StarfieldFG_0 around a bit, you'll see that it moves some stars in the foreground past stars in the background, creating a nifty parallax effect. Now duplicate Starfield_FG_0 and name the duplicate *Starfield_FG_1*. You will need two copies of the foreground for the scrolling trick that we're going to employ.

Create a new C# script named *Parallax* and edit it in MonoDevelop.

```
using UnityEngine;
using System.Collections;

public class Parallax : MonoBehaviour {

    public GameObject       poi; // The player ship
    public GameObject[]     panels; // The scrolling foregrounds
    public float            scrollSpeed = -30f;
    // motionMult controls how much panels react to player
movement
    public float            motionMult = 0.25f;

    private float panelHt; // Height of each panel
    private float depth;   // Depth of panels (that is, pos.z)

    // Use this for initialization
    void Start () {
        panelHt = panels[0].transform.localScale.y;
        depth = panels[0].transform.position.z;

        // Set initial positions of panels
        panels[0].transform.position = new Vector3(0,0,depth);
        panels[1].transform.position = new
Vector3(0,panelHt,depth);
    }

    // Update is called once per frame
```

```
    void Update () {
        float tY, tX=0;
        tY= Time.time * scrollSpeed % panelHt + (panelHt*0.5f);

        if (poi != null) {
            //tY += -poi.transform.position.y * motionMult;
            tX = -poi.transform.position.x * motionMult;
        }

        // Position panels[0]
        panels[0].transform.position = new Vector3(tX, tY, depth);
        // Then position panels[1] where needed to make a
continuous starfield
        if (tY >= 0) {
            panels[1].transform.position = new Vector3(tX, tY-
panelHt, depth);
        } else {
            panels[1].transform.position = new Vector3(tX,
tY+panelHt, depth);
        }
    }
}
```

Save the script, return to Unity, and assign the script to _MainCamera. Select _MainCamera in the Hierarchy and find the *Parallax (Script)* component in the Inspector. There, set the `poi` to _Hero and add StarfieldFG_0 and StarfieldFG_1 to the `panels` array. Now press Play, and you should see the starfield moving in response to the player.

And of course, remember to save your scene.

Next Steps

From your experience in the previous tutorials, you already understand how to do many of the things listed in this section. These

are just some recommendations on what you can do if you want to keep going with this prototype.

Tune Variables

As you have learned in both paper and digital games, tuning of numbers is critically important and has a significant effect on experience. Below is a list of variables you should consider tuning to change the feel of the game:

• _Hero: Change how movement feels

• Adjust the `speed`.

• Modify the gravity and sensitivity of the horizontal and vertical axes in the InputManager.

• Weapons: Differentiate weapons more

• Spread: The spread gun could shoot five projectiles instead of just three but have a much longer `delayBetweenShots`.

• Blaster: The blaster could fire more rapidly (smaller `delayBetweenShots`) but do less damage with each shot (reduced `damageOnHit`).

• Power-ups: Adjust drop rate

• Each `Enemy` class has a `powerUpDropChance` field that can be set to any number between 0 (never drop a power-up) to 1 (always drop a power-up). These were set to 1 for testing, but you can adjust them to whatever you want.

Add Additional Elements

While this prototype has so far shown five kinds of enemies and two kinds of weapons, there are infinite possibilities for either open to

you:

• Weapons: Add additional weapons

• Phaser: Shoots two projectiles that move in a sine wave pattern (similar to the movement of Enemy_1).

• Laser: Instead of doing all of its damage at once, the laser does continuous damage over time.

• Missiles: Missiles could have a lock-on mechanic and have a very slow fire-rate but would track enemies and always hit. Perhaps missiles could be a different kind of weapon with limited ammunition that were fired using a different button (that is, not the space bar).

• Swivel Gun: Like the blaster but actually shoots at the nearest enemy. However, the gun is very weak.

• Enemies: Add additional enemies. There are countless kinds of enemies that could be created for this game.

• Add additional enemy abilities

• Allow some enemies to shoot.

• Some enemies could track and follow the player, possibly acting like missiles homing in on the player.

• Add level progression

• Make specific, timed waves instead of the randomized infinite attack in the existing prototype. This could be accomplished using a `[System.Serializable] Wave` class as defined here:

```
    [System.Serializable]
    public class Wave {
        float       delayBeforeWave=1; // secs to delay after the prev
wave
        GameObject[] ships;             // array of ships in this wave
```

```
    // Delay the next wave until this wave is completely killed?
    bool        delayNextWaveUntilThisWaveIsDead=false;
}
```

• Add a Level class to contain the `Wave[]` array:

```
[System.Serializable]
public class Level {
    Wave[]      waves;  // Holder for waves
    float       timeLimit=-1; // If -1, there is no time limit
    string      name = ""; // If the name is left blank (i.e., ""),
                           //  the name could appear as "Level #1"
}
```

However, this will cause issues because even if Level is serializable, the `Wave[]` array won't appear properly because the Unity Inspector won't allow nested serializable classes. This means that you should probably try something like an XML document to define levels and waves which can then be read into `Level` and `Wave` classes. XML is covered in the "XML" section of Appendix B and is used in the next prototype, Prospector Solitaire.

• Add more game structure and GUI (graphical user interface) elements:

• Give the player a score and a specific number of lives (both of these were covered in the Mission Demolition prototype).

• Add difficulty settings.

• Track high scores (as covered in the Apple Picker and Mission Demolition prototypes).

• Create a title screen scene that welcomes the player to the game and allows them to choose the difficulty setting. This could also show high scores.

Summary

This was a long chapter, but it introduced a lot of important concepts that I hope will help you with your own game projects in the future. Over the years, I have made extensive use of linear interpolation and Bézier curves to make the motion in my games and other projects smooth and refined. Just a simple easing function can make the movement of an object look graceful, excited, or lethargic, which is a powerful when you're trying to balance and tune the feel of a game.

In the next chapter, we move on to a very different kind of game: a solitaire card game (actually, my favorite solitaire card game). The next chapter features reading information from an XML file to construct an entire deck of cards out of just a few art assets and also using XML to lay out the game itself. And, at the end, you'll have a fun digital card game to play.

Chapter 32. Prototype 4: Prospector Solitaire

In this chapter, you make your first card game, a digital version of the popular Tri-Peaks Solitaire game. By the end of the chapter, you'll have not only a working card game but also a great framework for future card games you wish to create.

This chapter includes several new techniques, including using XML configuration files, designing for mobile devices, and your first look at Unity's 2D sprite tools.

Getting Started: Prototype 4

As with Prototype 3, this starts with you being asked to download and import a unitypackage of assets for this game. The art assets we'll be using are constructed from parts of the publicly available *Vectorized Playing Cards 1.3* by Chris Aguilar.[1]

[1] Chris Aguilar, "Vectorized Playing Cards 1.3," http://code.google.com/p/vectorized-playing-cards.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see Appendix A, "Standard Project Setup Procedure." When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 2D for this project.

- **`Project name:`** Prospector.

• **`Download and import package:`** Go to Chapter 32 at http://book.prototools.net. Downloading this package should set up the scene and several folders.

• **`Scene name:`** (The scene *__Prospector_Scene_0* will import with the starter package, so you don't need to create it.)

• **`Project folders:`** _Prefabs (__Scripts, _Sprites, and Resources should be imported.)

• **`C# script names:`** (none yet)

• **`Rename:`** Change Main Camera to _MainCamera.

Open __Prospector_Scene_0 and double check the settings for _MainCamera.



Note that this unitypackage includes a version of the Utils script that has additional functions beyond what you wrote in the previous chapter.

Build Settings

This will be the first project designed to be able to be compiled on mobile devices. As an example, I'll be using settings for the Apple iPad, but it's perfectly fine to use Android or even a WebGL or Standalone build instead if you prefer. This project is designed for the 4:3 aspect ratio screen of an iPad in portrait mode. Though this project is designed to be able to be compiled for a mobile device, the actual build process for mobile devices is beyond the scope of this book (and would differ widely based on which device you own), but you can find a lot of information about it Unity's website. The link for iOS development is

http://docs.unity3d.com/Documentation/Manual/iphone-GettingStarted.html.

1. Double-click the __Prospector_Scene_0 scene in the Project pane to open it.

2. From the menu bar choose *File* > *Build Settings*, which brings up the window in Figure 32.1.

Figure 32.1. The Build Settings window



3. Click *Add Open Scenes* to add __Prospector_Scene_0 to the list of scenes for this build.

4. Select iOS from the list of platforms and click *Switch Platform*. Unity will reimport all of your images to match the default iOS settings. The *Switch Platform* button will turn gray once the switch is complete. Once your Build Settings look like the image in Figure 32.1, you can close this window. (Don't click Build yet; that would happen after actually making the game.)

Figure 32.2. Import settings for the Texture 2Ds that will become sprites



Importing Images as Sprites

Next, we'll need to properly import the images we're using as *sprites*. A sprite is a 2D image that can be moved around screen and rotated, and they are very common in 2D games:

1. Open the _Sprites folder in your Project pane and select all the images therein. (Click the top image and then Shift-click the bottom image in the _Sprites folder.) Looking at the Preview in the Inspector pane, you can see that all of them are currently imported as square

images with no transparency. We're going to change that and make them usable sprites.

2. In the *21 Texture 2Ds Import Settings* section of the Inspector pane, set the *Texture Type* to *Sprite (2D and UI)*. Then click *Apply*, and Unity will reimport all the images at their proper aspect ratio. Figure 32.2 shows the final import settings.

Looking at the Project pane, you will see that each of the images now has a disclosure triangle next to it. If you open the disclosure triangle, you'll find a sprite with the same name under each image.

3. Select the *Letters* image in the Project pane. For most of the images that were imported, a Sprite Mode of Single is appropriate because each image becomes a single sprite. However, the Letters image is actually a sprite atlas (a series of sprites saved as a single image), so it requires different settings.

4. In *Letters Import Settings* in the Inspector pane, change the Sprite Mode to *Multiple* and click *Apply*. This will add a new *Sprite Editor* button under the *Pixels to Units* field.

5. Click the *Sprite Editor* button to open the Sprite Editor. You'll see the Letters image there with a single blue box around it defining the bounds of the Letters sprite.

6. Click the small icon with either a rainbow or a letter *A* on it in the Sprite Editor (circled in the Figure 32.3) to switch between viewing the actual image and the alpha channel of the image. Because Letters is an image of white letters over a transparent background, it may be easier to see what's happening if you look at the alpha channel.

Figure 32.3. The Sprite Editor showing the correct settings for the grid slicing of Letters. The button circled in the top right switches between viewing the color channels and the alpha channel of Letters.



7. Now, click the *Slice* pop-up menu in the top-left corner of the Sprite Editor.

a. and change the *Type* from Automatic to *Grid by Cell Size* (see Figure 32.3).

b. Set the *Pixel size* to X:32 Y:32.

c. Click the *Slice* button. This will chop Letters horizontally into 16 sprites that are each 32x32 pixels in size.

d. Click *Apply* (in the top-right corner of the Sprite Editor) to generate these sprites in the Project pane. Now instead of one Letters sprite under the Letters texture in the Project pane, there are 16 sprites named Letters_0 to Letters_15. In this game, you will use Letters_1 to Letters_13 for each of the 13 ranks of cards (Ace through King). Now all the sprites are set up and ready to be used.

8. Save your scene. You haven't actually altered the scene yet, but it's good practice to save your scene all the time. You should be in the habit of saving your scene any time you change anything.

Constructing Cards from Sprites

One of the most important aspects of this project is that we're going to procedurally construct an entire deck of cards from the 21 images that were imported. This will make the final build size smaller and will give you a chance to see how XML works.

The image in Figure 32.4 shows an example of how this will be done. The 10 of Spades in the image is constructed from the following sprites: Card_Front, 12 copies of Spade, and 2 copies of the Letters_10.

Figure 32.4. The 10 of Spades showing autogenerated borders around each of the sprites from which it is made. The visible part of

this card is composed of 15 different sprites (12 Spades, 2 Letter_10s, and 1 Card_Front).



This is defined through the use of an XML file. Read the “XML” section of Appendix B, “Useful Concepts,” now to learn more about XML and how it can be read using the PT_XMLReader that was part of the imported unitypackage. The structure of the DeckXML.xml file used in this project is also covered in that section of Appendix B.

Making Use of XML Through Code

For the first part of this project, create three C# files named *Card*, *Deck*, and *Prospector*.

• `Card`: The class for each individual card in the deck. The Card script will also contain the `CardDefinition` class (which holds information about where sprites are to be positioned on each rank of card) and the `Decorator` class (which holds information about the decorators and pips described in the XML document).

• `Deck`: The `Deck` class interprets the information in DeckXML.xml and uses that information to create an entire deck of cards.

• `Prospector`: The `Prospector` class manages the overall game. While `Deck` handles the creation of cards, `Prospector` turns those cards into a game. Prospector collects the cards into various piles (like the draw pile and discard pile) and manages game logic.

1. Start by creating the Card C# script and entering the following code. These small classes in Card.cs will store the information created when `Deck` reads the XML file.

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

```
public class Card : MonoBehaviour {
   // This will be defined later
}

[System.Serializable]
public class Decorator {
// This class stores information about each decorator or pip
from DeckXML
   public string    type; // For card pips, type = "pip"
   public Vector3   loc;  // The location of the Sprite on the Card
   public bool      flip = false;  // Whether to flip the Sprite
vertically
   public float     scale = 1f;    // The scale of the Sprite
}

[System.Serializable]
public class CardDefinition {
// This class stores information for each rank of card
   public string            face; // Sprite to use for each face card
   public int               rank; // The rank (1-13) of this card
   public List<Decorator>    pips = new List<Decorator>();  //
Pips used
   // Because decorators (from the XML) are used the same way
on every card
   // in the deck, pips only stores information about the pips on
numbered
   // cards
}
```

2. Open the Deck C# script in MonoDevelop and enter the following code:

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

public class Deck : MonoBehaviour {
```

```
    [Header("These fields are set dynamically")]
    public PT_XMLReader          xmlr;

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        ReadDeck(deckXMLText);
    }

    // ReadDeck parses the XML file passed to it into
CardDefinitions
    public void ReadDeck(string deckXMLText) {
        xmlr = new PT_XMLReader(); // Create a new
PT_XMLReader
        xmlr.Parse(deckXMLText);   // Use that PT_XMLReader to
parse DeckXML

        // This prints a test line to show you how xmlr can be used.
        // For more information read about XML in the Useful
Concepts
        // Appendix
        string s = "xml[0] decorator[0] ";
        s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
        s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
        s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
        s += " scale="+xmlr.xml["xml"][0]["decorator"]
[0].att("scale");
        print(s);

    }
}
```

3. Now open the `Prospector` class and enter this code:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;    // This will be used later in
```

```
the project
using UnityEngine.SceneManagement;   // This will be used later
in the project
using UnityEngine.UI;                // This will be used later in the
project

public class Prospector : MonoBehaviour {
   static public Prospector    S;

   [Header("Set in the Unity Inspector")]
   public TextAsset          deckXML;

   [Header("These fields are set dynamically")]
   public Deck               deck;

   void Awake() {
      S = this; // Set up a Singleton for Prospector
   }

   void Start () {
      deck = GetComponent<Deck>(); // Get the Deck
      deck.InitDeck(deckXML.text); // Pass DeckXML to it
   }
}
```

4. Now that the code is ready, go back to Unity and attach both the `Prospector` and `Deck` classes to _MainCamera. (Drag each script from the Project pane onto _MainCamera in the Hierarchy pane.) Then select _MainCamera in the Hierarchy. You should see both scripts attached as Script components.

5. Drag DeckXML from the Resources folder in the Project pane into the `deckXML` TextAsset variable in the Inspector for the Prospector (Script) component.

6. Save your scene and click Play. You should see the following output in the console:

xml[0] decorator[0] type=letter x=-1.05 y=1.42 scale=1.25

This line comes from the test code in `Deck:ReadDeck()` and shows that `ReadDeck()` is properly reading the type, x, y, and scale attributes from the 0th decorator of the 0th xml in the XML file, as seen in the following lines from DeckXML.xml. (You can see the entire text of DeckXML.xml in the “XML” section of Appendix B.)

```
<xml>
    <decorator type="letter" x="-1.05" y="1.42" z="0" flip="0"
scale="1.25"/>
    ...
</xml>
```

Parsing Information from Deck XML

Now, let’s actually do something with this information.

1. Make the following changes to the `Deck` class:

```
public class Deck : MonoBehaviour {

    [Header("These fields are set dynamically")]
    public PT_XMLReader             xmlr;
    public List<string>             cardNames;
    public List<Card>               cards;
    public List<Decorator>          decorators;
    public List<CardDefinition>     cardDefs;
    public Transform                deckAnchor;
    public Dictionary<string,Sprite>  dictSuits;

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        ReadDeck(deckXMLText);
    }

    // ReadDeck parses the XML file passed to it into CardDefinitions
```

```
    public void ReadDeck(string deckXMLText) {
        xmlr = new PT_XMLReader(); // Create a new PT_XMLReader
        xmlr.Parse(deckXMLText);   // Use that PT_XMLReader to
parse DeckXML

        // This prints a test line to show you how xmlr can be used.
        // For more information read about XML in the Useful Concepts
        // Appendix
        string s = "xml[0] decorator[0] ";
        s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
        s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
        s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
        s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
        //print(s); // Comment out this line, since we're done with
the test

        // Read decorators for all Cards
        decorators = new List<Decorator>(); // Init the List of
Decorators
        // Grab an PT_XMLHashList of all <decorator>s in the XML
file
        PT_XMLHashList xDecos = xmlr.xml["xml"][0]
["decorator"];
        Decorator deco;
        for (int i=0; i<xDecos.Count; i++) {
            // For each <decorator> in the XML
            deco = new Decorator(); // Make a new Decorator
            // Copy the attributes of the <decorator> to the Decorator
            deco.type = xDecos[i].att("type");
            // Set the bool flip based on whether the text of the
attribute
            // is "1" or something else. This is an atypical but
perfectly
            // fine use of the == comparison operator. It will return a
true
            // or false, which will be assigned to deco.flip.
            deco.flip = ( xDecos[i].att ("flip") == "1" );
```

```
            // floats need to be parsed from the attribute strings
            deco.scale = float.Parse( xDecos[i].att ("scale") );
            // Vector3 loc initializes to [0,0,0], so we just need to
modify
            // it
            deco.loc.x = float.Parse( xDecos[i].att ("x") );
            deco.loc.y = float.Parse( xDecos[i].att ("y") );
            deco.loc.z = float.Parse( xDecos[i].att ("z") );
            // Add the temporary deco to the List decorators
            decorators.Add (deco);
        }

        // Read pip locations for each card number
        cardDefs = new List<CardDefinition>(); // Init the List of
Cards
        // Grab an PT_XMLHashList of all the <card>s in the XML
file
        PT_XMLHashList xCardDefs = xmlr.xml["xml"][0]["card"];
        for (int i=0; i<xCardDefs.Count; i++) {
            // For each of the <card>s
            // Create a new CardDefinition
            CardDefinition cDef = new CardDefinition();
            // Parse the attribute values and add them to cDef
            cDef.rank = int.Parse( xCardDefs[i].att ("rank") );
            // Grab an PT_XMLHashList of all the <pip>s on this
<card>
            PT_XMLHashList xPips = xCardDefs[i]["pip"];
            if (xPips != null) {
                for (int j=0; j<xPips.Count; j++) {
                    // Iterate through all the <pip>s
                    deco = new Decorator();
                    // <pip>s on the <card> are handled via the
                    // Class
                    deco.type = "pip";
                    deco.flip = ( xPips[j].att ("flip") == "1" );
                    deco.loc.x = float.Parse( xPips[j].att ("x") );
                    deco.loc.y = float.Parse( xPips[j].att ("y") );
```

```
                deco.loc.z = float.Parse( xPips[j].att ("z") );
                if ( xPips[j].HasAtt("scale") ) {
                    deco.scale = float.Parse( xPips[j].att ("scale") );
                }
                cDef.pips.Add(deco);
            }
        }
        // Face cards (Jack, Queen, & King) have a face attribute
        // cDef.face is the base name of the face card Sprite
        // e.g., FaceCard_11 is the base name for the Jack face
Sprites
        // the Jack of Clubs is FaceCard_11C, hearts is
FaceCard_11H,
        // etc.
        if (xCardDefs[i].HasAtt("face")) {
            cDef.face = xCardDefs[i].att ("face");
        }
        cardDefs.Add(cDef);
    }

  }

}
```

Now, the `ReadDeck()` method will parse the XML and turn it into a list of Decorators (the suit and rank in the corners of the card) and CardDefinitions (a class containing information about each of the ranks of card (Ace through King).

2. Switch back to Unity and press Play. Then click on the _MainCamera and look at the Inspector for the Deck (Script) component. Because both Decorator and CardDefinition were set to [System.Serializable], they appear properly in the Unity Inspector, as shown in Figure 32.5.

Figure 32.5. The Inspector for the Deck (Script) component of _MainCamera showing Decorators and Card Defs that have been

read from the DecXML.xml file



3. Stop playback and *save your scene.*

Assigning the Sprites That Become Cards

Now that the XML has been properly read and parsed into usable Lists, it's time to make some cards. The first step in doing so is to get references to all of those sprites that we made earlier in the chapter:

1. Add the following variables to the top of the `Deck` class to hold these sprites:

```
public class Deck : MonoBehaviour {
    [Header("Set in the Unity Inspector")]
    // Suits
    public Sprite          suitClub;
    public Sprite          suitDiamond;
    public Sprite          suitHeart;
    public Sprite          suitSpade;

    public Sprite[]        faceSprites;
    public Sprite[]        rankSprites;

    public Sprite          cardBack;
    public Sprite          cardBackGold;
    public Sprite          cardFront;
    public Sprite          cardFrontGold;

    // Prefabs
    public GameObject      prefabCard;
    public GameObject      prefabSprite;

    [Header("These fields are set dynamically")]
```

When you switch back to Unity, there will now be many new public variables that need to be defined in the Deck (Sprite) Inspector.

2. Drag the Club, Diamond, Heart, and Spade textures from the _Sprites folder in the Project pane into their respective variables under Deck (`suitClub`, `suitDiamond`, `suitHeart`, and `suitSpade`). Unity will automatically assign the sprite to the variable (as opposed to attempting to assign the Texture2D to a sprite variable).

3. The next bit is a touch trickier. Lock the Inspector on _MainCamera by selecting _MainCamera in the Hierarchy pane and then clicking the tiny lock icon at the top of the Inspector pane (circled in red in the Figure 32.6). Locking the Inspector pane ensures that it won't change which object is displayed when you select something new.

Figure 32.6. The Inspector for the Deck (Script) Component of _MainCamera showing the correct sprites assigned to each public sprite variable



4. Next, we'll assign each of the sprites starting with *FaceCard_* to an element of the array `faceSprites` in the Inspector for *Deck (Script)*.

a. Select *FaceCard_11C* in the *_Sprites* folder of the Project pane and then hold shift and click *FaceCard_13S*. This should select all 12 *FaceCard_* sprites.

b. Now, drag this group from the Project pane over the name of the array `faceSprites` under *Deck (Script)* in the Inspector. You should see a plus icon appear next to the word *<multiple>* when hovering over the variable name `faceSprites` (on PC, you may only see the + icon).

c. Release the mouse button, and if done correctly, this should expand the size of the `faceSprites` array to 12 and fill it with one

copy of each of the *FaceCard_* sprites. If this doesn't work, you can also add them individually. The order doesn't matter as long as there is exactly one of each when you're done (see Figure 32.6).

5. Open the disclosure triangle next to *Letters* in the Project pane. Use the same process as in the previous step to select Letters_0 through Letters_15. You should now have all 16 sprites under Letters selected. Drag this group of sprites onto the `rankSprites` variable in Deck (Script). If done correctly, the `rankSprites` list should now be full of 16 Letters sprites named Letters_0 through Letters_15. Double-check to make sure that they're in the correct order with Letters_0 in Element 0 and Letters_15 in Element 15; if not, you may have to add them one at a time.

6. Drag the sprites *Card_Back*, *Card_Back_Gold*, *Card_Front*, and *Card_Front_Gold* from the Project pane to their respective variable slots in the *Deck (Script)* Inspector.

7. Unlock the Inspector pane by clicking the tiny lock icon again (circled in red in Figure 32.6). Your Inspector for *Deck (Script)* should now look like what is shown in the figure.

Creating Prefab GameObjects for Sprites and Cards

Just like anything else on screen, sprites need to be enclosed in GameObjects. For this project, you will need two prefabs: a generic *PrefabSprite* that will be used for all decorators and pips (and was imported as part of the starter asset package), and a *PrefabCard* that will form the basis of all the cards in the deck.

PrefabCard

To create PrefabCard, do the following:

1. From the menu bar, choose *GameObject* > *2D Object* > *Sprite*. Name this GameObject *PrefabCard*.

2. Drag *Card_Front* from the Project pane into the *sprite* variable of the *Sprite Renderer* in the PrefabCard Inspector. Now you should see the Card_Front sprite in the Scene pane.

3. Drag the Card script from the Project pane onto PrefabCard in the Hierarchy. This will assign the Card script to PrefabCard (and the Card (Script) component will now appear in the Inspector for PrefabCard).

4. In the Inspector for PrefabCard, click the *Add Component* button. Choose *Physics* > *Box Collider* from the menu that appears. (This is the same as choosing *Component* > *Physics* > *Box Collider* from the menu bar.) The Size of the Box Collider should automatically set itself to [2.56, 3.56, 0.2], but if not, set the Size to those values.

5. Drag PrefabCard from the Hierarchy into the _Prefabs folder to make a prefab from it.

6. Delete the remaining instance of PrefabCard from the hierarchy, and save your Scene.

Now, you need to assign these two prefabs to their respective public variables in the Inspector for the *Deck (Script)* component on _MainCamera.

1. Select _MainCamera in the hierarchy, and drag PrefabCard and PrefabSprite from the Project pane into their respective variables in the *Deck (Script)* Inspector.

2. Save your scene.

Building the Cards in Code

Before actually adding the method to the `Deck` class to make the cards, we need to add variables to `Card`, as follows:

1. Replace the comment `// This will be defined later` in the Card class with the following code.

```
public class Card : MonoBehaviour {
    public string          suit; // Suit of the Card (C,D,H, or S)
    public int             rank; // Rank of the Card (1-14)
    public Color           color = Color.black; // Color to tint pips
    public string          colS = "Black"; // or "Red". Name of the
Color

    // This List holds all of the Decorator GameObjects
    public List<GameObject> decoGOs = new List<GameObject>
();
    // This List holds all of the Pip GameObjects
    public List<GameObject> pipGOs  = new List<GameObject>
();

    public GameObject      back; // The GameObject of the back
of the card

    public CardDefinition   def; // Parsed from DeckXML.xml
}
```

2. Now, add this code to `Deck`:

```
public class Deck : MonoBehaviour {
    ...

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        // This creates an anchor for all the Card GameObjects in
the
        // Hierarchy
        if (GameObject.Find("_Deck") == null) {
            GameObject anchorGO = new GameObject("_Deck");
            deckAnchor = anchorGO.transform;
        }

        // Initialize the Dictionary of SuitSprites with necessary
Sprites
```

```
        dictSuits = new Dictionary<string, Sprite>() {
            { "C", suitClub },
            { "D", suitDiamond },
            { "H", suitHeart },
            { "S", suitSpade }
        };

        ReadDeck(deckXMLText);
        MakeCards();
    }


    // ReadDeck parses the XML file passed to it into CardDefinitions
    public void ReadDeck(string deckXMLText) {
        ...
    }

    // Get the proper CardDefinition based on Rank (1 to 14 is Ace
to King)
    public CardDefinition GetCardDefinitionByRank(int rnk) {
        // Search through all of the CardDefinitions
        foreach (CardDefinition cd in cardDefs) {
            // If the rank is correct, return this definition
            if (cd.rank == rnk) {
                return( cd );
            }
        }
        return( null );
    }

    // Make the Card GameObjects
    public void MakeCards() {
        // cardNames will be the names of cards to build
        // Each suit goes from 1 to 14 (e.g., C1 to C14 for Clubs)
        cardNames = new List<string>();
        string[] letters = new string[] {"C","D","H","S"};
        foreach (string s in letters) {
```

```
        for (int i=0; i<13; i++) {
            cardNames.Add(s+(i+1));
        }
    }

    // Make a List to hold all the cards
    cards = new List<Card>();
    // Several variables that will be reused several times
    Sprite tS = null;
    GameObject tGO = null;
    SpriteRenderer tSR = null;

    // Iterate through all of the card names that were just made
    for (int i=0; i<cardNames.Count; i++) {
        // Create a new Card GameObject
        GameObject cgo = Instantiate(prefabCard) as
GameObject;
        // Set the transform.parent of the new card to the anchor.
        cgo.transform.parent = deckAnchor;
        Card card = cgo.GetComponent<Card>(); // Get the Card
Component

        // This just stacks the cards so that they're all in nice
rows
        cgo.transform.localPosition = new Vector3( (i%13)*3,
i/13*4, 0 Image);

        // Assign basic values to the Card
        card.name = cardNames[i];
        card.suit = card.name[0].ToString();
        card.rank = int.Parse( card.name.Substring(1) );
        if (card.suit == "D" || card.suit == "H") {
            card.colS = "Red";
            card.color = Color.red;
        }
        // Pull the CardDefinition for this card
        card.def = GetCardDefinitionByRank(card.rank);
```

```
        // Add Decorators
        foreach( Decorator deco in decorators ) {
            if (deco.type == "suit") {
                // Instantiate a Sprite GameObject
                tGO = Instantiate( prefabSprite ) as GameObject;
                // Get the SpriteRenderer Component
                tSR = tGO.GetComponent<SpriteRenderer>();
                // Set the Sprite to the proper suit
                tSR.sprite = dictSuits[card.suit];
            } else {
                tGO = Instantiate( prefabSprite ) as GameObject;
                tSR = tGO.GetComponent<SpriteRenderer>();
                // Get the proper Sprite to show this rank
                tS = rankSprites[ card.rank ];
                // Assign this rank Sprite to the SpriteRenderer
                tSR.sprite = tS;
                // Set the color of the rank to match the suit
                tSR.color = card.color;
            }
            // Make the deco Sprites render above the Card
            tSR.sortingOrder = 1;
            // Make the decorator Sprite a child of the Card
            tGO.transform.parent = cgo.transform;
            // Set the localPosition based on the location from
DeckXML
            tGO.transform.localPosition = deco.loc;
            // Flip the decorator if needed
            if (deco.flip) {
                // A Euler rotation of 180° around the Z-axis will flip
                // it
                tGO.transform.rotation = Quaternion.Euler(0,0,180);
            }
            // Set the scale to keep decos from being too big
            if (deco.scale != 1) {
                tGO.transform.localScale = Vector3.one *
deco.scale;
```

```
            }
            // Name this GameObject so it's easy to see
            tGO.name = deco.type;
            // Add this deco GameObject to the List card.decoGOs
            card.decoGOs.Add(tGO);

        }

        // Add the card to the deck
        cards.Add (card);
    }
  }
}
```

3. Press Play. You should see 52 cards lined up. They don't yet have pips, but they do appear, and the correct decorators and coloring are on them.

4. Now let's add the code for pips and faces. Add the following to the `MakeCards()` method of the Deck class:

```
  // Make the Card GameObjects
  public void MakeCards() {
      ...
      // Iterate through all of the card names that were just made
      for (int i=0; i<cardNames.Count; i++) {
          ...

          // Add Decorators
          foreach( Decorator deco in decorators ) {
              ...
          }

          // Add Pips
          // For each of the pips in the definition
          foreach( Decorator pip in card.def.pips ) {
              // Instantiate a Sprite GameObject
```

```
    tGO = Instantiate( prefabSprite ) as GameObject;
    // Set the parent to be the card GameObject
    tGO.transform.parent = cgo.transform;
    // Set the position to that specified in the XML
    tGO.transform.localPosition = pip.loc;
    // Flip it if necessary
    if (pip.flip) {
       tGO.transform.rotation = Quaternion.Euler(0,0,180);
    }
    // Scale it if necessary (only for the Ace)
    if (pip.scale != 1) {
       tGO.transform.localScale = Vector3.one * pip.scale;
    }
    // Give this GameObject a name
    tGO.name = "pip";
    // Get the SpriteRenderer Component
    tSR = tGO.GetComponent<SpriteRenderer>();
    // Set the Sprite to the proper suit
    tSR.sprite = dictSuits[card.suit];
    // Set sortingOrder so the pip is rendered above the
    // Card_Front
    tSR.sortingOrder = 1;
    // Add this to the Card's list of pips
    card.pipGOs.Add(tGO);
}

// Handle Face Cards
if (card.def.face != "") { // If this has a face in card.def
    tGO = Instantiate( prefabSprite ) as GameObject;
    tSR = tGO.GetComponent<SpriteRenderer>();
    // Generate the right name and pass it to GetFace()
    tS = GetFace( card.def.face+card.suit );
    tSR.sprite = tS;      // Assign this Sprite to tSR
    tSR.sortingOrder = 1; // Set the sortingOrder
    tGO.transform.parent = card.transform;
    tGO.transform.localPosition = Vector3.zero;
    tGO.name = "face";
```

```
            }

            // Add the card to the deck
            cards.Add (card);

        }
    } // This is the closing brace for MakeCards()

    // Find the proper face card Sprite
    public Sprite GetFace(string faceS) {
        foreach (Sprite tS in faceSprites) {
            // If this Sprite has the right name...
            if (tS.name == faceS) {
                // ...then return the Sprite
                return( tS );
            }
        }
        // If nothing can be found, return null
        return( null );
    }
```

5. Pressing Play now, you should see all 52 cards laid out properly with pips and faces for face cards.

The next thing to do is add a back to the cards. The back will have a higher *sortingOrder* than anything else on the card, and it will be visible when the card is face-down and invisible when the card is face-up.

6. To accomplish this visibility toggle, add the following `faceUp` property to the `Card` class. As a property, `faceUp` is actually two functions (a get and a set) masquerading as a single field:

```
public class Card : MonoBehaviour {
    ...
    public CardDefinition   def; // Parsed from DeckXML.xml

    public bool faceUp {
```

```
        get {
            return( !back.activeSelf );
        }
        set {
            back.SetActive(!value);
        }
    }
}
```

7. Now, the back can be added to the card in `MakeCards()`. Add the following lines to the `MakeCards()` method of the `Deck` class:

```
    // Make the Card GameObjects
    public void MakeCards() {
        ...

        // Iterate through all of the card names that were just made
        for (int i=0; i<cardNames.Count; i++) {
            ...
            // Handle Face Cards
            if (card.def.face != "") { // If this has a face in card.def
                ...
            }

            // Add Card Back
            // The Card_Back will be able to cover everything else on
the
            // Card
            tGO = Instantiate( prefabSprite ) as GameObject;
            tSR = tGO.GetComponent<SpriteRenderer>();
            tSR.sprite = cardBack;
            tGO.transform.parent = card.transform;
            tGO.transform.localPosition = Vector3.zero;
            // This is a higher sortingOrder than anything else
            tSR.sortingOrder = 2;
            tGO.name = "back";
            card.back = tGO;
```

```
        // Default to face-up
        card.faceUp = false; // Use the property faceUp of Card

        // Add the card to the deck
        cards.Add (card);

    }
}
```

8. Press Play, and you'll see that all the cards are now flipped face-down. If you change the last added line above to `card.faceUp = true;`, all of them will be face-up.

Shuffling the Cards

Now that cards can be built and displayed on screen, the last thing that we will need from the `Deck` class is the capability to shuffle cards.

1. Add the following `Shuffle()` method after the `GetFace()` method of the Deck class:

```
public class Deck : MonoBehaviour {
    ...
    public Sprite GetFace(string faceS) { ... }

    // Shuffle the Cards in Deck.cards
    static public void Shuffle(ref List<Card> oCards) {              //
a
        // Create a temporary List to hold the new shuffle order
        List<Card> tCards = new List<Card>();

        int ndx; // This will hold the index of the card to be moved
        tCards = new List<Card>(); // Initialize the temporary List
        // Repeat as long as there are cards in the original List
        while (oCards.Count > 0) {
            // Pick the index of a random card
```

```
            ndx = Random.Range(0,oCards.Count);
            // Add that card to the temporary List
            tCards.Add (oCards[ndx]);
            // And remove that card from the original List
            oCards.RemoveAt(ndx);
        }
        // Replace the original List with the temporary List
        oCards = tCards;
        // Because oCards is a reference variable, the original that
was
        // passed in is changed as well.
    }
}
```

a. The `ref` keyword is used to make sure that the `List<Card>` that is passed to `List<Card> oCards` is passed as a reference rather than copied into `oCards`. This means that anything that happens to `oCards` is actually happening to the variable that is passed in. In other words, if the cards of a `Deck` are passed in via reference, those cards will be shuffled without requiring a return variable.

2. Make the following change to the Prospector script to see this work:

```
public class Prospector : MonoBehaviour {
    static public Prospector    S;

    public Deck                 deck;
    public TextAsset            deckXML;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }

    void Start () {
        deck = GetComponent<Deck>();     // Get the Deck
        deck.InitDeck(deckXML.text);     // Pass DeckXML to it
```

```
        Deck.Shuffle(ref deck.cards);    // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which
allows
        // deck.cards to be modified by Deck.Shuffle()
    }
}
```

3. If you play the scene now, you can select _MainCamera in the scene hierarchy and look at the Deck.cards variable to see a shuffled array of cards.

Now that the `Deck` class can shuffle any list of cards, you have the basic tools to create *any* card game. The game you will make in this prototype is called Prospector.

The Prospector Game

The code up till now has given you the basic tools to make any card game. Now let's talk about the specific game we're going to make.

Prospector is based on the classic solitaire card game Tri-Peaks. The rules of both are the same, except for two things:

1. The premise of Prospector is that they player is digging down for gold, whereas the premise of Tri-Peaks is that they player is trying to climb three mountains.

2. The objective of Tri-Peaks is just to clear all of the cards. The objective of Prospector is to earn points by having long chains of cards, and each gold card in the chain doubles the value of the whole chain.

Prospector Rules

To try this out, grab a normal deck of playing cards (like a physical, real deck, not the virtual one we just made). Remove the Jokers and shuffle the remaining 52 cards:

1. Lay out 28 of the cards as shown in the Figure 32.7. Make the bottom three rows of cards face-down, and the top row face-up. The card edges don't need to be touching, but the lower cards do need to be covered by the upper cards. This sets up the initial tableau of cards for the "mine" that our prospector will be excavating.

Figure 32.7. The initial tableau layout for the mine in Prospector



2. The rest of the deck forms a draw pile. Place it above the top row of cards face-down.

3. Draw the top card from the draw deck and place it face-up centered above the top row of cards. This is the target card.

4. Any card that is either exactly one rank above or one rank below the target card may be moved from the tableau onto the target card, making it the new target. Aces and Kings wrap around, so an Ace can be played on a King and vice versa.

5. If a face-down card is no longer covered by a card from a higher row, it can be turned face-up.

6. If none of the face-up cards can be played on the target card, draw a new target card from the draw pile.

7. If the tableau is emptied before the draw pile, you win! (Scoring and gold cards will be saved for the digital version of the game.)

Example of Play

The image in Figure 32.8 shows an example initial layout for Prospector. In the situation shown, the player can initially play either the 9C (9 of Clubs) or the 7S (7 of Spades) onto the 8H.

Figure 32.8. An example initial layout for Prospector



The amber and green numbers show two possible sequences of play. In the amber sequence, the 9C is played, becoming the new target card. This allows the play of 8S, 8D, or 8C. The player chooses 8S because it will then reveal the card that was hidden by 9C and 8S. Then the amber sequence continues with 7S and finally 8C. This results in the layout shown in Figure 32.9.

Figure 32.9. The Prospector example game after the first run



Now, because there are no more valid face-up cards to play from the tableau, the player must draw a card from the draw pile to become the next target card.

Try playing the game a few times to get a feel for it.

Implementing Prospector in Code

As you have seen from playing, Prospector is a pretty simple game, but it's also pretty fun. We can add to that fun later with some nice visuals and scoring tweaks, but for now, let's just get the basic game working.

Laying Out the Mine Tableau

We'll need to implement the same tableau layout for the mine cards in the digital version of Prospector as we did with the paper prototype you just played. To do this, we'll generate some XML code from the layout diagram in Figure 32.7.

1. In Unity, open the LayoutXML.xml file in the Resources folder, and you'll see this layout information. Note that comments in XML are bounded by `<!--` and `-->` is a comment (just like code bounded by `/*` and `*/` in C#).

```
<xml>
   <!-- This file holds info for laying out the Prospector card
game. -->

   <!-- The multiplier is multiplied by the x and y attributes
below. -->
   <!-- This determines how loose or tight the layout is. -->
   <multiplier x="1.25" y="1.5" />

   <!-- In the XML below, id is the number of the card -->
   <!-- x and y set position -->
   <!-- faceup is 1 if the card is face-up -->
   <!-- layer sets the depth layer so cards overlap properly -->
   <!-- hiddenby is the ids of cards that keep a card face-down --
>

   <!-- Layer0, the deepest cards. -->
   <slot id="0" x="-6" y="-5" faceup="0" layer="0"
hiddenby="3,4" />
   <slot id="1" x="0"  y="-5" faceup="0" layer="0"
hiddenby="5,6" />
   <slot id="2" x="6"  y="-5" faceup="0" layer="0"
hiddenby="7,8" />

   <!-- Layer1, the next level. -->
   <slot id="3" x="-7" y="-4" faceup="0" layer="1"
hiddenby="9,10" />
   <slot id="4" x="-5" y="-4" faceup="0" layer="1"
hiddenby="10,11" />
   <slot id="5" x="-1" y="-4" faceup="0" layer="1"
hiddenby="12,13" />
   <slot id="6" x="1"  y="-4" faceup="0" layer="1"
hiddenby="13,14" />
   <slot id="7" x="5"  y="-4" faceup="0" layer="1"
hiddenby="15,16" />
   <slot id="8" x="7"  y="-4" faceup="0" layer="1"
hiddenby="16,17" />
```

```
   <!-- Layer2, the next level. -->
   <slot id="9"  x="-8" y="-3" faceup="0" layer="2"
hiddenby="18,19" />
   <slot id="10" x="-6" y="-3" faceup="0" layer="2"
hiddenby="19,20" />
   <slot id="11" x="-4" y="-3" faceup="0" layer="2"
hiddenby="20,21" />
   <slot id="12" x="-2" y="-3" faceup="0" layer="2"
hiddenby="21,22" />
   <slot id="13" x="0"  y="-3" faceup="0" layer="2"
hiddenby="22,23" />
   <slot id="14" x="2"  y="-3" faceup="0" layer="2"
hiddenby="23,24" />
   <slot id="15" x="4"  y="-3" faceup="0" layer="2"
hiddenby="24,25" />
   <slot id="16" x="6"  y="-3" faceup="0" layer="2"
hiddenby="25,26" />
   <slot id="17" x="8"  y="-3" faceup="0" layer="2"
hiddenby="26,27" />


   <!-- Layer3, the top level. -->
   <slot id="18" x="-9" y="-2" faceup="1" layer="3" />
   <slot id="19" x="-7" y="-2" faceup="1" layer="3" />
   <slot id="20" x="-5" y="-2" faceup="1" layer="3" />
   <slot id="21" x="-3" y="-2" faceup="1" layer="3" />
   <slot id="22" x="-1" y="-2" faceup="1" layer="3" />
   <slot id="23" x="1"  y="-2" faceup="1" layer="3" />
   <slot id="24" x="3"  y="-2" faceup="1" layer="3" />
   <slot id="25" x="5"  y="-2" faceup="1" layer="3" />
   <slot id="26" x="7"  y="-2" faceup="1" layer="3" />
   <slot id="27" x="9"  y="-2" faceup="1" layer="3" />

   <!-- This positions the draw pile and staggers it -->
   <slot type="drawpile" x="6" y="4" xstagger="0.15"
```

```
layer="4"/>

   <!-- This positions the discard pile and target card -->
   <slot type="discardpile" x="0" y="1" layer="5"/>

</xml>
```

As you can see, this has layout information for each of the cards in the tableau (which is formed of `<slot>`s without a `type` attribute) as well as two special slots (that do have `type` attributes), the `drawpile` and `discardpile` types.

2. Now, let's write some code to parse this LayoutXML into useful information. Create a new class named `Layout` in the __Scripts folder and enter the following code:

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

// The SlotDef class is not a subclass of MonoBehaviour, so it doesn't need
// a separate C# file.
[System.Serializable] // This makes SlotDefs visible in the Unity Inspector
                      // pane
public class SlotDef {
   public float        x;
   public float        y;
   public bool         faceUp=false;
   public string       layerName="Default";
   public int          layerID = 0;
   public int          id;
   public List<int>    hiddenBy = new List<int>();
   public string       type="slot";
   public Vector2      stagger;
}
```

```
public class Layout : MonoBehaviour {
   public PT_XMLReader      xmlr; // Just like Deck, this has an
                                  // PT_XMLReader
   public PT_XMLHashtable   xml;  // This variable is for faster
xml access
   public Vector2           multiplier; // The offset of the tableau's
                                  // center
   // SlotDef references
   public List<SlotDef>     slotDefs; // All the SlotDefs for Row0-
Row3
   public SlotDef           drawPile;
   public SlotDef           discardPile;
   // This holds all of the possible names for the layers set by
layerID
   public string[]          sortingLayerNames = new string[] {
"Row0", "RowÊ1", "Row2", "Row3", "Discard", "Draw" };

   // This function is called to read in the LayoutXML.xml file
   public void ReadLayout(string xmlText) {
      xmlr = new PT_XMLReader();
      xmlr.Parse(xmlText);      // The XML is parsed
      xml = xmlr.xml["xml"][0]; // And xml is set as a shortcut to
the XML

      // Read in the multiplier, which sets card spacing
      multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
      multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

      // Read in the slots
      SlotDef tSD;
      // slotsX is used as a shortcut to all the <slot>s
      PT_XMLHashList slotsX = xml["slot"];

      for (int i=0; i<slotsX.Count; i++) {
         tSD = new SlotDef();  // Create a new SlotDef instance
         if (slotsX[i].HasAtt("type")) {
```

```
            // If this <slot> has a type attribute parse it
            tSD.type = slotsX[i].att("type");
         } else {
            // If not, set its type to "slot"; it's a card in the rows
            tSD.type = "slot";
         }
         // Various attributes are parsed into numerical values
         tSD.x = float.Parse( slotsX[i].att("x") );
         tSD.y = float.Parse( slotsX[i].att("y") );
         tSD.layerID = int.Parse( slotsX[i].att("layer") );
         // This converts the number of the layerID into a text
layerName
         tSD.layerName = sortingLayerNames[ tSD.layerID ];
         // The layers are used to make sure that the correct cards
are
         //   on top of the others. In Unity 2D, all of our assets are
         //   effectively at the same Z depth, so the layer is used
         //   to differentiate between them.

         switch (tSD.type) {
            // pull additional attributes based on the type of this
            // <slot>
            case "slot":
               tSD.faceUp = (slotsX[i].att("faceup") == "1");
               tSD.id = int.Parse( slotsX[i].att("id") );
               if (slotsX[i].HasAtt("hiddenby")) {
                  string[] hiding = slotsX[i].att("hiddenby").Split(',
Ê');
                  foreach( string s in hiding ) {
                     tSD.hiddenBy.Add ( int.Parse(s) );
                  }
               }
               slotDefs.Add(tSD);
               break;

            case "drawpile":
               tSD.stagger.x = float.Parse( slotsX[i].att("xstagger")
```

```
);
                drawPile = tSD;
                break;
            case "discardpile":
                discardPile = tSD;
                break;
        }
    }
}
```

At this point, most of the preceding syntax should look familiar to you. The `SlotDef` class is created to store information read in from the XML `<slot>`s in a more accessible way. Then, the `Layout` class is defined, and the `ReadLayout()` method is created, which will take an XML-formatted string as input and turn it into a series of SlotDefs.

3. Open the `Prospector` class and add the following bolded lines:

```
public class Prospector : MonoBehaviour {
    static public Prospector    S;


    [Header("Set in the Unity Inspector")]
    public TextAsset          deckXML;
    public TextAsset          layoutXML;

    [Header("These fields are set dynamically")]
    public Deck               deck;
    public Layout             layout;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }

    void Start () {
        deck = GetComponent<Deck>();     // Get the Deck
```

```
        deck.InitDeck(deckXML.text);     // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards);    // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which
allows
        //   deck.cards to be modified by Deck.Shuffle()

        layout = GetComponent<Layout>();   // Get the Layout
        layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it
    }
}
```

4. Once this is done, you will need to set up a couple of things in Unity. Switch to Unity and select _MainCamera in the Hierarchy. From the menu bar, choose *Component* > *Scripts* > *Layout* to attach a Layout script to _MainCamera (this is just another different way to attach a script to a GameObject). You should now be able to scroll down in the Inspector pane and see the *Layout (Script)* component at the bottom.

5. Find the *Prospector (Script)* component of _MainCamera. You'll see that the public fields `layout` and `layoutXML` have appeared there. Click the target next to `layoutXML` and choose *LayoutXML* from the Assets tab. (You may need to click the *Assets* tab at the top of the *Select TextAsset* window that appeared.)

6. *Save your scene.*

7. Now, press Play. If you select _MainCamera in the Hierarchy and scroll down to the *Layout (Script)* component, you should be able to open the disclosure triangle next to `slotDefs` and see that all of the `<slot>`s have been parsed from the XML.

Working with CardProspector—A Subclass of Card

Before we can position the cards in the tableau, it will be necessary to add some features to the `Card` class that are specific to the Prospector game. Because `Card` and `Deck` are intended to be reused

on other card games, we will choose to create a `CardProspector` class as a subclass of `Card` rather than modifying `Card` directly.

1. Create a new C# script named *CardProspector* and enter this code:

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

// This is an enum, which defines a type of variable that only has a few
// possible named values. The CardState variable type has one of four values:
// drawpile, tableau, target, & discard
public enum CardState {
    drawpile,
    tableau,
    target,
    discard
}

public class CardProspector : Card { // Make sure CardProspector extends Card
    // This is how you use the enum CardState
    public CardState            state = CardState.drawpile;
    // The hiddenBy list stores which other cards will keep this one face down
    public List<CardProspector>   hiddenBy = new List<CardProspector>();
    // The layoutID matches this card to the tableau XML if it's a tableau card
    public int                  layoutID;
    // The SlotDef class stores information pulled in from the LayoutXML <slot>
    public SlotDef              slotDef;
}
```

These extensions to `Card` will handle things like the four types of locations that the card can occupy in the tableau (drawpile, tableau (one of the initial 28 cards in the mine), discard, or target [the active card on top of the discard pile]), the storage of layout information (`slotDef`), and the information that determines when a card should be face-up or face-down (`hiddenBy` and `layoutID`).

Now that this subclass is available, it'll be necessary to convert the cards in the deck from Cards to CardProspectors.

2. To do this, add the following code to the `Prospector` class:

```
public class Prospector : MonoBehaviour {
    ...
    public Layout                layout;
    public List<CardProspector>    drawPile;

    void Awake() { ... }

    void Start () {
        ...
        layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it

        drawPile = ConvertListCardsToListCardProspectors(
deck.cards );
    }

    List<CardProspector>
ConvertListCardsToListCardProspectors(List<Card> IC 
D)  {
        List<CardProspector> lCP = new List<CardProspector>();
        CardProspector tCP;
        foreach( Card tCD in lCD ) {
            tCP = tCD as CardProspector;                    // a
            lCP.Add( tCP );
        }
        return( lCP );
```

```
    }

}
```

3. Once this code is in, try running it and then look at the `drawPile` in the Inspector pane. You'll notice that all the cards in the `drawPile` are null. (You can also look at this happen by placing a break point on the line marked `// a` in the preceding code.). When we try to treat the `Card tCD` as a `CardProspector`, the `as` returns null instead of a converted `Card`. This is because of how object-oriented coding works in C# (see the "On Superclasses and Subclasses" sidebar).

On Superclasses and Subclasses

You're familiar of course with superclasses and subclasses from Chapter 25, "Classes." However, you might wonder why attempting to cast a superclass to a subclass doesn't work.

In Prospector, `Card` is the superclass, and the subclass is `CardProspector`. You could just as easily think of this as a superclass `Animal` and a subclass `Scorpion`. All `Scorpions` are `Animals`, but not all `Animals` are `Scorpions`. You can always refer to a `Scorpion` as "that `Animal`", but you can't refer to any `Animal` as "that `Scorpion`". Along the same lines, a `Scorpion` might have a `Sting()` function, but a `Cow` would not. This is why it's not possible to treat any `Animal` as a `Scorpion`, because trying to call `Sting()` on any other `Animal` might cause an error.

In Prospector, we want to use a bunch of `Cards` that are created by the `Deck` script as if they were `CardProspectors`. This is akin to having a bunch of `Animals` that we want to treat like `Scorpions` (but we've already decided this is impossible). However, it's always possible to refer to a `Scorpion` as an `Animal`, so the solution that we use in Prospector is to make `PrefabCard` have a CardProspector (Script) component instead of just a Card (Script) component. If we just create `Scorpions` from the beginning, and then treat them as `Animals` through several functions (which they can do because `Scorpion` is a

subclass of `Animal`), when we choose to call `Scorpion s = Animal as Scorpion;` later, that will work perfectly because the `Animal` was always secretly a `Scorpion`.

The solution in this case is to make sure that the `CardProspector` was always a `CardProspector` and was just masquerading as a `Card` for all of the code in the `Deck` class.

4. To do this, select PrefabCard in the Project pane, and you'll see that it appears in the Inspector with a *Card (Script)* component.

5. Click the Add Component button and choose *Add Component* > *Scripts* > *CardProspector*. This will add a *CardProspector (Script)* component to the PrefabCard GameObject.

6. To delete the old *Card (Script)* component, click the gear icon in the top-right corner of the *Card (Script)* Inspector and choose *Remove Component* from the pop-up menu.

7. Select _MainCamera from the Hierarchy and play the scene now, you will see that all of the entries in `drawPile` are now full of `CardProspectors` instead of null.

When the Deck script instantiates PrefabCard and gets the `Card` component of it, this will still work perfectly fine because a `CardProspector` can always be referred to as a `Card`. Then, when the `ConvertListCardsToListCardProspectors()` function attemps to call `tCP = tCD as CardProspector;`, it works just fine.

8. Save your scene. You know the drill.

Positioning Cards in the Tableau

Now that everything is ready, it's time to add some code to the `Prospector` class to actually lay out the game:

```
public class Prospector : MonoBehaviour {
    static public Prospector    S;

    [Header("Set in the Unity Inspector")]
    public TextAsset            deckXML;
    public TextAsset            layoutXML;
    public float                xOffset = 3;
    public float                yOffset = -2.5f;
    public Vector3              layoutCenter;


    [Header("These fields are set dynamically")]
    public Deck                 deck;
    public Layout               layout;
    public List<CardProspector>    drawPile;

    public Transform            layoutAnchor;
    public CardProspector       target;
    public List<CardProspector>    tableau;
    public List<CardProspector>    discardPile;

    void Awake() { ... }

    void Start () {
        ...
        drawPile = ConvertListCardsToListCardProspectors( deck.cards
);
        LayoutGame();
    }

    List<CardProspector>
ConvertListCardsToListCardProspectors(List<Card> lCÊD) {
        ...
    }

    // The Draw function will pull a single card from the drawPile
and
```

```
   // return it
   CardProspector Draw() {
      CardProspector cd = drawPile[0]; // Pull the 0th
CardProspector
      drawPile.RemoveAt(0);          // Then remove it from List<>
                                 // drawPile
      return(cd);                    // And return it
   }

   // LayoutGame() positions the initial tableau of cards, a.k.a.
the
   // "mine"
   void LayoutGame() {
      // Create an empty GameObject to serve as an anchor for
the
      // tableau //1
      if (layoutAnchor == null) {
         GameObject tGO = new GameObject("_LayoutAnchor");
         // ^ Create an empty GameObject named _LayoutAnchor
in the
         // Hierarchy
         layoutAnchor = tGO.transform;          // Grab its
Transform
         layoutAnchor.transform.position = layoutCenter;   //
Position it
      }

      CardProspector cp;
      // Follow the layout
      foreach (SlotDef tSD in layout.slotDefs) {
         // ^ Iterate through all the SlotDefs in the layout.slotDefs
as
         // tSD
         cp = Draw(); // Pull a card from the top (beginning) of the
draw
                  // Pile
         cp.faceUp = tSD.faceUp;    // Set its faceUp to the value in
```

```
                                    // SlotDef
            cp.transform.parent = layoutAnchor;  // Make its parent
                                                 // layoutAnchor
            // This replaces the previous parent: deck.deckAnchor,
which
            // appears as _Deck in the Hierarchy when the scene is
playing.
            cp.transform.localPosition = new Vector3(
                layout.multiplier.x * tSD.x,
                layout.multiplier.y * tSD.y,
                -tSD.layerID );
            // ^ Set the localPosition of the card based on slotDef
            cp.layoutID = tSD.id;
            cp.slotDef = tSD;
            cp.state = CardState.tableau;
            // CardProspectors in the tableau have the state
            // CardState.tableau

            tableau.Add(cp); // Add this CardProspector to the List<>
                             // tableau
        }
    }

}
```

When you play this, you will see that the cards are indeed laid out in the mine tableau layout described in LayoutXML.xml, and the right ones are face-up and face-down, but there are some serious issues with layering (see Figure 32.10).

Figure 32.10. Cards are laid out, but there are several layering issues (and remaining cards from the initial grid layout that existed previously)



Hold the Option/Alt key down and use the left mouse button in the Scene window to look around, and you will see that when using

Unity's 2D tools, the distance of the 2D object to the camera has nothing to do with the depth sorting of the objects (that is, which objects are rendered on top of each other). We actually got a little lucky with the construction of the cards because we built them from back to front so that all the pips and decorators showed up on top of the card face. However, here we're going to have to be more careful about it for the layout of the game to avoid the problems you can see in Figure 32.10.

Unity 2D has two methods of dealing with depth sorting:

• **Sorting Layers:** Sorting layers are used to group 2D objects. Everything in a lower sorting layer is rendered behind everything in a higher sorting layer. Each SpriteRenderer component has a `sortingLayerName` string variable that can be set to the name of a sorting layer.

• **Sorting order:** Each SpriteRenderer component also has a `sortingOrder` variable. This is used to position elements within each sorting layer relative to each other.

In the absence of sorting layers and sortingOrder, sprites are often rendered from back to front in the order that they were created, but this is not at all reliable.

Setting Up Sorting Layers

To set up sorting layer, follow these steps:

1. From the menu bar, choose *Edit* > *Project Settings* > *Tags and Layers*. You've used tags and layers for physics layers and tags before, but we haven't yet touched sorting layers.

2. Open the disclosure triangle next to *Sorting Layers*, and enter the layers as shown in Figure 32.11. You will need to click the + button at the bottom-right of the list to add new sorting layers.

Figure 32.11. The sorting layers required for Prospector



Because SpriteRenderers and depth sorting are something that will be necessary for any card game built using our code base, the code to deal with depth sorting should be added to the `Card` class.

3. Open the Card script and add the following code:

```
public class Card : MonoBehaviour {
    ...
    public CardDefinition   def; // Parsed from DeckXML.xml

    // List of the SpriteRenderer Components of this GameObject
and its
    // children
    public SpriteRenderer[]       spriteRenderers;

    void Start() {
        SetSortOrder(0);  // Ensures that the card starts properly
depth
                          // sorted
    }

    // If spriteRenderers is not yet defined, this function defines it
    public void PopulateSpriteRenderers() {
        // If spriteRenderers is null or empty
        if (spriteRenderers == null || spriteRenderers.Length == 0) {
            // Get SpriteRenderer Components of this GameObject
and its
            // children
            spriteRenderers =
GetComponentsInChildren<SpriteRenderer>();
        }
    }

    // Sets the sortingLayerName on all SpriteRenderer
Components
```

```
    public void SetSortingLayerName(string tSLN) {
        PopulateSpriteRenderers();

        foreach (SpriteRenderer tSR in spriteRenderers) {
            tSR.sortingLayerName = tSLN;
        }
    }

    // Sets the sortingOrder of all SpriteRenderer Components
    public void SetSortOrder(int sOrd) {
        PopulateSpriteRenderers();

        // The white background of the card is on bottom (sOrd)
        // On top of that are all the pips, decorators, face, etc.
(sOrd+1)
        // The back is on top so that when visisble, it covers the
rest
        // (sOrd+2)

        // Iterate through all the spriteRenderers as tSR
        foreach (SpriteRenderer tSR in spriteRenderers) {
            if (tSR.gameObject == this.gameObject) {
                // If the gameObject is this.gameObject, it's the
background
                tSR.sortingOrder = sOrd;  // Set it's order to sOrd
                continue;  // And continue to the next iteration of the
loop

            }
            // Each of the children of this GameObject are named
            // switch based on the names
            switch (tSR.gameObject.name) {
                case "back": // if the name is "back"
                    tSR.sortingOrder = sOrd+2;
                    // ^ Set it to the highest layer to cover everything
                    // else
                    break;
```

```
            case "face": // if the name is "face"
            default:     // or if it's anything else
               tSR.sortingOrder = sOrd+1;
               // ^ Set it to the middle layer to be above the
               // background
               break;
         }
      }
   }

   public bool faceUp { ... }

}
```

4. Now, Prospector needs one line added to make sure that the cards in the initial mine layout are placed in the proper sorting layer. Add this line near the end of `Prospector.LayoutGame()`:

```
public class Prospector : MonoBehaviour {
   ...
   // LayoutGame() positions the initial tableau of cards, the "mine"
   void LayoutGame() {
      ...
      foreach (SlotDef tSD in layout.slotDefs) {
         ...
         cp.state = CardState.tableau;
         // CardProspectors in the tableau have the state
CardState.tableau

         cp.SetSortingLayerName(tSD.layerName); // Set the
sorting layers

         tableau.Add(cp); // Add this CardProspector to the List<>
                  // tableau
      }
   }
}
```

Now, when you run the scene, you'll see that the cards stack properly on top of each other in the mine.

Implementing Game Logic

Before we move cards into place for the draw pile, let's start by delineating the possible actions that can happen in the game:

1. If the target card is replaced by any other card, the replaced target card then moves to the discard pile.

2. A card can move from the drawPile to become the target card.

3. A card that is one higher or one lower than the target card can move to become the target card.

4. If a face-down card has no more cards hiding it, it becomes face-up.

5. The game is over when either the mine is empty (win) or the draw pile is empty and there are no more possible plays (loss).

Actions number 2 and 3 here are the possible move actions, where a card is physically moved, and numbers 1, 4, and 5 look like passive actions that happen as a result of either action 2 or 3.

Making Cards Clickable

Because all of these actions are instigated by a click on one of the cards, we first need to make the cards clickable.

1. This is something that will be needed for every card game, so add the following method to the `Card` class:

```
public class Card : MonoBehaviour {
    ...
    public bool faceUp { ... }
```

```
    // Virtual methods can be overridden by subclass methods
with the same name
    virtual public void OnMouseUpAsButton() {
        print (name);  // When clicked, this outputs the card name
    }
}
```

Now, when you press Play, you can click any card in the scene, and it will output its name.

2. However, in Prospector, we need card clicks to do more than that, so add the following method to the end of the `CardProspector` class:

```
public class CardProspector : Card { // Make sure CardProspector
extends

                                // Card
    ...
    public SlotDef              slotDef;

    // This allows the card to react to being clickced
    override public void OnMouseUpAsButton() {
        // Call the CardClicked method on the Prospector singleton
        Prospector.S.CardClicked(this);
        // Also call the base class (Card.cs) version of this method
        base.OnMouseUpAsButton();
    }
}
```

3. The `CardClicked` method still must be written in the Prospector script (which is why it's currently red in the code we just typed), but first, we'll need a few helper functions. Add the MoveToDiscard(), MoveToTarget(), and UpdateDrawPile() methods to the end of the Prospector class.

```
public class Prospector : MonoBehaviour {
    ...
```

```
    void LayoutGame() { ... }

    // Moves the current target to the discardPile
    void MoveToDiscard(CardProspector cd) {
        // Set the state of the card to discard
        cd.state = CardState.discard;
        discardPile.Add(cd);  // Add it to the discardPile List<>
        cd.transform.parent = layoutAnchor; // Update its transform
parent
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID+0.5f );
        // ^ Position it on the discardPile
        cd.faceUp = true;
        // Place it on top of the pile for depth sorting
        cd.SetSortingLayerName(layout.discardPile.layerName);
        cd.SetSortOrder(-100+discardPile.Count);
    }

    // Make cd the new target card
    void MoveToTarget(CardProspector cd) {
        // If there is currently a target card, move it to discardPile
        if (target != null) MoveToDiscard(target);
        target = cd; // cd is the new target
        cd.state = CardState.target;
        cd.transform.parent = layoutAnchor;
        // Move to the target position
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID );
        cd.faceUp = true; // Make it face-up
        // Set the depth sorting
        cd.SetSortingLayerName(layout.discardPile.layerName);
        cd.SetSortOrder(0);
    }
```

```
    // Arranges all the cards of the drawPile to show how many
are left
    void UpdateDrawPile() {
        CardProspector cd;
        // Go through all the cards of the drawPile
        for (int i=0; i<drawPile.Count; i++) {
            cd = drawPile[i];
            cd.transform.parent = layoutAnchor;
            // Position it correctly with the layout.drawPile.stagger
            Vector2 dpStagger = layout.drawPile.stagger;
            cd.transform.localPosition = new Vector3(
                layout.multiplier.x * (layout.drawPile.x +
i*dpStagger.x),
                layout.multiplier.y * (layout.drawPile.y +
i*dpStagger.y),
                -layout.drawPile.layerID+0.1f*i );
            cd.faceUp = false; // Make them all face-down
            cd.state = CardState.drawpile;
            // Set depth sorting
            cd.SetSortingLayerName(layout.drawPile.layerName);
            cd.SetSortOrder(-10*i);
        }
    }

}
```

4. Next, let's add some code to the end of Prospector.LayoutGame() to draw the initial target card and arrange the drawPile. Additionallt, we'll add the beginning of the CardClicked() method, which will handle all clicks on CardProspectors. For now, CardClicked() will only handle moving a card from the drawPile to the target (letter b from the actions listed earlier), but we'll expand on it soon.

```
public class Prospector : MonoBehaviour {
    ...
    // LayoutGame() positions the initial tableau of cards, a.k.a. the
```

```
"mine"
   void LayoutGame() {
      ...
      foreach (SlotDef tSD in layout.slotDefs) {
         ...
         tableau.Add(cp); // Add this CardProspector to the List<>
                          // tableau
      }

      // Set up the initial target card
      MoveToTarget(Draw ());

      // Set up the Draw pile
      UpdateDrawPile();

   }

   // CardClicked is called any time a card in the game is clicked
   public void CardClicked(CardProspector cd) {
      // The reaction is determined by the state of the clicked
card
      switch (cd.state) {
         case CardState.target:
            // Clicking the target card does nothing
            break;
         case CardState.drawpile:
            // Clicking any card in the drawPile will draw the next
card
            MoveToDiscard(target); // Moves the target to the
                                   // discardPile
            MoveToTarget(Draw());  // Moves the next drawn card
to the
                                   // target
            UpdateDrawPile();      // Restacks the drawPile
            break;
         case CardState.tableau:
            // Clicking a card in the tableau will check if it's a valid
```

```
            // play
            break;
        }
    }

    // Moves the current target to the discardPile
    void MoveToDiscard(CardProspector cd) { ... }
    ...
}
```

Now, when you play the scene, you will see that you can click on the drawPile (in the top-right corner of the screen) to draw a new target card. We're getting close to having a game now!

Matching Cards from the Mine

To make the card in the mine work, we need to have a little code that checks to make sure that the clicked card is either one higher or one lower than the target card (and, of course, also handles A-to-King wraparound).

1. Add these bolded lines to the Prospector script:

```
public class Prospector : MonoBehaviour {
    ...

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
        ...
        case CardState.tableau:
            // Clicking a card in the tableau will check if it's a valid
            // play
            bool validMatch = true;
            if (!cd.faceUp) {
                // If the card is face-down, it's not valid
```

```
            validMatch = false;
        }
        if (!AdjacentRank(cd, target)) {
            // If it's not an adjacent rank, it's not valid
            validMatch = false;
        }
        if (!validMatch) return; // return if not valid
        // Yay! It's a valid card.
        tableau.Remove(cd); // Remove it from the tableau List
        MoveToTarget(cd);   // Make it the target card
        break;
      }
   }

   ...
   void UpdateDrawPile() { ... }

   // Return true if the two cards are adjacent in rank (A & K
wrap around)
   public bool AdjacentRank(CardProspector c0,
CardProspector c1) {
      // If either card is face-down, it's not adjacent.
      if (!c0.faceUp || !c1.faceUp) return(false);

      // If they are 1 apart, they are adjacent
      if (Mathf.Abs(c0.rank - c1.rank) == 1) {
         return(true);
      }
      // If one is A and the other King, they're adjacent
      if (c0.rank ==  1 && c1.rank == 13) return(true);
      if (c0.rank == 13 && c1.rank ==  1) return(true);

      // Otherwise, return false
      return(false);
   }
}
```

Now, you can play the game and actually play the top row correctly! However, as you play more, you'll notice that the face-down cards are never flipping to face-up. This is what the `List<CardProspector> CardProspector.hiddenBy` is for. We have the information about which cards hide others in `List<int> SlotDef.hiddenBy`, but we need to be able to convert from the integer IDs in `SlotDef.hiddenBy` to the actual CardProspectors that have that ID.

2. Add this code to `Prospector` to do so:

```
public class Prospector : MonoBehaviour {
    ...
    // LayoutGame() positions the initial tableau of cards, the "mine"
    void LayoutGame() {
        ...
        // Follow the layout
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
        }

        // Set which cards are hiding others
        foreach (CardProspector tCP in tableau) {
            foreach( int hid in tCP.slotDef.hiddenBy ) {
                cp = FindCardByLayoutID(hid);
                tCP.hiddenBy.Add(cp);
            }
        }

        // Set up the target card
        MoveToTarget(Draw ());
        ...
    }

    // Convert from the layoutID int to the CardProspector with
that ID
    CardProspector FindCardByLayoutID(int layoutID) {
        foreach (CardProspector tCP in tableau) {
```

```
        // Search through all cards in the tableau List<>
        if (tCP.layoutID == layoutID) {
            // If the card has the same ID, return it
            return( tCP );
        }
    }
    // If it's not found, return null
    return( null );
}

// This turns cards in the Mine face-up or face-down
void SetTableauFaces() {
    foreach( CardProspector cd in tableau ) {
        bool fup = true; // Assume the card will be face-up
        foreach( CardProspector cover in cd.hiddenBy ) {
            // If either of the covering cards are in the tableau
            if (cover.state == CardState.tableau) {
                fup = false; // then this card is face-down
            }
        }
        cd.faceUp = fup; // Set the value on the card
    }
}

// CardClicked is called any time a card in the game is clicked
public void CardClicked(CardProspector cd) {
    // The reaction is determined by the state of the clicked card
    switch (cd.state) {
    ...
    case CardState.tableau:
        ...
        MoveToTarget(cd);   // Make it the target card
        SetTableauFaces();  // Update tableau card face-ups
        break;
    }
}
```

```
    ...
}
```

Now, an entire round of the game is playable!

3. Next up, making the game know when it's over. This only needs to be checked once after each time the player has clicked a card, so the check will be called from the end of `Prospector.CardClicked()`. Add the following to the `Prospector` class:

```
public class Prospector : MonoBehaviour {
    ...

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
            ...
        }
        // Check to see whether the game is over or not
        CheckForGameOver();
    }

    // Test whether the game is over
    void CheckForGameOver() {
        // If the tableau is empty, the game is over
        if (tableau.Count==0) {
            // Call GameOver() with a win
            GameOver(true);
            return;
        }
        // If there are still cards in the draw pile, the game's not
over
        if (drawPile.Count>0) {
            return;
        }
        // Check for remaining valid plays
```

```
        foreach ( CardProspector cd in tableau ) {
            if (AdjacentRank(cd, target)) {
                // If there is a valid play, the game's not over
                return;
            }
        }
        // Since there are no valid plays, the game is over
        // Call GameOver with a loss
        GameOver (false);
    }

    // Called when the game is over. Simple for now, but
expandable
    void GameOver(bool won) {
        if (won) {
            print ("Game Over. You won! :)");
        } else {
            print ("Game Over. You Lost. :(");
        }
        // Reload the scene, resetting the game
        SceneManager.LoadScene("__Prospector_Scene_0");
    }

    void MoveToDiscard() { ... }
    ...
}
```

Now the game is playable and repeatable, and it knows when it has won or lost. Next up, it's time to add some scoring.

Adding Scoring to Prospector

The original card game of Prospector (or Tri-Peaks, on which it was based) had no scoring mechanism beyond the player winning or losing. But as a digital game, it's really helpful to have scores and a high score so that players have a reason to keep playing (to beat their high score).

Ways to Earn Points in the Game

We will implement several ways to earn points in Prospector:

1. Moving a card from the mine to the target card earns 1 point.

2. Every subsequent card removed from the mine without drawing from the draw pile increases the points awarded per card by 1, so a *run* of five cards removed without a draw would be worth 1, 2, 3, 4, and 5 points, respectively, for a total of 15 points for the run (1 + 2 + 3 + 4 + 5 = 15).

3. If the player wins the round, she carries her score on to the next round. Whenever a round is lost, her score for all rounds is totaled and checked against the high score list.

4. The number of points earned for a run will double for each special gold card in the run. If two of the cards in the example run from #2 were gold, then the run would be worth 60 points (15 x 2 x 2 = 60).

The scoring will be handled by the `Prospector` class because it is aware of all the conditions that could contribute to the score. We will also create a script named *Scoreboard* to handle all the visual elements of showing the score to the player.

We'll implement letters a through c in this chapter, and I'll leave letter d for you to implement on your own later.

Making the Chain Scoring Work

For now, let's just make some changes to Prospector to track the score. Because we're enabling runs and the doubling of the value for the run, it makes sense to store the score for the run separately and then apply that to the total score for the round once the run has been ended (by drawing a card from the drawPile). Add the following code to Prospector to implement this:

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

// An enum to handle all the possible scoring events
public enum ScoreEvent {
    draw,
    mine,
    mineGold,
    gameWin,
    gameLoss
}

public class Prospector : MonoBehaviour {
    static public Prospector     S;
    static public int          SCORE_FROM_PREV_ROUND = 0;
    static public int          HIGH_SCORE = 0;

    ...
    public List<CardProspector>  discardPile;

    // Fields to track score info
    public int                 chain = 0;
    public int                 scoreRun = 0;
    public int                 score = 0;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
        // Check for a high score in PlayerPrefs
        if (PlayerPrefs.HasKey ("ProspectorHighScore")) {
            HIGH_SCORE =
PlayerPrefs.GetInt("ProspectorHighScore");
        }
        // Add the score from last round, which will be >0 if it was a
win
        score += SCORE_FROM_PREV_ROUND;
```

```
        // And reset the SCORE_FROM_PREV_ROUND
        SCORE_FROM_PREV_ROUND = 0;
    }

    ...

    // CardClicked is called any time a card in the game is clicked
    public void CardClicked(CardProspector cd) {
        // The reaction is determined by the state of the clicked card
        switch (cd.state) {
        ...
        case CardState.drawpile:
            ...
            UpdateDrawPile();      // Restacks the drawPile
            ScoreManager(ScoreEvent.draw);
            break;
        case CardState.tableau:
            ...
            SetTableauFaces();  // Update tableau card face-ups
            ScoreManager(ScoreEvent.mine);
            break;
        }
        ...
    }

    ...

    // Called when the game is over. Simple for now, but expandable
    void GameOver(bool won) {
        if (won) {
            // print ("Game Over. You won! :");   // Comment out this
line
            ScoreManager(ScoreEvent.gameWin);
        } else {
            // print ("Game Over. You Lost. :");  // Comment out this
line
            ScoreManager(ScoreEvent.gameLoss);
```

```
        }
        // Reload the scene, resetting the game
        SceneManager.LoadScene("__Prospector_Scene_0");
    }

    // ScoreManager handles all of the scoring
    void ScoreManager(ScoreEvent sEvt) {
        switch (sEvt) {
        // Same things need to happen whether it's a draw, a win, or
a loss
        case ScoreEvent.draw: // Drawing a card
        case ScoreEvent.gameWin: // Won the round
        case ScoreEvent.gameLoss: // Lost the round
            chain = 0;        // resets the score chain
            score += scoreRun; // add scoreRun to total score
            scoreRun = 0;      // reset scoreRun
            break;
        case ScoreEvent.mine: // Remove a mine card
            chain++;          // increase the score chain
            scoreRun += chain; // add score for this card to run
            break;
        }

        // This second switch statement handles round wins and
losses
        switch (sEvt) {
        case ScoreEvent.gameWin:
            // If it's a win, add the score to the next round
            // static fields are NOT reset by
SceneManager.LoadScene()
            Prospector.SCORE_FROM_PREV_ROUND = score;
            print ("You won this round! Round score: "+score);
            break;
        case ScoreEvent.gameLoss:
            // If it's a loss, check against the high score
            if (Prospector.HIGH_SCORE <= score) {
                print("You got the high score! High score: "+score);
```

```
            Prospector.HIGH_SCORE = score;
            PlayerPrefs.SetInt("ProspectorHighScore", score);
        } else {
            print ("Your final score for the game was: "+score);
        }
        break;
    default:
        print ("score:
"+score+"  scoreRun:"+scoreRun+"  chain:"+chain);
        break;
    }
  }

  // Moves the current target to the discardPile
  void MoveToDiscard(CardProspector cd) { ... }
  ...
}
```

Now, as you play the game, you'll see little notes in the Console pane that tell you your score. This works fine for testing, but let's make things look a little better for our players.

Showing the Score to the Players

For this game, we'll make a couple reusable components that can show the score. One will be a Scoreboard class that will handle manage all of the score display. The other will be FloatingScore, which will be an on-screen number that can move around the screen on its own. We'll also make use of Unity's `SendMessage()` feature, which can call a method by name with one parameter on any GameObject:

1. Create a new C# script in the __Scripts folder named *FloatingScore* and enter this code:

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

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

// An enum to track the possible states of a FloatingScore
public enum FSState {
   idle,
   pre,
   active,
   post
}

// FloatingScore can move itself on screen following a Bézier curve
public class FloatingScore : MonoBehaviour {
   public FSState        state = FSState.idle;

   public int            _score = 0; // The score field
   public string         scoreString;

   // The score property also sets scoreString when set
   public int score {
      get {
         return(_score);
      }
      set {
         _score = value;
         scoreString = Utils.AddCommasToNumber(_score);
         GetComponent<Text>().text = scoreString;
      }
   }

   public List<Vector2>   bezierPts; // Bézier points for movement
   public List<float>     fontSizes; // Bézier points for font scaling
   public float          timeStart = -1f;
   public float          timeDuration = 1f;
   public string         easingCuve = Easing.InOut; // Uses Easing
```

in

```
                                        // Utils.cs

    // The GameObject that will receive the SendMessage when
this is done
    // moving
    public GameObject      reportFinishTo = null;

    private RectTransform  rectTrans;

    // Set up the FloatingScore and movement
    // Note the use of parameter defaults for eTimeS & eTimeD
    public void Init(List<Vector2> ePts, float eTimeS = 0, float
eTimeD = 1) {
        rectTrans = GetComponent<RectTransform>();
        rectTrans.anchoredPosition = Vector2.zero;

        bezierPts = new List<Vector2>(ePts);

        if (ePts.Count == 1) { // If there's only one point
            // ...then just go there.
            transform.position = ePts[0];
            return;
        }

        // If eTimeS is the default, just start at the current time
        if (eTimeS == 0) eTimeS = Time.time;
        timeStart = eTimeS;
        timeDuration = eTimeD;

        state = FSState.pre; // Set it to the pre state, ready to start
                             // moving
    }

    public void FSCallback(FloatingScore fs) {
        // When this callback is called by SendMessage,
        // add the score from the calling FloatingScore
```

```
    score += fs.score;
}

// Update is called once per frame
void Update () {
    // If this is not moving, just return
    if (state == FSState.idle) return;

    // Get u from the current time and duration
    // u ranges from 0 to 1 (usually)
    float u = (Time.time - timeStart)/timeDuration;
    // Use Easing class from Utils to curve the u value
    float uC = Easing.Ease (u, easingCuve);
    if (u<0) { // If u<0, then we shouldn't move yet.
        state = FSState.pre;
        // Move to the initial point
        transform.position = bezierPts[0];
    } else {
        if (u>=1) { // If u>=1, we're done moving
            uC = 1; // Set uC=1 so we don't overshoot
            state = FSState.post;
            if (reportFinishTo != null) { // If there's a callback
                                          // GameObject
                // Use SendMessage to call the FSCallback method
                // with this as the parameter.
                reportFinishTo.SendMessage("FSCallback", this);
                // Now that the message has been sent,
                // destroy this gameObject
                Destroy (gameObject);
            } else { // If there is nothing to callback,
                // then don't destroy this. Just let it stay still.
                state = FSState.idle;
            }
        } else {
            // 0<=u<1, which means that this is active and moving
            state = FSState.active;
        }
```

```
            // Use Bézier curve to move this to the right point
            Vector2 pos = Utils.Bezier(uC, bezierPts);
            // RectTransform anchors can be used to position UI
            // objects relative to total size of the screen
            rectTrans.anchorMin = rectTrans.anchorMax = pos;
            if (fontSizes != null && fontSizes.Count>0) {
               // If fontSizes has values in it
               // ...then adjust the fontSize of this GUIText
               int size = Mathf.RoundToInt( Utils.Bezier(uC, fontSizes)
);
               GetComponent<Text>().fontSize = size;
            }
         }
      }
```

2. Create a new C# script in the __Scripts folder named *Scoreboard* and enter this code into it:

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

// The Scoreboard class manages showing the score to the
player
public class Scoreboard : MonoBehaviour {
   public static Scoreboard S; // The singleton for Scoreboard

   [Header("Set in the Unity Inspector")]
   public GameObject    prefabFloatingScore;

   [Header("These fields are set dynamically")]
   public int           _score = 0;
   public string        _scoreString;

   private Transform    canvasTrans;
```

```
    // The score property also sets the scoreString
    public int score {
        get {
            return(_score);
        }
        set {
            _score = value;
            scoreString = Utils.AddCommasToNumber(_score);
        }
    }

    // The scoreString property also sets the Text.text
    public string scoreString {
        get {
            return(_scoreString);
        }
        set {
            _scoreString = value;
            GetComponent<Text>().text = _scoreString;
        }
    }

    void Awake() {
        S = this;
        canvasTrans = transform.parent;
    }

    // When called by SendMessage, this adds the fs.score to
this.score
    public void FSCallback(FloatingScore fs) {
        score += fs.score;
    }

    // This Instantiates a new FloatingScore GameObject and
initializes it.
    // It also returns a pointer to the FloatingScore created so that
```

```
the
    //  calling function can do more with it (like set fontSizes, and
so on)
    public FloatingScore CreateFloatingScore(int amt,
List<Vector2> pts) {
        GameObject go = Instantiate(prefabFloatingScore) as
GameObject;
        go.transform.SetParent( canvasTrans );
        FloatingScore fs = go.GetComponent<FloatingScore>();
        fs.score = amt;
        fs.reportFinishTo = this.gameObject; // Set fs to call back to
this
        fs.Init(pts);
        return(fs);
    }

}
```

Now, you need to make the GameObjects for both the Scoreboard and the FloatingScore.

Making the FloatingScore GameObject Prefab

We start with FloatingScore GameObject:

1. In Unity, from the menu bar, choose *GameObject* > *UI* > *Text*. Rename the *Text* GameObject to *PrefabFloatingScore*.

2. Before changing any of the settings on PrefabFloatingScore, make sure that the aspect ratio of your Game pane is set to *iPad Wide (1024x768)*. This will make sure that your settings and mine agree.

3. Give PrefabFloatingScore the settings shown in Figure 32.12.

Figure 32.12. The settings for PrefabFloatingScore



4. Now attach the script FloatingScore to the GameObject PrefabFloatingScore (by dragging the script onto FloatingScore in the Hierarchy).

5. Convert PrefabFloatingScore to a prefab by dragging it from the Hierarchy into the _Prefabs folder in the Project pane.

6. Finally, delete the instance of PrefabFloatingScore that remains in the Hierarchy pane.

Making the Scoreboard GameObject

Next we make the Scoreboard GameObject:

1. To make the scoreboard, create another Text GameObject in the scene (*GameObject* > *UI* > *Text*).

2. Rename this GameObject to *Scoreboard*.

3. Attach the Scoreboard C# script to the Scoreboard GameObject and give Scoreboard the settings shown in Figure 32.13. This includes dragging the PrefabFloatingScore prefab from the _Prefabs folder into the public `prefabFloatingScore` field of the *Scoreboard (Script)* component.

Figure 32.13. The settings for Scoreboard



4. Now, all you need to do is make a few changes to the `Prospector` class to incorporate the new code and GameObjects. Add the following bolded code to the `Prospector` class:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in the Unity Inspector")]
    ...
    public Vector3              layoutCenter;
```

```
    public Vector2              fsPosMid  = new Vector2( 0.5f, 0.90f );
    public Vector2              fsPosRun  = new Vector2( 0.5f, 0.75f
);
    public Vector2              fsPosMid2 = new Vector2( 0.4f, 1.0f  );
    public Vector2              fsPosEnd  = new Vector2( 0.5f, 0.95f
);

    [Header("These fields are set dynamically")]
    ...

    // Fields to track score info
    public int                  chain = 0;
    public int                  scoreRun = 0;
    public int                  score = 0;
    public FloatingScore        fsRun;

    void Awake() { ... }

    void Start () {
        Scoreboard.S.score = score;

        deck = GetComponent<Deck>(); // Get the Deck
        ...
    }

    ...

    // ScoreManager handles all of the scoring
    void ScoreManager(ScoreEvent sEvt) {
        List<Vector2> fsPts;
        switch (sEvt) {
            // Same things need to happen whether it's a draw, a win, or
a
            // loss
            case ScoreEvent.draw: // Drawing a card
            case ScoreEvent.gameWin: // Won the round
            case ScoreEvent.gameLoss: // Lost the round
```

```
            chain = 0;        // resets the score chain
            score += scoreRun; // add scoreRun to total score
            scoreRun = 0;      // reset scoreRun

            // Add fsRun to the Scoreboard score
            if (fsRun != null) {
               // Create points for the Bézier curve
               fsPts = new List<Vector2>();
               fsPts.Add( fsPosRun );
               fsPts.Add( fsPosMid2 );
               fsPts.Add( fsPosEnd );
               fsRun.reportFinishTo = Scoreboard.S.gameObject;
               fsRun.Init(fsPts, 0, 1);
               // Also adjust the fontSize
               fsRun.fontSizes = new List<float>(new float[]
{28,36,4})Image;
               fsRun = null; // Clear fsRun so it's created again
            }
            break;

         case ScoreEvent.mine: // Remove a mine card
            chain++;          // increase the score chain
            scoreRun += chain; // add score for this card to run

            // Create a FloatingScore for this score
            FloatingScore fs;
            // Move it from the mousePosition to fsPosRun
            Vector2 p0 = Input.mousePosition;
            p0.x /= Screen.width;
            p0.y /= Screen.height;
            fsPts = new List<Vector2>();
            fsPts.Add( p0 );
            fsPts.Add( fsPosMid );
            fsPts.Add( fsPosRun );
            fs = Scoreboard.S.CreateFloatingScore(chain,fsPts);
            fs.fontSizes = new List<float>(new float[] {4,50,28});
            if (fsRun == null) {
```

```
                fsRun = fs;
                fsRun.reportFinishTo = null;
            } else {
                fs.reportFinishTo = fsRun.gameObject;
            }
            break;

        }
        ...
    }
    ...
}
```

Now when you play the game, you should see the score flying around. This is actually pretty important because it helps your players understand where the score is coming from and helps reveal the mechanics of the game to them through play (rather than requiring them to read instructions).

Adding Some Art to the Game

Let's add some theming to the game by adding a background. In the Materials folder that you imported at the beginning of the project are a PNG named *ProspectorBackground* and a material named *ProspectorBackground Mat*. These are already set up for you, since you learned how to do so in previous chapters.

1. In Unity, add a quad to the scene (*GameObject* > *3D Object* > *Quad*).

2. Drag the *ProspectorBackground Mat* from the Materials folder onto the quad.

3. Rename the quad *ProspectorBackground* and set its transform as follows:

ProspectorBackground (Quad) P:[0,0,0] R:[0,0,0] S:[26.667,20,1]

Because _MainCamera's orthographic size is 10, that means that it is 10 units between the center of the screen and the nearest edge (which in this case is the top and bottom), for a total height of 20 units visible on screen. The ProspectorBackground quad is 20 units high (y-scale) because of this. And, because the screen is at a 4:3 aspect ratio, 20 / 3 * 4 = 26.667 is the width (x-scale) that we need to set the background to.

When you play the game now, it should look something like Figure 32.14.

Figure 32.14. The Prospector game with a background



Announcing the Beginning and End of Rounds

I'm sure you've noticed that the rounds of the game end rather abruptly. Let's do something about that. First off, we'll delay the actual reloading of the level using an `Invoke()` function. Add the following bolded code to Prospector:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in the Unity Inspector")]
    ...
    public Vector2              fsPosEnd  = new Vector2( 0.5f, 0.95f );
    public float                reloadDelay = 1f; // Time delay between
rounds

    [Header("These fields are set dynamically")]

    ...

    // Called when the game is over. Simple for now, but expandable
    void GameOver(bool won) {
        if (won) {
```

```
            // print ("Game Over. You won! :)");
            ScoreManager(ScoreEvent.gameWin);
        } else {
            // print ("Game Over. You Lost. :(");
            ScoreManager(ScoreEvent.gameLoss);
        }
        // Reload the scene in reloadDelay seconds
        // This will give the score a moment to travel
        Invoke ("ReloadLevel", reloadDelay);                    // a
        // SceneManager.LoadScene("__Prospector_Scene_0"); //
Now commented out!
    }

    void ReloadLevel() {
        // Reload the scene, resetting the game
        SceneManager.LoadScene("__Prospector_Scene_0");
    }

    // ScoreManager handles all of the scoring
    void ScoreManager(ScoreEvent sEvt) { ... }
    ...
}
```

a. The Invoke() command works by calling a function named `ReloadLevel` in `reloadDelay` seconds. This is similar to how `SendMessage()` works, but it does so with a delay. Now when you play the game, it will wait for one full second before the game reloads.

Giving the Player Feedback on Her Score

We also want to tell the player how she did at the end of each round.

1. Add a new UI Text to the scene: Select *Canvas* in the Hierarchy and from the menu bar choose *GameObject* > *UI* > *Text*.

2. Rename the *Text* to *GameOver* and give it the settings shown on the left side of Figure 32.15.

Figure 32.15. The settings for the GameOver and RoundResult UI Texts



3. Add another UI Text to the scene: Right-click on *Canvas* in the Hierarchy and choose > *UI* > *Text* from the pop-up menu.

4. Rename this *Text* to *RoundResult* and give it the settings shown on the right side of Figure 32.15.

5. Add the third UI Text as a child of Canvas and name it *HighScore*.

6. Give HighScore the settings shown in Figure 32.16.

Figure 32.16. The settings for the HighScore UI Text



The numbers in these settings were determined by trial and error, and you should feel free to adjust them as you see fit. It should nestle the high score right above the sign on the right.

7. To make these UI Texts functional, add the following bolded code to the `Prospector` class:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in the Unity Inspector")]
    ...
    public float              reloadDelay = 1f; // The delay between
                                                // rounds
    public Text               gameOverText, roundResultText,
highScoreT  ext;

    [Header("These fields are set dynamically")]
    ...
```

```
    void Awake() {
        ...
        SCORE_FROM_PREV_ROUND = 0;

        SetUpUITexts();
    }

    void SetUpUITexts() {
        // Set up the HighScore UI Text
        GameObject go = GameObject.Find("HighScore");
        if (go != null) {
            highScoreText = go.GetComponent<Text>();
        }
        string hScore = "High Score:
"+Utils.AddCommasToNumber(HIGH_SCORE);
        go.GetComponent<Text>().text = hScore;

        // Set up the UI Texts that show at the end of the round
        go = GameObject.Find ("GameOver");
        if (go != null) {
            gameOverText = go.GetComponent<Text>();
        }

        go = GameObject.Find ("RoundResult");
        if (go != null) {
            roundResultText = go.GetComponent<Text>();
        }

        // Make the end of round texts invisible
        ShowResultsUI( false );
    }

    void ShowResultsUI(bool show) {
        gameOverText.gameObject.SetActive(show);
        roundResultText.gameObject.SetActive(show);
    }
```

```
    ...

    // ScoreManager handles all of the scoring
    void ScoreManager(ScoreEvent sEvt) {
        List<Vector2> fsPts;
        switch (sEvt) {
            // Same things need to happen whether it's a draw, a win, or
a
            // loss
            ...
        }

        // This second switch statement handles round wins and losses
        switch (sEvt) {
            case ScoreEvent.gameWin:
                // If it's a win, add the score to the next round
                // static fields are NOT reset by
SceneManager.LoadScene()
                Prospector.SCORE_FROM_PREV_ROUND = score;
                print ("You won this round! Round score: "+score);
                gameOverText.text = "Round Over";
                roundResultText.text ="You won this round!\nRound
Score: Ê"+score;
                ShowResultsUI( true );
                break;
            case ScoreEvent.gameLoss:
                gameOverText.text = "Game Over";
                // If it's a loss, check against the high score
                if (Prospector.HIGH_SCORE <= score) {
                    print("You got the high score! High score: "+score);
                    string str = "You got the high score!\nHigh score:
Ê"+score;
                    roundResultText.text = str;
                    Prospector.HIGH_SCORE = score;
                    PlayerPrefs.SetInt("ProspectorHighScore", score);
                } else {
                    print ("Your final score for the game was: "+score);
```

```
                roundResultText.text = "Your final score was:
"+score;
            }
            ShowResultsUI( true );
            break;
        default:
            print ("score: "+score+"  scoreRun:"+scoreRun+"  chain:"
Ê+chain);
            break;
    }
  }

  ...

}
```

Now, when you finish a round or game, you should see messages like those in Figure 32.17.

Figure 32.17. Example game over messages



Summary

In this chapter, you created a complete card game that constructs itself from XML files and that contains scoring, background images, and theming. One of the purposes of the tutorials in this book is to give you a framework on which to build your own games. In the next chapter, we do just that. I'll guide you through building the Bartok game from the first chapter of the book based on this project.

Next Steps

The following are some possible directions that you can take this game yourself.

Gold Cards

We mentioned this as letter d in the list of ways to add scoring to the game, but gold cards were not implemented in the chapter. There are graphics in the package you imported for gold cards (both Card_Back_Gold and Card_Front_Gold). The purpose of the gold cards is to double the value of any run that they are part of. Gold cards can only start in the mine, and any card in the mine has a 10% chance of being a gold card. Try implementing the gold cards on your own.

Compile This Game on a Mobile Device

Though the build settings in this game were designed for an iPad, it's not really within the scope of this book to instruct you on actual compilation for a mobile device. Unity has several pages that document this however, and I recommend that you look at the proper one for the device that you own. In order to keep the information here as current as possible, my best recommendation for you is to do a web search for *Unity getting started* and the name of the mobile platform on which you want to develop (e.g., *Unity getting started iOS*). Right now, that could be *iOS* or *Android* for mobile platforms or *WebGL* for embedding in a website. The Unity documentation includes "getting started" pages for all of these platforms.

In my personal experience, I have found compilation on Android devices to be the easiest. Including the time to install and configure the additional software to do so, compiling this game for iOS took about two hours (most of which was spent setting up my Apple iOS developer account and provisioning profile), and compiling this game for Android took about 20 minutes.

I also highly recommend looking into some of the tools out there that can help you with mobile development. The Test Flight service that Apple acquired a few years ago helps you to distribute test builds of your game to iOS devices easily over the Internet (https://developer.apple.com/testflight/), and nearly everyone doing

iOS development uses it. If you want a cross-platform approach that can distribute to Android as well (but is less convenient for iOS), check out TestFairy (http://testfairy.com).

I also highly recommend looking into Unity Cloud Build, which used to be the independent company, Tsugi (that was mentioned in the first edition of the book). Unity Cloud Build watches your Git repository for changes in your code and automatically compiles new versions if it senses that anything has changed. If you're doing cross-platform mobile or WebGL development, Unity Cloud Build can save you a ton of time by offloading the heavy compiling tasks to a server instead of your personal machine.

Chapter 33. Prototype 5: Bartok

This chapter differs somewhat from the other tutorials because instead of creating an entirely new project, this one shows you how you can build a different game on top of the kinds of tutorials that you've developed while reading this book..

Before starting this project, you should have first completed Prototype 4: Prospector Solitaire so that you understand the inner workings of the card game framework developed in that chapter.

Bartok is the game you first encountered in Chapter 1, "Thinking Like a Designer." Now you'll build it yourself.

Getting Started: Prototype 5

This time, instead of downloading a unitypackage as you did before, just make a duplicate of your entire project folder for the Prospector game from the previous chapter (or you can download it from http://book.prototools.net under Chapter 33). Again, the art assets we'll be using are constructed from parts of the Vectorized Playing Cards 1.3 by Chris Aguilar.[1]

[1] Vectorized Playing Cards 1.3 (http://code.google.com/p/vectorized-playing-cards/) ©2011 - Chris Aguilar

Note that *this project will only work with Unity version 5 and later.*

Understanding Bartok

For a description of Bartok and how to play, see Chapter 1, where it is used extensively as an example. In short, Bartok is very similar to

the commercial game *Uno*, except that it is played with a standard deck of cards, and in the traditional Bartok card game, the winner of each round is able to add a rule to the game. In the Chapter 1 example, we also included three variations of the rules, but those will not be created in this chapter; I'll leave that to you to accomplish later.

To play an online version of the Bartok game, you can visit http://book.prototools.net and look under Chapter 1.

Making a New Scene

As with much of this project, the scene we will use will be based off of the scene from Prospector.

1. Select *__Prospector_Scene_0* in the Project pane and then choose *Edit* > *Duplicate* from the menu bar. This will make a new Scene named *__Prospector_Scene_1*.

2. Rename this to *__Bartok_Scene_0* and double-click it to open it. You can tell that it has opened because the title bar of the Unity window will change to reflect the new scene name and __Bartok_Scene_0 will appear at the top of the Hierarchy pane.

Cleaning the Scene

Let's get rid of some of the things we don't need.

1. Select _Scoreboard and HighScore under the Canvas in the Hierarchy pane and delete them (*Edit* > *Delete* from the menu bar). This game won't be scored, so we don't need either of those.

2. Similarly, you can delete both the GameOver and RoundResult children of Canvas from this scene. We'll be making use of them later but can always grab copies from __Prospector_Scene_0 when we need them.

3. Select _MainCamera and remove the *Prospector (Script)* and *Layout (Script)* components (right-click the name of each [or click the gear to the right of the name of each] and choose Remove Component). You should be left with a _MainCamera that has all the proper settings for Transform and Camera and also still has a *Deck (Script)* component.

4. Lastly, let's change the background. Start by selecting the ProspectorBackground GameObject in the Hierarchy pane (not the Texture2D ProjectPane) and renaming it *BartokBackground*. Then create a new Material in the _Materials folder (*Assets* > *Create* > *Material* from the menu bar) and name it *BartokBackground Mat*. Drag this new material on to BartokBackground. You'll notice in the Game pane that this made things very dark. (This is because the new material has a Diffuse shader while the previous material used the Unlit shader.) To remedy this, add a directional light to the scene (*GameObject* > *Light* > *Directional Light*). The transform for the BartokBackground and directional light should be as follows:



This should set the scene properly. Note that the position of the Directional Light doesn't matter at all to the scene, but it does get the light out of the way of what we need to do in the Scene pane.

The Importance of Adding Card Animation

This will be a game for a single human player, but the game of Bartok works best with four players, so three of the players will be AIs (artificial intelligences). Because Bartok is such a simple game, they won't have to be good AIs; they just need to act. The other thing that will need to happen is that we will have to let the player know whose turn it is and what the other players are doing. For this to work, we're going to make the cards animate from place to place in this game. This wasn't necessary in Prospector because the player was taking all of the actions herself, and it was obvious to her what the result should be. Because the player of Bartok is presented with

three other hands that will be face-down to her, the animation can be used as an important way to message what actions are being taken by the AI players.

Much of the challenge in designing this tutorial was in creating good animations and making sure that the game waited properly for each animation to end before moving on to the next thing. Because of that, you will see use of `SendMessage()` and `Invoke()` in this project as well as the use of more specific callback objects than `SendMessage()` allows. Instead, we will be passing a C# class instance to an object to and then call a *callback function* on the instance when the object is done moving, which is less flexible than `SendMessage()` but faster and more specific and can also be used for C# classes that don't extend `MonoBehaviour`.

Build Settings

Whereas the last project was designed as a mobile app, this will be a standalone application for Mac or PC, so the build settings will need to change.

1. From the menu bar, choose *File* > *Build Settings*, which will bring up the window shown in Figure 33.1.

Figure 33.1. The Build Settings window



You'll see that __Prospector_Scene_0 is currently in the list of *Scenes In Build*, but __Bartok_Scene_0 is not.

2. Click the *Add Ope Scenes* button to add __Bartok_Scene_0 to the list of scenes for this build.

3. Uncheck the box next to __Prospector_Scene_0 to remove it from the list of scenes.

4. Select *WebGL* from the list of platforms and click *Switch Platform*. The *Switch Platform* button will turn gray once the switch is complete. This may take a second, but it should be pretty fast. All the other settings should be fine as they are.

Once your build settings look like the image in Figure 33.1, you can close this window. (Don't click Build yet; that will happen after actually making the game.)

5. Look at the pop-up menu under the title of the Game pane. From that list of aspect ratios, change it to *Standalone (1024x768)*. This will ensure that your game aspect ratio looks the same as the examples that you'll see throughout this tutorial.

Coding Bartok

Just as we had a `Prospector` class to manage the game and a `CardProspector:Card` class to extend Card and add game-specific capabilities, we will need a `Bartok` and `CardBartok:Card` class in this game.

1. Create both a Bartok and a CardBartok C# script in the __Scripts folder of the Project pane (*Assets* > *Create* > *C# Script*).

2. Double-click CardBartok to open it in MonoDevelop and enter the following code. (If you want, you can copy some of this from CardProspector.)

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

// CBState includes both states for the game and to... states for movement
                                                    // a
public enum CBState {
    toDrawpile,
```

```
    drawpile,
    toHand,
    hand,
    toTarget,
    target,
    discard,
    to,
    idle
}

public class CardBartok : Card {                                    // b
    // Static variables are shared by all instances of CardBartok
    static public float      MOVE_DURATION = 0.5f;
    static public string     MOVE_EASING = Easing.InOut;
    static public float      CARD_HEIGHT = 3.5f;
    static public float      CARD_WIDTH = 2f;

    [Header("Set Dynamically")]
    public CBState           state = CBState.drawpile;

    // Fields to store info the card will use to move and rotate
    public List<Vector3>     bezierPts;
    public List<Quaternion>  bezierRots;
    public float             timeStart, timeDuration;

    // When the card is done moving, it will call
    // reportFinishTo.SendMessage()
    public GameObject        reportFinishTo = null;

    // MoveTo tells the card to interpolate to a new position and
rotation
    public void MoveTo(Vector3 ePos, Quaternion eRot) {
        // Make new interpolation lists for the card.
        // Position and Rotation will each have only two points.
        bezierPts = new List<Vector3>();
        bezierPts.Add ( transform.localPosition );  // Current
position
```

```
        bezierPts.Add ( ePos );                    // Current rotation
        bezierRots = new List<Quaternion>();
        bezierRots.Add ( transform.rotation );     // New position
        bezierRots.Add ( eRot );                   // New rotation

        if (timeStart == 0) {                                  // c
            timeStart = Time.time;
        }
        // timeDuration always starts the same but can be
overwritten
        // later
        timeDuration = MOVE_DURATION;

        state = CBState.to;                                    // d
    }

    public void MoveTo(Vector3 ePos) {                              // e
        MoveTo(ePos, Quaternion.identity);
    }

    void Update() {
        switch (state) {
            case CBState.toHand:                               // f
            case CBState.toTarget:

            case CBState.toDiscard:
            case CBState.to:
                float u = (Time.time - timeStart)/timeDuration;      // g
                float uC = Easing.Ease (u, MOVE_EASING);

                if (u<0) {                                     // h
                    transform.localPosition = bezierPts[0];
                    transform.rotation = bezierRots[0];
                    return;
                } else if (u>=1) {                             // i
                    uC = 1;
                    // Move from the to... state to the proper next state
```

```
            if (state == CBState.toHand)    state = CBState.hand;
            if (state == CBState.toTarget)  state = CBState.targe
```

```
t;
            if (state == CBState.toDrawpile) 
```

```
state =
CBState.drawpile;
            if (state == CBState.to)       state = CBState.idle;
            // Move to the final position
            transform.localPosition =
bezierPts[bezierPts.Count-
```

```
1];
            transform.rotation = bezierRots[bezierPts.Count-1];
            // Reset timeStart to 0 so it gets overwritten next
            // time
            timeStart = 0;

            if (reportFinishTo != null) {                  // j
               reportFinishTo.SendMessage("CBCallback", this);
               reportFinishTo = null;
            } else { // If there is nothing to callback
               // Just let it stay still.
            }
         } else {                                          // k
            Vector3 pos = Utils.Bezier(uC, bezierPts);
            transform.localPosition = pos;
            Quaternion rotQ = Utils.Bezier(uC, bezierRots);
            transform.rotation = rotQ;
         }
         break;
      }

   }

}
```

a. The enum CBState includes both states for the game purpose of the CardBartok and various to... states that describe where a moving CardBartok is headed.

b. CardBartok extends Card, just as CardProspector did.

c. If timeStart is 0, then it's set to start immediately, otherwise, it starts at timeStart. This way, if timeStart is already set, it won't be overwritten.

d. Initially, state is set to just CBState.to. The calling method will later specify whether state should be CBState.toHand or CBState.toTarget.

e. This is an overload of MoveTo() that doesn't require a rotation to be passed in.

f. Because switch statements allow cases to "fall through" as long as there isn't any code between them, all of the the to... CBStates (i.e., toHand, toTarget, and to)—where the card is interpolating from one place to another—can be handled together.

g. If in one of the to... states, we get u from the current time and duration and then use the Easing class from Utils.cs to curve that u value and return uC.

h. u usually ranges from 0 to 1. This handles it when u<0, in which case we shouldn't move yet and stay at the initial position. Thus u<0 case can happen when we set the timeStart some time in the future to delay the beginning of movement.

i. In the case where u>=1, we want to clamp it to 1 so that the card doesn't overshoot its movement target. Then we stop movement by switching to another CBState.

j. If there's a callback GameObject, then use SendMessage to call the CBCallback method with this as the parameter. After calling SendMessage(), reportFinishTo must be set to null so that it the card doesn't continue to report to the same GameObject every subsequent time it moves.

k. When 0<=u<1, we should just interpolate from the previous location to the next one. Use a Bézier curve function to move this to the right point. Position and rotation are handled separately by different overloads of the Utils.Bezier() method.

A lot of this is an adaptation and expansion on the code that you saw in the preceding chapter for the `FloatingScore` class. The CardBartok version of interpolation also interpolates Quaternions (a class that handles rotations), which will be important because we want the cards in Bartok to fan as if they were being held by a player.

3. Open the `Bartok` class and enter this code. The first thing we want to do in the Bartok class is to make sure that the `Deck` class is working properly to create all 52 cards:

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

public class Bartok : MonoBehaviour {
    static public Bartok S;

    [Header("Set in Inspector")]
    public TextAsset          deckXML;
    public TextAsset          layoutXML;
    public Vector3            layoutCenter = Vector3.zero;

    [Header("Set Dynamically")]
    public Deck               deck;
    public List<CardBartok>   drawPile;
    public List<CardBartok>   discardPile;


    void Awake() {
        S = this;
```

```
    }

    void Start () {
        deck = GetComponent<Deck>();     // Get the Deck
        deck.InitDeck(deckXML.text);     // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards);    // This shuffles the deck    //
a

    }

}
```

a. The ref keyword passes a reference to deck.cards, which allows deck.cards to be modified directly by Deck.Shuffle().

As you can see, most of this is the same as what you saw in Prospector, except that you're now dealing with the `CardBartok` class for cards rather than the `CardProspector` class.

Setting Up PrefabCard in the Inspector

At this time, you should also adjust other aspects of PrefabCard in the Inspector.

1. Select PrefabCard in _Prefabs folder of the Project pane.

2. Set the Box Collider component's *Is Trigger* field to true.

3. Set the Size.z of the Box Collider component to 0.1.

4. Add a Rigidbody component to PrefabCard (*Component* > *Physics* > *Rigidbody*).

5. Set the Rigidbody's *Use Gravity* field to false.

6. Set the Rigidbody's *Is Kinematic* field to true.

When finished, the Box Collider and Rigidbody components on PrefabCard should look like Figure 33.2.

Figure 33.2. Box Collider and Rigidbody settings for PrefabCard



7. You're going to need to swap a new *CardBartok (Script)* component for the existing *CardProspector (Script)* component.

a. Click the gear icon to the right of the name of the *CardProspector (Script)* component and choose *Remove Component*.

b. Attach a CardBartok script to PrefabCard.

Setting Up _MainCamera in the Inspector

Follow these steps to set up _MainCamera in the inspector:

1. Attach the Bartok script to _MainCamera in the Hierarchy (assign it however you like; you should know what you're doing by now).

2. In the Hierarchy pane, select _MainCamera. The attached Bartok (Script) component is at the bottom of the inspector. (If you want to move it up, you can click the gear next to its name and choose Move Up.)

3. Set the `DeckXML` field of Bartok (Script) to the DeckXML file that is in the Resources folder of the Project pane. (Because the deck remains unchanged [still 13 cards of 4 suits], this is the same file that was used by Prospector.)

4. Set the startFaceUp field of the Deck (Script) component to true (checked). This will show all the cards face up when you press play.

Now when you press Play, you should see a grid of cards just as you saw in the early stages of Prospector. In only a few pages, we're pretty far along.

The Game Layout

The layout for Bartok differs significantly from Prospector. In Bartok, there will be a draw pile and discard pile in the middle of the screen as well as four hands of cards distributed to the top, left, bottom, and right sides of the screen. The hands should be fanned as if they were being held by a player (see Figure 33.3).

Figure 33.3. The eventual layout of Bartok



This will require a somewhat different layout XML document than was used for Prospector.

1. Select LayoutXML in the Resources folder of the Project pane and duplicate it (*Edit* > *Duplicate*).

2. Name the duplicate BartokLayoutXML and enter the following text. (Bold text differs from the original LayoutXML text.)

```
<xml>
    <!-- This file includes info for laying out the Bartok card game. -->

    <!-- The multiplier is multiplied by the x and y attributes below. -->
    <!-- This determines how loose or tight the layout is. -->
    <multiplier x="1" y="1" />

    <!-- This positions the draw pile and staggers it -->
    <slot type="drawpile" x="1.5" y="0" xstagger="0.05"
layer="1"/>

    <!-- This positions the discard pile -->
    <slot type="discardpile" x="-1.5" y="0" layer="2"/>

    <!-- This positions the target card -->
    <slot type="target" x="-1.5" y="0" layer="4"/>
```

```
   <!-- These slots are for the four hands held by the four
players -->
   <slot type="hand" x="0"   y="-8" rot="0"   player="1"
layer="3"/>
   <slot type="hand" x="-10" y="0"  rot="270" player="2"
layer="3"/>
   <slot type="hand" x="0"   y="8"  rot="180" player="3"
layer="3"/>
   <slot type="hand" x="10"  y="0"  rot="90"  player="4"
layer="3"/>

</xml>
```

The BartokLayout C# Script

Now, the class that does the layout must also be rewritten to both fan the cards properly and to take advantage of the new ability to interpolate cards.

1. Create a new C# script named *BartokLayout* in the Scripts folder and enter this code:

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

[System.Serializable]                                   // a
public class SlotDef {                                  // b
   public float       x;
   public float       y;
   public bool        faceUp=false;
   public string      layerName="Default";
   public int         layerID = 0;
   public int         id;
   public List<int>    hiddenBy = new List<int>(); // Unused in
Bartok
   public float       rot;       // rotation of hands
```

```
    public string      type="slot";
    public Vector2     stagger;
    public int         player;    // player number of a hand
    public Vector3     pos;       // pos derived from x, y, &
multiplier
}

public class BartokLayout : MonoBehaviour {

}
```

a. [System.Serializable] makes SlotDef able to be seen in the Unity Inspector.

b. The SlotDef class is not based on MonoBehaviour, so it doesn't need its own file.

2. Save this code and return to Unity. You'll notice that this causes an error in the console:

"error CS0101: The namespace 'global::' already contains a definition for 'SlotDef'.

This is because the public class `SlotDef` in the Layout script (from Prospector) is conflicting with the public class `SlotDef` in the new BartokLayout script.

3. Either delete the Layout script entirely or open the Layout script in MonoDevelop and comment out the section defining `SlotDef`. To comment out a large chunk of code, just place a `/*` before the code and a `*/` after the code you wish to comment. You can also comment out a large section by selecting the lines of code in MonoDevelop and choosing *Edit > Format > Toggle Line Comment(s)* from the menu bar, which will place a single line comment (`//`) before each line you have selected.

4. Regardless of which method you use to comment out SlotDef from the Layout script, make sure that you also comment out the

[System.Serializable] line preceding the SlotDef definition there.

5. After you have eliminated the `SlotDef` class from the Layout script, save the Layout script.

6. Return to the BartokLayout script and continue editing it by adding the bolded lines in the following code listing:

```
public class BartokLayout : MonoBehaviour {
    [Header("Set Dynamically")]
    public PT_XMLReader     xmlr; // Just like Deck, this has an
Image PT_XMLReader
    public PT_XMLHashtable  xml; // This variable is for faster
xml access
    public Vector2          multiplier; // Sets the spacing of the
tableau
    // SlotDef references
    public List<SlotDef>    slotDefs; // The SlotDefs hands
    public SlotDef          drawPile;
    public SlotDef          discardPile;
    public SlotDef          target;

    // Bartok calls this method to read in the
BartokLayoutXML.xml file
    public void ReadLayout(string xmlText) {
        xmlr = new PT_XMLReader();
        xmlr.Parse(xmlText);      // The XML is parsed
        xml = xmlr.xml["xml"][0]; // And xml is set as a shortcut to
the
Image XML

        // Read in the multiplier, which sets card spacing
        multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
        multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

        // Read in the slots
        SlotDef tSD;
```

```
        // slotsX is used as a shortcut to all the <slot>s
        PT_XMLHashList slotsX = xml["slot"];

        for (int i=0; i<slotsX.Count; i++) {
            tSD = new SlotDef();  // Create a new SlotDef instance
            if (slotsX[i].HasAtt("type")) {
                // If this <slot> has a type attribute parse it
                tSD.type = slotsX[i].att("type");
            } else {
                // If not, set its type to "slot"; it's a card in the rows
                tSD.type = "slot";
            }

            // Various attributes are parsed into numerical values
            tSD.x = float.Parse( slotsX[i].att("x") );
            tSD.y = float.Parse( slotsX[i].att("y") );
            tSD.pos = new Vector3( tSD.x*multiplier.x,
tSD.y*multiplier.y, Image0 );

            // Sorting Layers
            tSD.layerID = int.Parse( slotsX[i].att("layer") );       // a
            tSD.layerName = tSD.layerID.ToString();                  // b

            // pull additional attributes based on the type of each
<slot>
            switch (tSD.type) {
                case "slot":
                    // ignore slots that are just of the "slot" type
                    break;

                case "drawpile":                                     // c
                    tSD.stagger.x = float.Parse( slotsX[i].att("xstagger")
Image);
                    drawPile = tSD;
                    break;

                case "discardpile":
```

```
                discardPile = tSD;
                break;

            case "target":
                // The target card has a different layer from
                // discardPile
                target = tSD;
                break;

            case "hand":                                    // d
                tSD.player = int.Parse( slotsX[i].att("player") );
                tSD.rot = float.Parse( slotsX[i].att("rot") );
                slotDefs.Add (tSD);
                break;

        }
    }
  }
}
```

a. In this game, the Sorting Layers are named 1, 2, 3, ...through 10. The layers are used to make sure that the correct cards are on top of the others. In Unity 2D, all of our assets are effectively at the same Z depth, so the sorting layers are used to differentiate between them.

b. This converts the number of the layerID to a text layerName.

c. The drawpile xstagger value is read in, but this is not used in Bartok since the players don't actually need to know how many cards are in the draw pile.

d. This section reads in data particular to each player's hand, including the rotation of the hand and the number of the player who will have access to that hand.

7. Now attach the BartokLayout script to _MainCamera. (Drag the BartokLayout script from the Project pane onto _MainCamera in the Hierarchy pane.)

8. In the *Bartok (Script)* component on _MainCamera, assign the BartokLayoutXML file in the Resources folder of the Project pane to the `layoutXML` field.

9. Open the Bartok script and add the following bolded code to have it make use of BartokLayout:

```
public class Bartok : MonoBehaviour {
   static public Bartok S;


   ...
   public List<CardBartok>     discardPile;

   private BartokLayout       layout;
   private Transform          layoutAnchor;

   void Awake() { ... }

   void Start () {
      deck = GetComponent<Deck>();     // Get the Deck
      deck.InitDeck(deckXML.text);     // Pass DeckXML to it
      Deck.Shuffle(ref deck.cards);    // This shuffles the deck

      layout = GetComponent<BartokLayout>();   // Get the
Layout
      layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it

      drawPile = UpgradeCardsList( deck.cards );
   }

   List<CardBartok> UpgradeCardsList(List<Card> lCD)
{            // a
      List<CardBartok> lCB = new List<CardBartok>();
      foreach( Card tCD in lCD ) {
         lCB.Add ( tCD as CardBartok );
      }
      return( lCB );
```

```
    }

}
```

a. This method upgrades all of the Cards in the `List<Card> lCD` to be CardBartoks and creates a new List<CardBartok> to hold them. Of course, they were always CardBartoks, but this lets Unity know that.

10. Return to Unity and run the project. When you run the project now, you should be able to select _MainCamera from the Hierarchy pane and expand the variables in the *BartokLayout (Script)* component to see that they're being populated with the correct values from BartokLayoutXML. You should also look at the `drawPile` field of *Bartok (Script)* to see that it is properly filled with 52 shuffled CardBartok instances.

The Player Class

Because this game has four players, I've chosen to create a class to represent players that can do things like gather cards into a hand and eventually choose what to play using simple artificial intelligence. One thing that is unique about the `Player` class relative to others that you've written is that the `Player` class does *not* extend `MonoBehaviour` (or any other class). This means that it doesn't receive calls from `Awake()`, `Start()`, or `Update()` and that you can't call some functions like `print()` from within it or attach it to a GameObject as a component. However, none of that is necessary for the `Player` class, so it is actually easier in this case to work without it.

1. Create a new C# script in the __Scripts folder named *Player* and enter this code:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Linq; // Enables LINQ queries, which will be explained soon
```

```
// The player can either be human or an ai
public enum PlayerType {
   human,
   ai
}

[System.Serializable]                                    // a
public class Player {                                    // b
   public PlayerType       type = PlayerType.ai;
   public int              playerNum;
   public SlotDef          handSlotDef;
   public List<CardBartok>   hand; // The cards in this player's
hand

   // Add a card to the hand
   public CardBartok AddCard(CardBartok eCB) {
      if (hand == null) hand = new List<CardBartok>();

      // Add the card to the hand
      hand.Add (eCB);

      return( eCB );
   }

   // Remove a card from the hand
   public CardBartok RemoveCard(CardBartok cb) {
      hand.Remove(cb);
      return(cb);
   }

}
```

a. `[System.Serializable]` instructs Unity to serialize the Player class, enabling it to be viewed and edited within the Unity Inspector.

b. The `Player` class stores information that is important to each player. As mentioned before, it does not extend `MonoBehaviour` or any other class, so you must delete the "`: MonoBehaviour`" from this line.

2. Now, add the following code to Bartok to make use of the Player class:

```
public class Bartok : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public Vector3            layoutCenter = Vector3.zero;
    public float              handFanDegrees = 10f;              // a

    [Header("Set Dynamically")]
    ...
    public List<CardBartok>    discardPile;
    public List<Player>        players;                          // b
    public CardBartok          targetCard;

    private BartokLayout       layout;
    private Transform          layoutAnchor;

    void Awake() { ... }

    void Start () {
        ...
        drawPile = UpgradeCardsList( deck.cards );
        LayoutGame();
    }

    List<CardBartok> UpgradeCardsList(List<Card> lCD) { ... }

    // Position all the cards in the drawPile properly
    public void ArrangeDrawPile() {
        CardBartok tCB;
```

```
        for (int i=0; i<drawPile.Count; i++) {
            tCB = drawPile[i];
            tCB.transform.parent = layoutAnchor;
            tCB.transform.localPosition = layout.drawPile.pos;
            // Rotation should start at 0
            tCB.faceUp = false;
            tCB.SetSortingLayerName(layout.drawPile.layerName);
            tCB.SetSortOrder(-i*4); // Order them front-to-back
            tCB.state = CBState.drawpile;
        }
    }

    // Perform the initial game layout
    void LayoutGame() {
        // Create empty GameObject to serve as the tableau's
anchor   // c
        if (layoutAnchor == null) {
            GameObject tGO = new GameObject("_LayoutAnchor");
            layoutAnchor = tGO.transform;
            layoutAnchor.transform.position = layoutCenter;
        }

        // Position the drawPile cards
        ArrangeDrawPile();

        // Set up the players
        Player pl;
        players = new List<Player>();
        foreach (SlotDef tSD in layout.slotDefs) {
            pl = new Player();
            pl.handSlotDef = tSD;
            players.Add(pl);
            pl.playerNum = tSD.player;
        }
        players[0].type = PlayerType.human;  // Make only the 0th
player
                                    // human
```

```
    }

    // The Draw function will pull a single card from the drawPile
and
    // return it
    public CardBartok Draw() {
        CardBartok cd = drawPile[0];     // Pull the 0th
CardProspector

        if (drawPile.Count == 0) {      // If the drawPile is now empty
            // We need to shuffle the discards into the drawPile
            int ndx;
            while (discardPile.Count > 0) {
                // Pull a random card from the discard pile
                ndx = Random.Range(0, discardPile.Count);           // d
                drawPile.Add( discardPile[ndx] );
                discardPile.RemoveAt( ndx );
            }
            ArrangeDrawPile();
            // Show the cards moving to the drawPile
            float t = Time.time;
            foreach (CardBartok tCB in drawPile) {
                tCB.transform.localPosition = layout.discardPile.pos;
                tCB.callbackPlayer = null;
                tCB.MoveTo(layout.drawPile.pos);
                tCB.timeStart = t;
                t += 0.02f;
                tCB.state = CBState.toDrawpile;
                tCB.eventualSortLayer = "0";
            }
        }

        drawPile.RemoveAt(0);       // Then remove it from List<>
drawPile
        return(cd);                 // And return it
    }
```

```
    // This Update() is temporarily used to test adding cards to
    // players' hands
    void Update() {                                    // e
        if (Input.GetKeyDown(KeyCode.Alpha1)) {
            players[0].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha2)) {
            players[1].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha3)) {
            players[2].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha4)) {
            players[3].AddCard(Draw ());
        }
    }

}
```

a. `handFanDegrees` determines how many degrees rotation there should be between each card in a fanned hand.

b. This List of Players holds a reference to the data for each player. Because the Player class is [System.Serializable], it's possible to explore the depths of the `players` List within the Unity Inspector.

c. The `layoutAnchor` is a Transform created to be the parent in the Hierarchy over all of the cards in the tableau. First, an empty GameObject is created named *_LayoutAnchor*. Then the Transform is pulled from that GameObject and assigned to the field `layoutAnchor`. Finally, the position of `layoutAnchor` is set to the location specified by `layoutCenter`.

d. It's easier to use this while loop to pull random cards from the `discardPile` than to convert the `discardPile` from a `List<CardBartok>` to a `List<Card>` so we can call `Deck.Shuffle()` on it.

e. This temporary Update() function will be used to test the code we've written to add cards to each player's hand. The Keycode.Alpha1 through Keycode.Alpha4 refer to the number keys 1-4 on the main keyboard above the letters. When one of those keys is hit, a Card is added to that player's hand.

3. Save the scripts, return to Unity, and run the game again.

4. Select _MainCamera in the Hierarchy and find the `players` field on the *Bartok (Script)* component. Open the disclosure triangle for `players`, and you'll see four elements, one for each player. Open those disclosure triangles, as well, and then open up the disclosure triangles for `hand` under each. Because of the new `Update()` method, if you click in the Game pane (which gives the game focus and allows it to react to keyboard input), you can press the number keys 1 to 4 on your keyboard (across the top of the keyboard, not the keypad) and watch cards be added to the players' hands. The Inspector for the *Bartok (Script)* component should show cards being added to hands as shown in Figure 33.4.

Figure 33.4. Bartok (Script) component showing players and their hands



This `Update()` method, of course, won't be used in the final version of the game, but it is often useful to build little functions like this that allow you to test features before other aspects of the game are ready. In this case, we needed a way to test whether the `Player.AddCard()` method worked properly, and this was a quick way to do so.

Fanning the Hands

Now that cards are being moved from the drawPile into players' hands, it's time to graphically move them there.

1. Add the following code to the Player class to make this happen:

```
public class Player {
    ...

    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Add the card to the hand
        hand.Add (eCB);
        FanHand();
        return( eCB );
    }

    // Remove a card from the hand
    public CardBartok RemoveCard(CardBartok cb) {
        hand.Remove(cb);
        FanHand();
        return(cb);
    }

    public void FanHand() {                                    // a
        // startRot is the rotation about Z of the first card        // b
        float startRot = 0;
        startRot = handSlotDef.rot;
        if (hand.Count > 1) {
            startRot += Bartok.S.handFanDegrees * (hand.Count-1) /
2;
        }

        // Move all the cards to their new positions
        Vector3 pos;
        float rot;
        Quaternion rotQ;
        for (int i=0; i<hand.Count; i++) {
            rot = startRot - Bartok.S.handFanDegrees*i;
            rotQ = Quaternion.Euler( 0, 0, rot );                  // c

            pos = Vector3.up * CardBartok.CARD_HEIGHT /
```

```
2f;          // d

            pos = rotQ * pos;                                   // e

            // Add the base position of the player's hand (which will
be
            // at the bottom-center of the fan of the cards)
            pos += handSlotDef.pos;                             // f
            pos.z = -0.5f*i;                                    // g

            // Set the localPosition and rotation of the ith card in the
            // hand
            hand[i].transform.localPosition = pos;              // h
            hand[i].transform.rotation = rotQ;
            hand[i].state = CBState.hand;

            hand[i].faceUp = (type == PlayerType.human);        // i

            // Set the SortOrder of the cards so that they overlap
            // properly
            hand[i].SetSortOrder(i*4);                          // j
        }

    }

}
```

a. `FanHand()` will rotate the Cards to appear fanned in an arc as shown in Figure 33.1.

b. `startRot` is the rotation about Z of the first card (the one rotated the most counter-clockwise). It starts with the rotation of the entire hand as specified in `BartokLayoutXML` and then rotates counterclockwise so that the fanned cards will appear to be centered when rotated. After choosing `startRot`, each subsequent card is rotated `Bartok.S.handFanDegrees` clockwise from the previous card.

c. `rotQ` is the expression of the `rot` about the Z axis as a Quaternion.

d. Then pos is chosen, which is a Vector3 location half a Card height above the center of this hand (i.e., localPosition = [0,0,0]), so pos is initially [0,1.75,0].

e. The Quaternion rotQ is then multiplied by the Vector3 pos. When a Quaternion is multiplied by a Vector3, it rotates the Vector3, so now pos is rotated rot degrees around the local origin.

f. The base position of the hand is added to pos.

g. The pos of the various cards in the hand is staggered in the Z direction. While this isn't actually visible (because we're dealing with 2D sprites), it does keep the 3D Box Colliders we're using from overlapping.

h. Apply the pos and rotQ we calculated to the i^{th} Card in the hand.

i. Only the human player's Cards should be face up.

j. Setting the sort order of each card causes them to overlap properly within a single sorting layer.

2. Save the Player script, return to Unity, and press Play.

3. Try pressing the numbers 1, 2, 3, and 4 on the top row of your keyboard, you should see cards jumping into the players' hands and being fanned correctly. However, you probably noticed that the cards aren't sorted by rank in the human player's hand, which looks kind of sloppy. Luckily, we can do something about that.

A Tiny Introduction to LINQ

LINQ, which stands for *Language INtegrated Query*, is a fantastic extension to C# that has had many books written about it. Fully 24 pages of Josepha and Ben Albahari's fantastic *C# 5.0 Pocket Reference*[2] are devoted to LINQ (wherein they only devote 4 pages

to arrays). Most of LINQ is far beyond the scope of this book, but it's important that you know that it exists and what it can do.

[2] Joseph Albahari and Ben Albahari, *C# 5.0 Pocket Reference: Instant Help for C# 5.0 Programmers* (Beijing: O'Reilly Media, Inc., 2012).

LINQ has the capability to do database-like queries within a single line of C#, allowing you to select and order specific elements in an array. This is how we will sort the cards in the human player's hand.

1. Add the following bolded lines to `Player.AddCard()`:

```
public class Player {
    ...

    // Add a card to the hand
    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Add the card to the hand
        hand.Add (eCB);

        // Sort the cards by rank using LINQ if this is a human
        if (type == PlayerType.human) {
            CardBartok[] cards = hand.ToArray();                    // a

            // Below is the LINQ call
            cards = cards.OrderBy( cd => cd.rank ).ToArray();       //
b

            hand = new List<CardBartok>(cards);                     // c
            // Note: LINQ operations can be a bit slow (like it could
take
            // a couple milliseconds), but since we're only doing it
once
            // every turn, it isn't a problem.
```

```
        }

        FanHand();
        return( eCB );
    }

    ...
}
```

a. LINQ works on arrays of values, so we create a `CardBartok[]` array `cards` from the `List<CardBartok> hand`.

b. This line is a LINQ call that works on the `cards` array of CardBartoks. It is similar to doing a `foreach(CardBartok cd in cards)` and sorting them by `rank` (which is what is meant by `cd => cd.rank`). It then returns a sorted array, which is assigned to `cards`, replacing the old, unsorted array.

Note that LINQ operations can be a bit slow—a single call could take a couple of milliseconds—but since we're only making this LINQ call once every turn, it isn't a problem.

c. Once the `cards` array is sorted, we create a new `List<CardBartok>` from it and assign that to `hand`, replacing the old unsorted List.

As you can see, with a single line of LINQ code, we were able to sort the list. LINQ has tremendous capabilities that are beyond the scope of this book, but I highly recommend you look them up if you need to do sorting or other query-like operations on elements in an array (for example, if you had an array of people and needed to find all of them between the ages of 18 and 25).

2. Save the Player script, return to Unity, and play the scene. You'll see that the cards in the human player's hand are now always in order by rank.

The cards are going to need to animate into position for the game to be intelligible to the player, so it's time to make the cards move.

Making Cards Move!

Now comes the fun part where we make the cards actually interpolate from one position and rotation to the next. This will make the card game look much more like it's actually being played, and as you'll see, it makes it easier for the player to understand what is happening in the game.

A lot of the interpolation that we'll do here is based on that which was done for FloatingScore in Prospector. Just like FloatingScore, we'll start an interpolation that will be handled by the card itself, and when the card is done moving, it will send a callback message to notify the game that it's done.

Let's start by moving the cards smoothly into the players' hands. CardBartok already has a lot of the movement code written, so let's take advantage of it.

1. Modify the following bolded code of the `Player.FanHand()` method:

```
public class Player {
    ...

    public void FanHand() {
        ...
        for (int i=0; i<hand.Count; i++) {
            ...
            pos.z = -0.5f*i;

            // Set the localPosition and rotation of the ith card in the
            // hand
            hand[i].MoveTo(pos, rotQ); // Tell CardBartok to
interpolate
            hand[i].state = CBState.toHand;
            // ^ After the move, CardBartok will set the state to
            // CBState.hand
```

```
            /* <= This "/*" begins a multiline comment          // a
            hand[i].transform.localPosition = pos;
            hand[i].transform.rotation = rotQ;
            hand[i].state = CBState.hand;
            */                                              // b

            hand[i].faceUp = (type == PlayerType.human);

            ...
        }

    }
}
```

a. The `/*` begins a multiline comment, so all lines of code between it and the following `*/` are considered to be commented out (and are ignored by C#). This is the same way that you could have commented out the `SlotDef` class in the Layout script at the beginning of this chapter.

b. The `*/` ends the multiline comment.

2. Save the Player script, return to Unity, and play the scene.

Now, when you play the scene and press the number keys (1, 2, 3, 4), you will see the cards actually move into place! Because most of the heavy lifting is done by CardBartok, this took very little code to implement. This is one of the great advantages of object-oriented code. We trust that CardBartok knows how to move on it's own so we can just call `MoveTo()` with a position and rotation, and CardBartok will do the rest.

Managing the Initial Card Deal

In the beginning of a round of Bartok, seven cards are dealt to each player, and then a single card is turned up from the drawPile to

become the first target card. Add the following code to Bartok to make this happen:

```
public class Bartok : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public float              handFanDegrees = 10f;
    public int                numStartingCards = 7;
    public float              drawTimeStagger = 0.1f;


    ...
    void LayoutGame() {
        ...
        players[0].type = PlayerType.human; // Make the 0th player human

        CardBartok tCB;
        // Deal seven cards to each player
        for (int i=0; i<numStartingCards; i++) {
            for (int j=0; j<4; j++) {                          // a
                tCB = Draw (); // Draw a card
                // Stagger the draw time a bit.
                tCB.timeStart = Time.time + drawTimeStagger * ( i*4 + j ); Image// b

                players[ (j+1)%4 ].AddCard(tCB);                    // c
            }
        }

        Invoke("DrawFirstTarget", drawTimeStagger * (numStartingCards*4+4)
Image); // d

    }

    public void DrawFirstTarget() {
```

```
        // Flip up the first target card from the drawPile
        CardBartok tCB = MoveToTarget( Draw () );
    }

    // This makes a new card the target
    public CardBartok MoveToTarget(CardBartok tCB) {
        tCB.timeStart = 0;
        tCB.MoveTo(layout.discardPile.pos+Vector3.back);
        tCB.state = CBState.toTarget;
        tCB.faceUp = true;

        targetCard = tCB;

        return(tCB);
    }

    // The Draw function will pull a single card from the drawPile and
    // return it
    public CardBartok Draw() { ... }
    ...
}
```

a. The `j` variable ranges from 0 to 3 because there are four players. If the game accommodated different numbers of players, this would need to be dynamic rather than using the integer literal `j<4` as the terminal clause.

b. By staggering the `timeStart` for each Card, we cause them to be dealt out one after the other. Remember the order of operations for math here: `drawTimeStagger * ( i*4 + j )` happens before adding `Time.time`. The default `timeStart` for each Card is `Time.time`, so this adds a nice stagger to it.

c. Add the card to a Player's hand. The %4 will cause the players index to range from 0 to 3, starting with 1 (the Player clockwise after the human Player[0]).

d. Once all the initial cards have been drawn, DrawFirstTarget() will be called.

2. Save the Bartok script, return to Unity, and play the scene.

Upon playing the scene, you will see that the distribution of the seven cards and the draw of the first target happen properly on schedule, however, the human player's cards are overlapping each other in strange ways. Just as we did with Prospector, we need to very carefully manage both the `sortingLayerName` and the `sortingOrder` of each element of the cards.

Managing 2D Depth-Sorting Order

In addition to the standard issue of depth-sorting 2D objects, we now have to deal with the fact that the cards are moving, and there will be some times that we want them in one sort order at the beginning of the move and another when they arrive. To enable that, we will add fields for an `eventualSortLayer` and `eventualSortOrder` to CardBartok. This way, when a card is moving, it will switch to the `eventualSortLayer` and `eventualSortOrder` partway through the move.

1. The first thing you need to do is rename all of the sorting layers. Open the Tags & Layers settings by choosing *Edit* > *Project Settings* > *Tags & Layers* from the menu bar.

2. Set the names of Sorting Layers 1 through 10 to *1* through *10*, as shown in Figure 33.5.

Figure 33.5. Simply named sorting layers for Bartok



3. Once this is done, add the following bolded code to CardBartok:

```
public class CardBartok : Card {
    ...
```

```
    [Header("Set Dynamically")]
    ...
    public float            timeStart, timeDuration;
    public int              eventualSortOrder;
    public string           eventualSortLayer;

    ...

    void Update() {
        switch (state) {
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
                ...
                } else {
                    Vector3 pos = Utils.Bezier(uC, bezierPts);
                    transform.localPosition = pos;
                    Quaternion rotQ = Utils.Bezier(uC, bezierRots);
                    transform.rotation = rotQ;

                    if (u>0.5f) {                           // a
                        SpriteRenderer sRend = spriteRenderers[0];
                        if (sRend.sortingOrder != eventualSortOrder) {
                            // Jump to the proper sort order
                            SetSortOrder(eventualSortOrder);
                        }
                        if (sRend.sortingLayerName != eventualSortLayer)
{
                            // Jump to the proper sort layer
                            SetSortingLayerName(eventualSortLayer);
                        }
                    }

                }
                break;
        }
```

```
    }
}
```

a. When the move is halfway done (i.e., `u>0.5f`), the Card jumps to the `eventualSortOrder` and the `eventualSortLayer`.

Now that the `eventualSortOrder` and `eventualSortLayer` fields exist, we need to use them throughout the code that has already been written.

4. We'll incorporate this into the `MoveToTarget()` method of the Bartok script and also add a `MoveToDiscard()` function that moves the target card into the discardPile:

```
public class Bartok : MonoBehaviour {
    ...

    public CardBartok MoveToTarget(CardBartok tCB) {
        tCB.timeStart = 0;
        tCB.MoveTo(layout.discardPile.pos+Vector3.back);
        tCB.state = CBState.toTarget;
        tCB.faceUp = true;

        tCB.SetSortingLayerName("10");//layout.target.layerName);
        tCB.eventualSortLayer = layout.target.layerName;
        if (targetCard != null) {
            MoveToDiscard(targetCard);
        }

        targetCard = tCB;

        return(tCB);
    }

    public CardBartok MoveToDiscard(CardBartok tCB) {
        tCB.state = CBState.discard;
        discardPile.Add ( tCB );
        tCB.SetSortingLayerName(layout.discardPile.layerName);
```

```
        tCB.SetSortOrder( discardPile.Count*4 );
        tCB.transform.localPosition = layout.discardPile.pos +
Vector3.back/2;

        return(tCB);
    }

    // The Draw function will pull a single card from the drawPile and
    // return it
    public CardBartok Draw() { ... }
    ...
}
```

5. And there are a couple of changes to be made to the `AddCard()` and `FanHand()` methods of Player as well:

```
public class Player {
    ...
    public CardBartok AddCard(CardBartok eCB) {
        ...
        // Sort the cards by rank using LINQ if this is a human
        if (type == PlayerType.human) {
            ...
        }

        eCB.SetSortingLayerName("10"); // Sorts the moving card
to the top
// a
        eCB.eventualSortLayer = handSlotDef.layerName;

        FanHand();
        return( eCB );
    }

    // Remove a card from the hand
    public CardBartok RemoveCard(CardBartok cb) { ... }
```

```
    public void FanHand() {
        ...
            hand[i].faceUp = (type == PlayerType.human);

            // Set the SortOrder of the cards so that they overlap
            // properly
            hand[i].eventualSortOrder = i*4;                    // b
            //hand[i].SetSortOrder(i*4);
        }
    }
}
```

a. Setting the sorting layer of the moving card to `"10"` causes it to be above all other cards while it's moving. Based on the code we added to CardBartok in step 3 of this section, halfway through the move, the card will then jump to its `eventualSortLayer`.

b. Comment out the line that was here (shown on the subsequent line now) and replace it with this one.

6. Make sure you've saved the changes to all of these scripts, return to Unity, and press Play. You should now see the cards layering much better.

Handling Turns

In this game, players will need to take turns. We'll start by having the Bartok script track whose turn it is.

1. Open the Bartok script and add the bolded code shown here:

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

// This enum contains the different phases of a game turn
public enum TurnPhase {
```

```
    idle,
    pre,
    waiting,
    post,
    gameOver
}

public class Bartok : MonoBehaviour {
    static public Bartok S;
    static public Player CURRENT_PLAYER;                    // a

    ...

    [Header("Set Dynamically")]
    ...
    public CardBartok          targetCard;
    public TurnPhase           phase = TurnPhase.idle;

    private BartokLayout       layout;

    ...

    public void DrawFirstTarget() {
        // Flip up the first target card from the drawPile
        CardBartok tCB = MoveToTarget( Draw () );
        // Set the CardBartok to call CBCallback on this Bartok
when it is
        // done
        tCB.reportFinishTo = this.gameObject;               // b
    }

    // This callback is used by the last card to be dealt at the
beginning
    public void CBCallback(CardBartok cb) {                 //
c       // You sometimes want to have reporting of method calls
like this
        Utils.tr("Bartok.CBCallback()",cb.name);            // d
```

```
      StartGame(); // Start the Game
   }

   public void StartGame() {
      // Pick the player to the left of the human to go first.
      PassTurn(1);                                    // e
   }

   public void PassTurn(int num=-1) {                          // f
      // If no number was passed in, pick the next player
      if (num == -1) {
         int ndx = players.IndexOf(CURRENT_PLAYER);
         num = (ndx+1)%4;
      }
      int lastPlayerNum = -1;
      if (CURRENT_PLAYER != null) {
         lastPlayerNum = CURRENT_PLAYER.playerNum;
      }
      CURRENT_PLAYER = players[num];
      phase = TurnPhase.pre;

//       CURRENT_PLAYER.TakeTurn();                          // g

      // Report the turn passing
      Utils.tr("Bartok.PassTurn()", "Old:
"+lastPlayerNum,  Ê"New:
"+CURRENT_PLAYER.playerNum);                          // h
   }

   // ValidPlay verifies that the card chosen can be played on the
   // discard pile
   public bool ValidPlay(CardBartok cb) {
      // It's a valid play if the rank is the same
      if (cb.rank == targetCard.rank) return(true);

      // It's a valid play if the suit is the same
      if (cb.suit == targetCard.suit) {
```

```
            return(true);
        }

        // Otherwise, return false
        return(false);
    }

    ...


    /* Now is a good time to comment out this testing
code          // i
    // This Update method is used to test adding cards to players'
hands
    void Update() {
        if (Input.GetKeyDown(KeyCode.Alpha1)) {
            players[0].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha2)) {
            players[1].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha3)) {
            players[2].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha4)) {
            players[3].AddCard(Draw ());
        }
    }
    */

}
```

a. `CURRENT_PLAYER` is static and public here for two reasons: 1. There should only ever be one current player in the game, and 2. Making it static will allow the TurnLight that we'll set up in a subsequent section to access it easily.

b. `reportFinishTo` is a GameObject field that already exists on the CardBartok class. It gives the CardBartok a reference back to the `gameObject` of this Bartok instance (which is _MainCamera in this case). Our existing CardBartok code will already call `SendMessage("CBCallback",this)` on the `reportFinishTo` GameObject if it is not `null`.

c. The `CDCallback()` method is called by the first target card when it is done moving into place (as just described in b.).

d. The call to `Utils.tr()` on this line reports to the Console that `CBCallback()` was called. This is the first use you've seen of the `static public Utils.tr()` method (`tr` being short for "trace"). This method takes any number of arguments (via the params keyword), concatenates them with tabs in between, and outputs them to the Console pane. It is one of the elements that were added to the Utils class in the unitypackage that you imported into Prospector.

`tr()` is called here with a string literal of the method name (`"Bartok.CBCallback()"`) and the name of the GameObject that called `CBCallback()`.

e. The game will always start with the player clockwise from the human player. Since the human is `players[0]`, passing the turn to player 1 will make `players[1]` active.

f. The `PassTurn()` method has an optional parameter allowing us to specify which player to pass the turn to. Without any int passed in, `num` defaults to -1, and in the following 4 lines, that is converted into the number for the next clockwise player.

g. This line is currently commented out because the Player class does not yet have a `TakeTurn()` method. You will uncomment this line as part of the next section.

h. These two lines are actually a single line that was too long to fit in the book. You can type it as one line or as two. Because the first of these two lines does not have a semicolon (`;`) at the end, Unity

reads both lines as a single statement. Note that the `// h` at the end of the first line also does not break Unity's interpretation of it as a single line. In these cases, I will use the Ê code continuation symbol at the beginning of the second line. You **do not** need to type the Ê symbol.

i. This `Update()` method was initially used for testing, but we no longer need it. You can comment out the whole thing by adding `/*` before it and `*/` after it.

2. Save the Bartok script, return to Unity, and press play. You should see the initial hands deal out and then a console message that reads something like:



Shedding Some Light on the Situation

While the Bartok.PassTurn() Console message from the previous lets us know whose turn it is while running Unity, our players won't have access to the Console. We'll need another way to show them whose turn it is. We will accomplish this by highlighting the background behind the current player with a light.

1. In Unity, choose *GameObject* > *Light* > *Point Light* from the menu bar to create a new Point Light.

2. Name the new light *TurnLight* and set it's transform to the following:



As you can see, this casts a nice, obvious light on the background. Now, we'll need to add code that will have it show who is the CURRENT_PLAYER.

3. Create a new script named TurnLight in the __Scripts folder of the Project pane.

4. Attach the TurnLight C# script to the GameObject TurnLight in the Hierarchy.

5. Open the TurnLight script and add the following code.

```
using UnityEngine;
using System.Collections;

public class TurnLight : MonoBehaviour {

    void Update () {
        transform.position = Vector3.back*5;                    // a

        if (Bartok.CURRENT_PLAYER == null) {                    // b
            return;
        }

        transform.position +=
Bartok.CURRENT_PLAYER.handSlotDef.pos;  // c
    }
}
```

a. This moves the light to its default position above the center of the board ([0,0,-5]).

b. If `Bartok.CURRENT_PLAYER` is null, then we're done.

c. If `Bartok.CURRENT_PLAYER` is not null, then just add the position of the current Player to move the light above it.

In the previous edition of the book, the code to move TurnLight was part of the Bartok class, but in the time since I wrote that edition, I've been moving more towards component-based code, the core idea of which is to separate out your code into smaller chunks that have simpler purposes. There's no reason for the Bartok script to even know that a light exists, so we have the light manage itself.

6. Save the TurnLight script, return to Unity, and press play. Now, once the cards are dealth, you should see the TurnLight move to hover over the left player, signifying that it is that player's turn.

A Simple Bartok AI

Now, let's make the AI players able to take turns.

1. Open the Bartok script and look for the line that was marked `// g` in the code listing under the heading Handling Turns a few pages back. Remove the comment slashes from the beginning of that line. After doing so, the line should read:

```
CURRENT_PLAYER.TakeTurn();                    // g
```

2. Save the Bartok script.

3. Open the Player script and add the following bolded code:

```
public class Player {
    ...

    public void FanHand() {
        ...
        Quaternion rotQ;
        for (int i=0; i<hand.Count; i++) {
            ...
            pos += handSlotDef.pos;
            pos.z = -0.5f*i;

            // If  not the initial deal, move the card immediately.
            if (Bartok.S.phase != TurnPhase.idle) {          // a
                hand[i].timeStart = 0;
            }

            // Set the localPosition and rotation of the ith card in the
            // hand
```

```
            hand[i].MoveTo(pos, rotQ); // Tell CardBartok to interpolate
            ...
        }

    }

    // The TakeTurn() function enables the AI of the computer
Players
    public void TakeTurn() {
        Utils.tr ("Player.TakeTurn");

        // Don't need to do anything if this is the human player.
        if (type == PlayerType.human) return;

        Bartok.S.phase = TurnPhase.waiting;

        CardBartok cb;

        // If this is an AI player, need to make a choice about what
to
        // play
        // Find valid plays
        List<CardBartok> validCards = new List<CardBartok>
();         // b
        foreach (CardBartok tCB in hand) {
            if (Bartok.S.ValidPlay(tCB)) {
                validCards.Add ( tCB );
            }
        }
        // If there are no valid cards
        if (validCards.Count == 0) {                          // c
            // ...then draw a card
            cb = AddCard( Bartok.S.Draw () );
            cb.callbackPlayer = this;                         // e
            return;
        }
```

```
        // So, there is a card or more to play, so pick one
        cb = validCards[ Random.Range (0,validCards.Count)
];        // d
        RemoveCard(cb);
        Bartok.S.MoveToTarget(cb);
        cb.callbackPlayer  = this;                              // e

    }

    public void CBCallback(CardBartok tCB) {
        Utils.tr ("Player.CBCallback()",tCB.name,"Player
"+playerNum);
        // The card is done moving, so pass the turn
        Bartok.S.PassTurn();
    }

}
```

a. Though we want cards to move in a staggered way during the initial deal at the start of the game, we don't want any delays later, so this makes sure the card gets moving.

b. Here, the AI looks for valid plays. It calls `ValidPlay()` on each card in its hand, and if the card is a valid play, it adds the card to a list `validCards`.

c. If the count of `validCards` is 0 (i.e., there are no valid plays), then the AI will draw a card and return.

d. If there are valid cards to pick from, the AI chooses one at random and makes it the new target card (i.e., it plays it to the discard pile).

e. `callbackPlayer` is red on these two lines because we have not yet added a public `callbackPlayer` field to CardBartok.

4. Save the Player script.

At the end of the Player script, we added a `CBCallback()` function that a CardBartok should call when it's done moving; however, because Player does not extend MonoBehaviour, we cannot use `SendMessage()` to call `CBCallback()`. Instead, we'll pass the CardBartok a reference to this Player, and then the CardBartok can call `CBCallback` directly on the Player instance. This Player reference will be stored on CardBartok as the field `callbackPlayer`.

5. Open CardBartok and add this code:

```
public class CardBartok : Card {
    ...
    [Header("Set Dynamically")]
    ...
    public GameObject        reportFinishTo = null;
    [System.NonSerialized]                                  // a
    public Player            callbackPlayer = null;             // b

    // MoveTo tells the card to interpolate to a new position and
rotation
    public void MoveTo(Vector3 ePos, Quaternion eRot) { ... }
    ...

    void Update() {
        switch (state) {
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
                ...
                if (u<0) {
                    ...
                } else if (u>=1) {
                    ...
                    if (reportFinishTo != null) {
                        reportFinishTo.SendMessage("CBCallback", this);
                        reportFinishTo = null;
                    } else if (callbackPlayer != null) {          // c
```

```
                    // If there's a callback Player
                    // Call CBCallback directly on the Player
                    callbackPlayer.CBCallback(this);
                    callbackPlayer = null;
                } else { // If there is nothing to callback
                    // Just let it stay still.
                }
            } else {
                ...
            }
            break;

        }
    }
}
```

a. As with `[System.Serialized]`, `[System.NonSerialized]` affects the line below it. In this case, we are asking that the `callbackPlayer` field not be serialized, meaning two things: 1. It will not appear in the Inspector, and 2. it will not be given a value by the Inspector. The second one is most important in this case. See b. that follows for why.

b. Now that we've defined `callbackPlayer`, it will no longer be red in the Player script.

c. Here we will only call `callbackPlayer.CBCallback()` if `callbackPlayer` is not null. This is why we needed `callbackPlayer` to be NonSerialized. If we allowed the Inspector to serialize `callbackPlayer`, it would have created a new Player instance for `callbackPlayer` so that it could be shown in the Inspector. To say that a different way, if `callbackPlayer` were serialized by the Inspector, it would be set to something other than null before the game even started. We make `callbackPlayer` NonSerialized to prevent this from happening. To test this, you can try commenting out the `[System.NonSerialized]` line and playing the game. You will

exceptions as a result because CardBartoks are trying to call `CBCallback()` on invalid Players that were assigned by the Inspector.

6. Save the CardBartok script and return to Unity.

Now, you'll see that when you play the scene, the three AI players each play their turn.

Enabling the Human Player

It's time to make the human able to play as well. We do this by making the cards clickable.

1. Add the following bolded code to the end of the CardBartok class:

```
public class CardBartok : Card {

    ...

    void Update() { ... }

    // This allows the card to react to being clickced
    override public void OnMouseUpAsButton() {
        // Call the CardClicked method on the Bartok singleton
        Bartok.S.CardClicked(this);                    // a
        // Also call the base class (Card.cs) version of this method
        base.OnMouseUpAsButton();
    }

}
```

a. `CardClicked` is red here because we have not yet added the `CardClicked()` method to the Bartok class.

2. Save the CardBartok script.

3. Now add the `CardClicked()` method to the end of the Bartok script:

```
public class Bartok : MonoBehaviour {
    ...
    public CardBartok Draw() { ... }

    public void CardClicked(CardBartok tCB) {
        if (CURRENT_PLAYER.type != PlayerType.human)
return;         // a
        if (phase == TurnPhase.waiting) return;                    // b

        switch (tCB.state) {                                       // c
            case CBState.drawpile:                                 // d
                // Draw the top card, not necessarily the one clicked.
                CardBartok cb = CURRENT_PLAYER.AddCard( Draw()
);
                cb.callbackPlayer = CURRENT_PLAYER;
                Utils.tr ("Bartok.CardClicked()","Draw",cb.name);
                phase = TurnPhase.waiting;
                break;
            case CBState.hand:                                     // e
                // Check to see whether the card is valid
                if (ValidPlay(tCB)) {
                    CURRENT_PLAYER.RemoveCard(tCB);
                    MoveToTarget(tCB);
                    tCB.callbackPlayer = CURRENT_PLAYER;
                    Utils.tr("Bartok.CardClicked()","Play",tCB.name,
                        ImagetargetCard.name+" is target");
                    phase = TurnPhase.waiting;
                } else {
                    // Just ignore it but report what the player tried
                    Utils.tr("Bartok.CardClicked()","Attempted to Play",
                        ImagetCB.name,targetCard.name+" is target");
                }
                break;
        }
    }
}
```

a. If it's not the human's turn, don't respond to the click at all, just return.

b. If the game is waiting on a card to move, don't respond. This forces the player to wait until the game is still before playing.

c. This switch statement acts differently based on whether the clicked card was a card in the player's hand or on the `drawPile`.

d. If the card clicked was in the `drawPile`, draw the top card of the `drawPile`. Because the sprites of the `drawPile` are not sorted using a sort order or anything, this may not actually be the card that was clicked.

e. If the card clicked was in the player's hand, check to see if it was a valid play. If it was valid, play the card to the target (discard pile). If it was invalid, ignore the click, but report the attempt to the Console.

4. Save the Bartok script, return to Unity, and press play.

Now, you can play as well, and the game works! But right now there is no logic to end the game when it's over. Just a few more additions, and this prototype will be done.

Adding Game UI

Just as with Prospector, we want to message the player when she finishes the game. To make it possible to do so, we'll need to create some uGUI Text fields.

1. From the menu bar, choose *GameObject* > *UI* > *Text* to add a new Text as a child of Canvas in the Hierarchy.

2. Rename this Text to *GameOver* and give it the settings shown on the left of Figure 34.6.

Figure 33.6. Settings for GameOver and RoundResult



3. Duplicate GameOver by selecting it and choosing Edit > Duplicate from the menu bar.

4. Rename *GameOver (1)* to *RoundResult*, and give it the settings shown on the right of Figure 33.6. Often when editing the Min and Max Anchors values, Unity will change the Pos X and Pos Y values as well. You can limit this by checking the [R] button in the RectTransform as I have, but Unity still messes with it a bit even after you do that.

Just like the TurnLight, we can give each of these Texts its own script so that Bartok doesn't have to worry about them.

5. Create a C# script named GameOverUI inside the __Scripts folder of the Project pane, attach it to GameOver in the Hierarchy, and give it this code:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class GameOverUI : MonoBehaviour {
    private Text    txt;

    void Awake() {
        txt = GetComponent<Text>();
        txt.text = "";
    }

    void Update () {
        if (Bartok.S.phase != TurnPhase.gameOver) {
            txt.text = "";
            return;
        }
        // We only get here if the game is over
        if (Bartok.CURRENT_PLAYER == null) return;                  //
```

```
a
        if (Bartok.CURRENT_PLAYER.type == PlayerType.human) {
            txt.text = "You won!";
        } else {
            txt.text = "Game Over";
        }
    }
}
```

a. `Bartok.CURRENT_PLAYER` is null at the beginning of the game, so we need to accommodate that case.

6. Save the GameOverUI script.

7. Create a C# script named RoundResultUI inside the __Scripts folder of the Project pane, attach it to RoundResult in the Hierarchy, and give it this code:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class RoundResultUI : MonoBehaviour {
    private Text    txt;

    void Awake() {
        txt = GetComponent<Text>();
        txt.text = "";
    }

    void Update () {
        if (Bartok.S.phase != TurnPhase.gameOver) {
            txt.text = "";
            return;
        }
        // We only get here if the game is over
        Player cP = Bartok.CURRENT_PLAYER;
        if (cP == null || cP.type == PlayerType.human) {           // a
```

```
            txt.text = "";
        } else {
            txt.text = "Player "+(cP.playerNum+1)+" won";          // b
        }
    }
}
```

a. Remember that || (logical OR) is a shorting function, so if `cP` is null, this line will never ask for `cP.type` and won't encounter a null reference exception.

b. We add 1 to the playerNum because most players will count themselves as player 1 in the game (not player 0).

8. Save the RoundResultUI script.

Game Over Logic

Now that we have the UI to show game over messages, let's actually allow the game to realize that it's over.

1. Open the Bartok script and add this bolded code to manage finishing the game.

```
public class Bartok : MonoBehaviour {
    ...

    public void PassTurn(int num=-1) {
        ...
        if (CURRENT_PLAYER != null) {
            lastPlayerNum = CURRENT_PLAYER.playerNum;
            // Check for Game Over and need to reshuffle discards
            if ( CheckGameOver() ) {
                return;                                    // a
            }
        }
        ...
```

```
    }

    public bool CheckGameOver() {
        // See if we need to reshuffle the discard pile into the draw
pile
        if (drawPile.Count == 0) {
            List<Card> cards = new List<Card>();
            foreach (CardBartok cb in discardPile) {
                cards.Add (cb);
            }
            discardPile.Clear();
            Deck.Shuffle( ref cards );
            drawPile = UpgradeCardsList(cards);
            ArrangeDrawPile();
        }

        // Check to see if the current player has won
        if (CURRENT_PLAYER.hand.Count == 0) {
            // The player that just played has won!
            phase = TurnPhase.gameOver;
            Invoke("RestartGame", 1);                          // b
            return(true);
        }

        return(false);
    }

    public void RestartGame() {
        CURRENT_PLAYER = null;
        SceneManager.LoadScene("__Bartok_Scene_0");
    }


    // ValidPlay verifies that the card chosen can be played on the
    // discard pile
    public bool ValidPlay(CardBartok cb) { ... }
    ...
```

}

a. If the game is over, we return before advancing the turn. This leaves `CURRENT_PLAYER` set to the player who won, which allows GameOverUI and RoundResultUI to read from it.

b. We invoke `RestartGame()` in 1 second, which will show the results for a second before restarting the game.

2. Save the Bartok script, return to Unity, and press play.

Now the game will play properly, it will end when it's over, and it will restart properly as well.

Summary

The goal of this chapter was to demonstrate how possible it is to take the digital prototypes that you make in this book and adapt them to your own games. Once you finish all the tutorial chapters, you will have the framework for a classic arcade game (Apple Catcher), a physics-based casual game (Mission Demolition), a space shooter (Space SHMUP), a card game (Prospector and Bartok), a word game (Word Game), a first-person shooter (Quick Snap), and a third-person adventure game (Omega Mage). As prototypes, none of these are finished games, but any of them could serve as a foundation on which to build your own games.

Next Steps

The classic paper version of the Bartok card game included the ability for the winner of any round to add additional rules to the game. While it's not possible to allow the player to just make up rules for this digital game, it is certainly possible to add your own optional rules through code just as I did for the version you played with in Chapter 1.

If you visit the http://book.prototools.net website, you can look under Chapter 32 for the Unity project of the expanded version of Bartok that includes all the optional rules you were able to play with in Chapter 1. That should be a good starting point for you to use to add your own rules to the game.

Chapter 34. Protype 6: Word Game

In this chapter, you learn how to create a simple word game. This game uses several concepts that you have already learned, and it introduces the concept of *coroutines*, methods that can yield during execution to allow the processor to handle other methods.

By the end of this chapter, you'll have a fun word game that you can expand yourself.

Getting Started: Word Game Prototype

As usual, you'll import a unitypackage to start this chapter. This package contains a few art assets and some C# Scripts that you created in previous chapters.

Set Up the Project for this Chapter

Following the standard project setup procedure, create a new project in Unity. If you need a refresher on the standard project setup procedure, see Appendix A, "Standard Project Setup Procedure." When you are creating the project, you will be asked if you want to set up defaults for 2D or 3D. Choose 3D for this project.

For this project, we will import the main scene from the unitypackage, so you do not need to set up the _MainCamera.

• **Project name:** Word Game

• **Download and import package:** See Chapter 34 at http://book.prototools.net

• **Scene name:** _WordGame_Scene_0 (imported in unitypackage)

• **Project folders:** _Scripts, _Prefabs, Materials & Textures, Resources

• **C# script names:** Just the imported scripts in the ProtoTools folder

---

Open the scene __WordGame_Scene_0, and you will find a _MainCamera that is already set up for an orthographic game. You'll also notice that some of the reusable C# scripts that were created in previous chapters have been moved into the folder __Scripts/ProtoTools to keep them separate from the new scripts you'll create for this project. I find this is useful because it enables me to just place a copy of the ProtoTools folder into the __Scripts folder of any new project and have all that functionality ready to go.

In your Build Settings, make sure that this one is set to *PC, Mac, & Linux Standalone*. Set the aspect ratio of the Game pane to *Standalone (1024 x 768)*.

About the Word Game

This game is a classic form of word game. Commercial examples of this game include *Word Whomp* by Pogo.com, *Jumbline 2* by Branium, *Pressed for Words* by Words and Maps, and many others. The player will be presented with six letters that spell at least one word of a certain length (usually six letters), and she is tasked with finding all of the words that can be created with the letters that form that word. Our version of the game will include some slick animations (using Bézier interpolations) and a scoring paradigm that encourages the player to find long words before short ones. Figure 34.1 shows an image of the game you'll create.

Figure 34.1. An image of the Word Game created in this chapter using an 8-letter word as the base



In this image, you can see that each of the words are composed of individual letter tiles, and there are two sizes of letter tiles. For the sake of object orientation, we'll create a `Letter` class that handles each letter and a `Word` class to collect them into words. We'll also create a `WordList` class to read the large dictionary of possible words that we have and turn it into usable data for the game. The game will be by a `WordGame` class, and the `Scoreboard` and `FloatingScore` classes from previous prototypes will be used to show the score to the player. In addition, the `Utils` class will be used for interpolation and easing. The `PT_XMLReader` class is imported with this project, but is unused. I left this script in the unitypackage because I want to encourage you to start building your own collection of useful scripts that you can import into any project to help you get started (just as the ProtoTools folder is for the projects in this book). Feel free to add any useful scripts that you create to this collection, and think about importing it as the first thing you do for each new game prototype that you start.

Parsing the Word List

This game uses a modified form of the *2of12inf* word list created by Alan Beale.[1] I've removed some offensive words and attempted to correct others. You are more than welcome to use this word list however you wish in the future, as long as you follow the copyright wishes of both Alan Beale and Kevin Atkinson (as listed in footnote #1). I also modified the list by shifting all of the letters to uppercase and by changing the line ending from `\r\n` (a carriage return and a line feed, which is the standard Windows text file format) to `\n` (just a line feed, the standard Macintosh text format). This was done because it makes it easier to split the file into individual words based on line feed, and for our purposes, it will work on Windows just as well as Mac.

[1] Alan Beale has released all of his word lists into the public domain apart from the aspects of the 2of12inf list that were based on the AGID word list, Copyright 2000 by Kevin Atkinson. Permission to

use, copy, modify, distribute and sell this [the AGID] database, the associated scripts, the output created form the scripts and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Kevin Atkinson makes no representations about the suitability of this array for any purpose. It is provided “as is” without express or implied warranty.

The decision to remove offensive words was based on the kind of game this is. In a game like *Scrabble* or *Letterpress*, the player is given a series of letter tiles, and she is able to choose which words she wishes to spell with those tiles. If this game were of that ilk, I would not have removed any words from the word list. However, in this game, the player is forced to spell every word in the list that can be made from the collection of letters that she is given. This means that the game could force players to spell some terms that would be very offensive to them. In this game, the decision of which words are chosen has shifted from the player to the computer, and I did not feel comfortable forcing players to spell potentially offensive words. However, in the over 75,000 words in the list, there are probably some words that I missed, so if you find any words in the game that you feel I should omit (or ones I should add), please let me know by sending me a message via the website http://book.prototools.net. Thanks.

To read the word list file, we need to pull its text into an array of strings, split by `\n`, and to do that, we’ll make use of a *coroutine* (see the sidebar “Using Coroutines”).

Using Coroutines

A coroutine is a function that can pause in its execution to allow other functions to update. It’s a Unity C# feature that allows developers to have control over repeating tasks in code or to handle very large tasks. In this chapter, we’ll use it for the latter by having a coroutine parse all 75,000 words from the *2of12inf* word list.

Coroutines are initiated with a call to `StartCoroutine()`, which can only be called within a class that extends MonoBehaviour. Once initiated in this way, a corountine executes until it encounters a `yield` statement. yield tells the coroutine to wait for a certain amount of time and allow other code to execute. This means that you could have an infinite while(true) {} loop in a coroutine, and it wouldn't freeze your game as long as there was a yield statement somewhere within the while loop. In this game, the coroutine `ParseLines()` will yield every 10,000 words that it parses.

Look for how the coroutine is used in the fist code listing of this chapter. While the coroutine in this chapter probably isn't strictly necessary as long as you have a very fast computer, this kind of thing becomes much more important when you're developing for mobile devices (or other devices with slower processors). Parsing this same word list on an older iPhone can take as much as 10 to 20 seconds, so it's important to include breaks in the parsing where the app can handle other tasks and not just look frozen.

You can learn more about coroutines in the Unity documentation.

---

1. Create a new C# script named *WordList* in the __Scripts folder and enter the following code:

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

public class WordList : MonoBehaviour {
    private static WordList   S;                              // a

    [Header("Set in Inspector")]
    public TextAsset        wordListText;
    public int              numToParseBeforeYield = 10000;
    public int              wordLengthMin = 3;
    public int              wordLengthMax = 7;
```

```
   [Header("Set Dynamically")]
   public int            currLine = 0;
   public int            totalLines;
   public int            longWordCount;
   public int            wordCount;

   // Private fields
   private string[]      lines;                          // b
   private List<string>  longWords;
   private List<string>  words;

   void Awake() {
      S = this;  // Set up the WordList Singleton
   }

   void Start () {
      lines = wordListText.text.Split('\n');             // c
      totalLines = lines.Length;

      StartCoroutine( ParseLines() );                    // d
   }

   // All coroutines have IEnumerator as their return type.
   public IEnumerator ParseLines() {                     // e
      string word;
      // Init the Lists to hold the longest words and all valid
words
      longWords = new List<string>();                    // f
      words = new List<string>();

      for (currLine = 0; currLine < totalLines; currLine++) {      //
g
         word = lines[currLine];

         // If the word is as long as wordLengthMax...
         if (word.Length == wordLengthMax) {
            longWords.Add(word);    // ...then store it in
```

```
longWords
            }
            // If it's between wordLengthMin and wordLengthMax in
length...
            if ( word.Length>=wordLengthMin &&
word.Length<=wordLengthMax ) Image{
                words.Add(word);  // ...then add it to the list of all valid
                                  // words
            }

            // Determine whether the coroutine should yield
            if (currLine % numToParseBeforeYield == 0) {            // h
                // Count the words in each list to show that the parsing
is
                //  progressing
                longWordCount = longWords.Count;
                wordCount = words.Count;
                // This yields execution until the next frame
                yield return null;                                  // i

                // The yield will cause the execution of this method to
wait
                // here while other code executes and then continue
from this
                // point into the next iteration of the for loop.
            }
        }
    }

// These methods allow other classes to access the private
List<string>s // j
    static public List<string> GET_WORDS() {
        return( S.words );
    }

    static public string GET_WORD(int ndx) {
        return( S.words[ndx] );
```

```
    }

    static public List<string> GET_LONG_WORDS() {
        return( S.longWords );
    }

    static public string GET_LONG_WORD(int ndx) {
        return( S.longWords[ndx] );
    }

    static public int WORD_COUNT {
        get { return S.wordCount; }
    }

    static public int LONG_WORD_COUNT {
        get { return S.longWordCount; }
    }

    static public int NUM_TO_PARSE_BEFORE_YIELD {
        get { return S.numToParseBeforeYield; }
    }

    static public int WORD_LENGTH_MIN {
        get { return S.wordLengthMin; }
    }

    static public int WORD_LENGTH_MAX {
        get { return S.wordLengthMax; }
    }
}
```

a. This is the first private Singleton that we've seen in the book (because it's private, it is not exactly a Singleton any more). Making the singleton `s` private ensures that only instances of the WordList class can see it, protecting it from other code. This private Singleton is used by the accessors discussed in `// j` below.

b. Because these fields are private, they will not appear in the Inspector. These variables will be so long that it can drastically slow playback if the Inspector is trying to display them, so we will make these private—restricting their access to this instance of WordList—and make accessor functions at the end of the class to allow code outside of this instance to access them.

c. Split the text of wordListText on line feeds (`\n`), which creates a large, populated string[] with all the words from the list.

d. This starts the coroutine `ParseLines()`. See the sidebar "Using Coroutines" for more info.

e. All coroutines must have the return type of `IEnumerator`. This enables them to yield their execution and allow other methods to run before returning to the coroutine, which is extremely important for processes like loading large files or like parsing a large amount of data (as we're doing in this case).

f. The string array `lines[]` will be sorted into two lists: `longWords` is for all the words that are composed of `wordLengthMax` characters, and `words` is for all the words that are between `wordLengthMin` and `wordLengthMax` (inclusive) characters. For example, if `wordLengthMin` were 3 characters and `wordLengthMax` were 6, DESIGN would be in `longWords`, while DIE, DICE, GAME, BOARD, and DESIGN would be in `words`.

We parse the whole list here so that the player only has to wait once and is then able to play many rounds in a row with many different words.

g. This for loop will iterate over all 75,000 entries in `lines[]`. Every `numToParseBeforeYield` words, the yield statement will pause this for loop and allow other code to run. Then, on the next frame, execution will return to the for loop for another `numToParseBeforeYield` lines.

h. Determine whether the coroutine should yield This uses a modulus (%) function to yield every 10,000th record (or whatever

you have `numToParseBeforeYield` set to)

i. This yield statement will yield execution of the coroutine until the next frame because it returns null. It is also possible to yield for a certain number of seconds with a statement like `yield return new WaitForSeconds(1);` , which would wait for at least 1 second before continuing coroutine execution (note that coroutine yield times are pretty accurate but not exact). This also means that you can put repeating tasks in a coroutine rather than using the `InvokeRepeating()` method.

j. The four methods below the `// i` line are static public *accessors* for the private fields `words[]` and `longWords[]`. Code anywhere in the game can call `WordList.GET_WORD(10)` to get the tenth word in the private `words[]` array of this singleton instance of WordList. Additionally, the last several accessors are read only static public properties, showing another way to access private variables of WordList. By convention, static variables and methods are named with ALL_CAPS_SNAKE_CASE.

2. Once the code is written and saved, switch back to Unity.

3. Attach the WordList C# script to _MainCamera.

4. Select _MainCamera in the Hierarchy and set the `wordListText` variable of the *WordList (Script)* component in the Inspector to be the file *2of12inf*, which you can find in the Resources folder of the Project pane.

5. Press Play. You will see that the `currLine`, `longWordCount`, and `wordCount` will count up progressively by 10,000s. This is happening because the numbers are allowed to update every time the coroutine `ParseLines()` yields.

If you stop, use the Inspector to change `numToParseBeforeYield` to 100, and play again, you will see that these numbers build much more slowly because the coroutine is yielding every 100 words. However, if you change it to something like 100000, these numbers

will update only once because there are fewer than 100,000 words in the word list. If you're interested in seeing how much time each pass through the `ParseLines()` coroutine is taking, try using the profiler, as described in the sidebar titled The Unity Profiler.

The Unity Profiler

The Unity profiler is one of the most powerful tools for optimizing the performance of your games, and it's one of the many free tools available to you in the free version of Unity. For every frame of your game, the profiler maintains stats on the amount of time spent on each C# function, calls to the graphics engine, handling user input, and so on. You can see a great example of how this works by running the profiler on this project.

First, make sure that the WordList code from the preceding pages is working properly. Next, we'll add a Profiler pane to the same group as the Scene pane. That will ensure that you can see both the Game pane and the Profiler pane simultaneously. To add the Profiler pane, click the pop-up menu button at the top right of the current Scene pane and choose *Add Tab* > *Profiler* (as shown in Figure 34.2).

Figure 34.2. The Profiler pane



To see the profiler in action, first click the Pause button at the top of the Unity window and then click Play. This will cause Unity to prepare itself to run your game but to be paused before the first frame. If you click Pause again, you will see a graph start to appear in the profiler. Pause the game again when the graph is about an inch from the left side of the screen.

With the game paused, the profiler should stop graphing yet maintain the graph of the frames that have already passed. Each of the colors in the graph next to CPU Usage covers a different aspect of things for which the CPU (the main processor in your computer) is used. In

the later frames, if you're on a fast computer, you should see that most of the chart is yellow; the yellow represents the time Unity spends on VSync (that is, waiting for the screen to be ready to display another frame). This is blocking our view of how much time is taken by the scripts (which are light blue), so we'll hide it from the graph. The little colored boxes below CPU Usage on the left side of the profiler each represent a different kind of process that runs on the CPU. For our purposes now, you want to turn off all of them off except for the Scripts box (which is blue). To do this, click the colored box next to everything except for Scripts. This should leave you with a blue graph like the one shown in Figure 34.2.

Now, click and drag the mouse along the blue graph, and you should see a white line following the mouse. This white line represents a single frame in the graph. As you move, the text in the bottom half of the profiler will update to show how much processing time was taken by each function or background process during that frame. The function we're interested in is the `WordList.SetupCoroutine() [Coroutine: InvokeMoveNext]` coroutine. This only runs in the first few frames, so you won't see it on the right side of the graph; however, you should see a spike of script activity at the beginning of the graph (as shown in Figure 34.2), which is the time taken by the coroutine `ParseLines()`. Move the white line to that part of the graph and click `WordList.SetupCoroutine()` in the Overview column. This will highlight the graph contribution of that one routine and dim the others as shown in the figure. If you use the left and right arrow buttons at the top-right corner of the Profiler pane, you can step one frame back or forward (respectively) and see the CPU resources used by the coroutine in each frame. In my profiling shown in figure 34.2, I set `numToParseBeforeYield` to 1000 and found that for the first several frames, the coroutine took up about 6.7% of the CPU time spent on each frame (although your numbers may vary due to computer type and processing speed).

In addition to script profiling, the profiler can also help you find what aspects of rendering or physics simulation are taking the most time in your game. If you ever run into frame rate issues in one of your

games, try checking the profiler to see what's happening. (You'll want to be sure to turn all of the other types of CPU profiling back on when you do [that is, re-check all the boxes that we unchecked to isolate scripts]).

To see a very different profiler graph, you can try running the profiler on the Hello World project from Chapter 19, "Hello World." You'll see that in Hello World, much more time is spent on physics than scripts. (You may need to turn the VSync element of the graph off again to see this clearly.)

You can learn more about the profiler in the Unity documentation.

After playing with the Profiler, be sure to set `numToParseBeforeYield` back to 10000.

Setting Up the Game

We're going to create a WordGame class to manage the game, but before we do so, we need to make a couple changes to WordList. First, we need to make it not start parsing the words on `Start()` but instead wait until an `Init()` method is called by another class. Second, we need to make WordList notify the upcoming WordGame script when the parsing is complete. To do this, we will have the WordList send a message to the _MainCamera GameObject using the `SendMessage()` command. This message will be interpreted by WordGame as you'll soon see.

1. Change the name of the void `Start()` method in WordList to public void `Init()` and add the following bold code, including the static public void `INIT()` function and the lines at the end of the `ParseLines` method in WordList:

```
public class WordList : MonoBehaviour {
    ...
    void Awake() { ... }
```

```
    void Init () { // This line replaces "void Start()"
        lines = wordListText.text.Split('\n');
        totalLines = lines.Length;

        StartCoroutine( ParseLines() );
    }

    static public void INIT () {                    // a
        S.Init();
    }

    // All coroutines have IEnumerator as their return type.
    public IEnumerator ParseLines() {
        ...
        for (currLine = 0; currLine < totalLines; currLine++) {
            ...
        }

        // Send a message to this gameObject to let it know the
parse is done
        gameObject.SendMessage("WordListParseComplete");
         // b
    }
}
```

a. This `INIT()` method is static and public, meaning that the WordGame class will be able to call it.

b. The `SendMessage()` command is executed on the GameObject _MainCamera (because WordList is a Script Component of _MainCamera). This command will call a `WordListParseComplete()` method on any script that is attached to the GameObject on which it is called (that is, _MainCamera).

2. Create a WordGame C# script in the __Scripts folder and attach it to _MainCamera. Then enter the following code to take advantage of

the changes just made to WordList:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;   // We'll be using List<> & Dictionary<>
using System.Linq;                  // We'll be using LINQ

public enum GameMode {
    preGame,    // Before the game starts
    loading,    // The word list is loading and being parsed
    makeLevel,  // The individual WordLevel is being created
    levelPrep,  // The level visuals are Instantiated
    inLevel     // The level is in progress
}

public class WordGame : MonoBehaviour {
    static public WordGame    S; // Singleton

    [Header("Set Dynamically")]
    public GameMode          mode = GameMode.preGame;

    void Awake() {
        S = this; // Assign the singleton
    }

    void Start () {
        mode = GameMode.loading;
        // Call the static Init() method of WordList
        WordList.INIT();
    }

    // Called by the SendMessage() command from WordList
    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
    }
}
```

3. Select _MainCamera in the Hierarchy pane, and look at the WordGame Component in the Inspector. Press Play, and you'll see the value of the `mode` field initially move from `preGame` to `loading`. Then, after all the words have been parsed, it will change from `loading` to `makeLevel`. This shows us that everything is working as we had hoped.

Building a Level with the WordLevel Class

Now, it's time to take the words in the WordList and make a level from them. The `WordLevel` class will include the following:

• The long word on which the level is based. (If `maxWordLength` is 6, this is the six-letter word whose letters will be reshuffled into the other words.)

• The index number of that word in the `longWords[]` array of WordList.

• The level number as `int levelNum`. In this chapter, every time the game starts, we'll choose a random word.[2]

[2] If you wished, you could call `Random.InitState(1);` in the WordGame.Awake() method, which would set the initial random number seed of Random to 1 and ensure that the eighth level would always be the same word as long as you only ever used Random to choose the level. Another way to approach this is described in the Next Steps section at the end of the chapter.

• A `Dictionary<,>` of each character in the word and how many times it is used. Dictionaries are part of System.Collections.Generic along with Lists.

• A `List<>` of all the other words that can be formed from the characters in the Dictionary above.

A `Dictionary<,>` is a generic collection type that holds a series of key, value pairs that is covered in detail in Chapter 23, "Collections in C#." In each level, the `Dictionary<,>` will use char keys and int values to hold information about how many times each char is used in the long word. For example, this is how the long word MISSISSIPPI would look:

```
Dictionary<char,int> charDict = new Dictionary<char,int>();
charDict.Add('M',1); // MISSISSIPPI has 1 M
charDict.Add('I',4); // MISSISSIPPI has 4 Is
charDict.Add('S',4); // MISSISSIPPI has 4 Ss
charDict.Add('P',2); // MISSISSIPPI has 2 Ps
```

WordLevel will also contain two useful static methods:

• `MakeCharDict()`: Creates a charDict like the one above from any string.

• `CheckWordInLevel()`: Checks to see whether a word can be spelled using the chars in a WordLevel's charDict.

1. Create a new C# script named *WordLevel* in the __Scripts folder and enter the following code. Note that WordLevel *does not* extend MonoBehaviour, so it is not a class that can be attached to a GameObject as a Script component, and it cannot have `StartCoroutine()`, `SendMessage()`, or many other Unity-specific functions called within it.

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

[System.Serializable] // WordLevels can be viewed & edited in the Inspector
public class WordLevel { // WordLevel does NOT extend MonoBehaviour
    public int          levelNum;
    public int          longWordIndex;
```

```
   public string               word;
   // A Dictionary<,> of all the letters in word
   public Dictionary<char,int>    charDict;
   // All the words that can be spelled with the letters in charDict
   public List<string>          subWords;

   // A static function that counts the instances of chars in a
string and
   //  returns a Dictionary<char,int> that contains this
information
   static public Dictionary<char,int> MakeCharDict(string w)
{
      Dictionary<char,int> dict = new Dictionary<char, int>();
      char c;
      for (int i=0; i<w.Length; i++) {
         c = w[i];
         if (dict.ContainsKey(c)) {
            dict[c]++;
         } else {
            dict.Add (c,1);
         }
      }
      return(dict);
   }

   // This static method checks to see whether the word can be
spelled with
   //  the chars in level.charDict
   public static bool CheckWordInLevel(string str, WordLevel
level) {
      Dictionary<char,int> counts = new Dictionary<char, int>();
      for (int i=0; i<str.Length; i++) {
         char c = str[i];
         // If the charDict contains char c
         if (level.charDict.ContainsKey(c)) {
            // If counts doesn't already have char c as a key
            if (!counts.ContainsKey(c)) {
```

```
                // ...then add a new key with a value of 1
                counts.Add (c,1);
            } else {
                // Otherwise, add 1 to the current value
                counts[c]++;
            }
            // If this means that there are more instances of char c
in
            //  str than are available in level.charDict
            if (counts[c] > level.charDict[c]) {
                //  ... then return false
                return(false);
            }
        } else {
            // The char c isn't in level.word, so return false
            return(false);
        }
    }
    return(true);
  }
}
```

2. Now, to make use of this, make the following bolded changes to WordGame:

```
public class WordGame : MonoBehaviour {
  static public WordGame    S; // Singleton

  [Header("Set Dynamically")]
  public GameMode          mode = GameMode.preGame;
  public WordLevel         currLevel;

  ...

  public void WordListParseComplete() {
    mode = GameMode.makeLevel;
    // Make a level and assign it to currLevel, the current
```

```
WordLevel
        currLevel = MakeWordLevel();
    }

    public WordLevel MakeWordLevel(int levelNum = -1)
{               // a
        WordLevel level = new WordLevel();
        if (levelNum == -1) {
            // Pick a random level
            level.longWordIndex =
Random.Range(0,WordList.LONG_WORD_COUNT);
        } else {
            // This will be added later in the chapter
        }
        level.levelNum = levelNum;
        level.word =
WordList.GET_LONG_WORD(level.longWordIndex);
        level.charDict = WordLevel.MakeCharDict(level.word);

        StartCoroutine( FindSubWordsCoroutine(level) );             //
b

        return( level );                                    // c
    }

    // A coroutine that finds words that can be spelled in this level
    public IEnumerator FindSubWordsCoroutine(WordLevel
level) {
        level.subWords = new List<string>();
        string str;

        List<string> words =
WordList.GET_WORDS();                   // d

        // Iterate through all the words in the WordList
        for (int i=0; i<WordList.WORD_COUNT; i++) {
            str = words[i];
```

```
            // Check whether each one can be spelled using
level.charDict
            if (WordLevel.CheckWordInLevel(str, level)) {
                level.subWords.Add(str);
            }
            // Yield if we've parsed a lot of words this frame
            if (i%WordList.NUM_TO_PARSE_BEFORE_YIELD == 0) {
                // yield until the next frame
                yield return null;
            }
        }

        level.subWords.Sort ();                                    // e
        level.subWords =
SortWordsByLength(level.subWords).ToList();

        // The coroutine is complete, so call
SubWordSearchComplete()
        SubWordSearchComplete();
    }

// Use LINQ to sort the array received and return a
copy                // f
    public static IEnumerable<string>
SortWordsByLength(IEnumerable<string> w Images) {
        ws = ws.OrderBy(s => s.Length);
        return ws;
    }

    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;

    }

}
```

a. With the default value of -1, this method will pick a random word from which to generate a WordLevel.

b. This starts a coroutine to check all the words in the WordList and see whether each word can be spelled by the chars in `level.charDict`.

c. This returns the WordLevel `level` before the coroutine finishes, so `SubWordSearchComplete()` will be called when the coroutine is done.

d. This is very fast because List<string>s are passed by reference.

e. These two lines sort the words in our WordLevel.subWords List. `List<string>.Sort()` sorts the words alphabetically (because that is the default for List<String>). Then, the custom SortWordsByLength() method is called to sort the words by the number of characters in each word. This will group alphabetized words of the same length.

f. Our custom sorting function uses LINQ to sort the array received and return a copy. The LINQ syntax inside this method is different from regular C# and is beyond the scope of this book. You can learn more about it by searching for “C# LINQ” online.

There is also a good explanation of LINQ at the Unity Gems website. This link is from the Internet Archive to ensure that it remains valid.

https://web.archive.org/web/20140209060811/http://unitygems.com/linq-1-time-linq/

This code creates the level, chooses a goal word, and populates it with subWords that can be spelled using the characters in the goal word. When you press Play, you should now see the `currLevel` field populate in the _MainCamera Inspector.

3. Save your scene! If you haven’t been saving your scene all along—and this served as a reminder to do so—you need to be reminding yourself to save more often.

Laying Out the Screen

Now that the level has been created, it's time to generate on-screen visuals to represent both the big letters that can be used to spell words and the regular letters of the words. To start, you need to create a PrefabLetter to be instantiated for each letter.

Making PrefabLetter

Follow these steps to make PrefabLetter:

1. From the menu bar, choose *GameObject* > *3D Object* > *Quad*. Rename the quad to *PrefabLetter*.

2. From the menu bar, choose *Assets* > *Create* > *Material*. Name the material *LetterMat* and place it in the *Materials & Textures* folder.

3. Drag LetterMat onto PrefabLetter in the Hierarchy to assign it. Click on PrefabLetter, and set the *shader* of LetterMat to *ProtoTools* > *UnlitAlpha*.

4. Select *Rounded Rect 256* as the texture for the LetterMat material (you may need to open the disclosure triangle in the LetterMat area of the PrefabLetter Inspector).

5. Double-click PrefabLetter in the Hierarchy, and you should now see a nice rounded rectangle there. If you can't see it, you may need to move the camera around to the other side. (Backface culling makes quads visible only from one side and invisible from the other.)

6. Right-click PrefabLetter in the Hierarchy and choose *GameObject* > *Create Other* > *3D Text* from the pop-up menu. This will create a New Text GameObject as a child of PrefabLetter.

7. Change the name of the *New Text* child GameObject to *3D Text*. Then select 3D Text in the Hierarchy and give it the settings in Figure 34.3. If the W doesn't line up in the center of the box, you may have

accidentally typed a tab in the Text box after the W (as I did when writing this).

Figure 34.3. The Inspector settings for 3D Text, a child of PrefabLetter



8. Once PrefabLetter is ready, drag it into the _Prefabs folder in the Project pane and delete the remaining instance from the Hierarchy.

The Letter C# Script

PrefabLetter will have its own C# script to handle setting the character it shows, its color, and various other things.

1. Create a new C# script named *Letter*, place it in the __Scripts folder, and attach it to PrefabLetter.

2. Open the Letter script in MonoDevelop and enter the following code:

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

public class Letter : MonoBehaviour {
    [Header("Set Dynamically")]
    public TextMesh       tMesh; // The TextMesh shows the char
    public Renderer       tRend; // The Renderer of 3D Text. This
will
                          //   determine whether the char is visible
    public bool           big = false; // Big letters act a little
                          //  differently

    private char          _c;    // The char shown on this Letter
    private Renderer      rend;
```

```
    void Awake() {
        tMesh = GetComponentInChildren<TextMesh>();
        tRend = tMesh.GetComponent<Renderer>;
        rend = GetComponent<Renderer>();
        visible = false;
    }

    // Property to get or set _c and the letter shown by 3D Text
    public char c {
        get { return( _c ); }
        set {
            _c = value;
            tMesh.text = _c.ToString();
        }
    }

    // Gets or sets _c as a string
    public string str {
        get { return( _c.ToString() ); }
        set { c = value[0]; }
    }

    // Enables or disables the renderer for 3D Text, which causes
the char to
    //  be visible or invisible respectively.
    public bool visible {
        get { return( tRend.enabled ); }
        set { tRend.enabled = value; }
    }

    // Gets or sets the color of the rounded rectangle
    public Color color {
        get { return(rend.material.color); }
        set { rend.material.color = value; }
    }

    // Sets the position of the Letter's gameObject
```

```
    public Vector3 pos {
        set {
            transform.position = value;
            // More will be added her later
        }
    }
}
```

This class makes use of several properties (faux fields with get{} and set{}) to perform various actions when variables are set. This enables, for instance, WordGame to set the char `c` of a Letter without worrying about how that gets converted to a string and then shown by 3D Text. This kind of encapsulation of functionality within a class is central to object-oriented programming. You'll notice that if the get{} or set{} clause is just one statement, I'll often collapse it to a single line.

The Wyrd Class: A Collection of Letters

The Wyrd class will act as a collection of Letters, and its name is spelled with a *y* to differentiate it from the other instances of the word *word* throughout the code and the text of the book. Wyrd is another class that does not extend MonoBehaviour and cannot be attached to a GameObject, but it can still contain `List<>`s of classes that are attached to GameObjects.

1. Create a new C# script named *Wyrd* inside the __Scripts folder.

2. Open Wyrd in MonoDevelop and enter the following code:

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

public class Wyrd {
    public string       str; // A string representation of the word
    public List<Letter> letters = new List<Letter>();
```

```
   public bool        found = false; // True if the player has found
this
                                  //  word

   // A property to set visibility of the 3D Text of each Letter
   public bool visible {
      get {
         if (letters.Count == 0) return(false);
         return(letters[0].visible);
      }
      set {
         foreach( Letter l in letters) {
            l.visible = value;
         }
      }
   }

   // A property to set the rounded rectangle color of each Letter
   public Color color {
      get {
         if (letters.Count == 0) return(Color.black);
         return(letters[0].color);
      }
      set {
         foreach( Letter l in letters) {
            l.color = value;
         }
      }
   }

   // Adds a Letter to letters
   public void Add(Letter l) {
      letters.Add(l);
      str += l.c.ToString();
   }

}
```

The WordGame.Layout() Method

The `Layout()` method will generate Wyrds and Letters for the game as well as big Letters that the player will use to spell each of the words in the level (shown as large gray letters at the bottom of Figure 34.1). We'll start with the small letters, and for this phase of the prototype, we'll make the character of each Letter initially visible (rather than hiding it as we'll do in the final version).

1. Add the following bold code to WordGame:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Singleton

    [Header("Set in Inspector")]
    public GameObject        prefabLetter;
    public Rect              wordArea = new Rect(-24,19,48,28);
    public float             letterSize = 1.5f;
    public bool              showAllWyrds = true;

    [Header("Set Dynamically")]
    public GameMode          mode = GameMode.preGame;
    public WordLevel         currLevel;
    public List<Wyrd>        wyrds;

    private Transform        letterAnchor, bigLetterAnchor;

    void Awake() {
        S = this; // Assign the singleton
        letterAnchor = new
GameObject("LetterAnchor").transform;
        bigLetterAnchor = new
GameObject("BigLetterAnchor").transform;
    }

    ...
```

```
    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;
        Layout(); // Call the Layout() function once WordSearch is
done
    }

    void Layout() {
        // Place the letters for each subword of currLevel on screen
        wyrds = new List<Wyrd>();

        // Declare a lot of variables that will be used in this method
        GameObject go;
        Letter lett;
        string word;
        Vector3 pos;
        float left = 0;
        float columnWidth = 3;
        char c;
        Color col;
        Wyrd wyrd;

        // Determine how many rows of Letters will fit on screen
        int numRows =
Mathf.RoundToInt(wordArea.height/letterSize);

        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            wyrd = new Wyrd();
            word = currLevel.subWords[i];

            // if the word is longer than columnWidth, expand it
            columnWidth = Mathf.Max( columnWidth, word.Length );

            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                c = word[j]; // Grab the jth char of the word
                go = Instantiate(prefabLetter) as GameObject;
```

```
            go.transform.SetParent(letterAnchor);
            lett = go.GetComponent<Letter>();
            lett.c = c; // Set the c of the Letter
            // Position the Letter
            pos = new Vector3(wordArea.x+left+j*letterSize,
wordArea.y, Ê0);
            // The % here makes multiple columns line up
            pos.y -= (i%numRows)*letterSize;
            lett.pos = pos;    // You'll add more code around this
line
                        // later
            go.transform.localScale = Vector3.one*letterSize;
            wyrd.Add(lett);
         }

         if (showAllWyrds) wyrd.visible = true; // This line is for
                                    //  testing

         wyrds.Add(wyrd);

         // If we've gotten to the numRows(th) row, start a new
column
         if (i%numRows == numRows-1) {
            left += (columnWidth+0.5f)*letterSize;
         }
      }
   }

}
```

2. Before pressing Play, you need to assign the PrefabLetter prefab from the Project pane to the `prefabLetter` field of the WordGame Script component of _MainCamera. After doing so, press Play, and you should see a list of words pop up on screen, as shown in Figure 34.4.

Figure 34.4. An example of the current state of the game: the level for the word TORNADO



Adding the Big Letters at the Bottom

The next step in `Layout()` is to place the large letters at the bottom of the screen.

1. Add the following code to do so:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Singleton

    [Header("Set in Inspector")]
    ...
    public bool             showAllWyrds = true;
    public float            bigLetterSize = 4f;
    public Color            bigColorDim = new Color(0.8f, 0.8f, 0.8f);
    public Color            bigColorSelected = new Color(1f, 0.9f,
0.7f);
    public Vector3          bigLetterCenter = new Vector3(0, -16, 0);

    [Header("Set Dynamically")]
    ...
    public List<Wyrd>       wyrds;
    public List<Letter>     bigLetters;
    public List<Letter>     bigLettersActive;

    ...

    void Layout() {
        ...

        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
```

```
        ...
    }

    // Place the big letters
    // Initialize the List<>s for big Letters
    bigLetters = new List<Letter>();
    bigLettersActive = new List<Letter>();

    // Create a big Letter for each letter in the target word
    for (int i=0; i<currLevel.word.Length; i++) {
        // This is similar to the process for a normal Letter
        c = currLevel.word[i];
        go = Instantiate(prefabLetter) as GameObject;
        go.transform.SetParent( bigLetterAnchor );
        lett = go.GetComponent<Letter>();
        lett.c = c;
        go.transform.localScale = Vector3.one*bigLetterSize;

        // Set the initial position of the big Letters below screen
        pos = new Vector3( 0, -100, 0 );
        lett.pos = pos;    // You'll add more code around this line
later

        col = bigColorDim;
        lett.color = col;
        lett.visible = true; // This is always true for big letters
        lett.big = true;
        bigLetters.Add(lett);
    }
    // Shuffle the big letters
    bigLetters = ShuffleLetters(bigLetters);
    // Arrange them on screen
    ArrangeBigLetters();

    // Set the mode to be in-game
    mode = GameMode.inLevel;
}
```

```
    // This method shuffles a List<Letter> randomly and returns
the result
    List<Letter> ShuffleLetters(List<Letter> letts) {
        List<Letter> newL = new List<Letter>();
        int ndx;
        while(letts.Count > 0) {
            ndx = Random.Range(0,letts.Count);
            newL.Add(letts[ndx]);
            letts.RemoveAt(ndx);
        }
        return(newL);
    }

    // This method arranges the big Letters on screen
    void ArrangeBigLetters() {
        // The halfWidth allows the big Letters to be centered
        float halfWidth = ( (float) bigLetters.Count )/2f-0.5f;
        Vector3 pos;
        for (int i=0; i<bigLetters.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            bigLetters[i].pos = pos;
        }
        // bigLettersActive
        halfWidth = ( (float) bigLettersActive.Count )/2f-0.5f;
        for (int i=0; i<bigLettersActive.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            pos.y += bigLetterSize*1.25f;
            bigLettersActive[i].pos = pos;
        }
    }

}
```

2. Now, in addition to the Letters up top, you should also see big Letters below, the shuffled form of the goal word. It's time to add some interactivity.

Adding Interactivity

For this game, we want the player to be able to type words from the available big Letters on her keyboard and press Return/Enter to submit them. She can also press Backspace/Delete to remove a letter from the end of what she has typed and press the space bar to shuffle the remaining unselected letters.

When she presses Enter, the word she typed will be compared with the possible words in the WordLevel. If the word she typed is in the WordLevel, she will get a point for each letter in the word. In addition, if the word she typed contains any smaller words that are in the WordLevel, she will also get points for those plus a multiplier for each word. Looking at the TORNADO example earlier, if a player typed TORNADO as her first word and hit return, she would get 36 total points as follows:



All of this interactivity will be handled by the `Update()` function and will be based on `Input.inputString`, a string of all the keyboard input that occurred this frame.

1. Add the following `Update()` method and supporting methods to WordGame:

```
public class WordGame : MonoBehaviour {
    ...

    [Header("Set Dynamically")]
    ...
    public List<Letter>      bigLettersActive;
    public string            testWord;
```

```
   private string          upperCase =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   ...

   void ArrangeBigLetters() { ... }

   void Update() {
      // Declare a couple useful local variables
      Letter ltr;
      char c;

      switch (mode) {
         case GameMode.inLevel:
            // Iterate through each char input by the player this
frame
            foreach (char clt in Input.inputString) {
               // Shift clt to UPPERCASE
               c = System.Char.ToUpperInvariant(clt);

               // Check to see if it's an uppercase letter
               if (upperCase.Contains(c)) { // Any uppercase letter
                  // Find an available Letter in bigLetters with this
                   //  char
                  ltr = FindNextLetterByChar(c);
                  // If a Letter was returned
                  if (ltr != null) {
                     // ... then add this char to the testWord and
                     //  move the returned big Letter to
                      //  bigLettersActive
                     testWord += c.ToString();
                     // Move it from the inactive to the active List<>
                     bigLettersActive.Add(ltr);
                     bigLetters.Remove(ltr);
                     ltr.color = bigColorSelected; // Make it look
                                                   // active
                     ArrangeBigLetters();     // Rearrange the big
```

```
                                        // Letters
                }
            }

            if (c == '\b') { // Backspace
                // Remove the last Letter in bigLettersActive
                if (bigLettersActive.Count == 0) return;
                if (testWord.Length > 1) {
                    // Clear the last char of testWord
                    testWord =
testWord.Substring(0,testWord.Length-1);
                } else {
                    testWord = "";
                }

                ltr = bigLettersActive[bigLettersActive.Count-1];
                // Move it from the active to the inactive List<>
                bigLettersActive.Remove(ltr);
                bigLetters.Add (ltr);
                ltr.color = bigColorDim;    // Make it the inactive
                                            // color
                ArrangeBigLetters();       // Rearrange the big
                                            //  Letters
            }

            if (c == '\n') { // Return/Enter
                // Test the testWord against the words in
WordLevel
                CheckWord();
            }

            if (c == ' ') { // Space
                // Shuffle the bigLetters
                bigLetters = ShuffleLetters(bigLetters);
                ArrangeBigLetters();
            }
        }
```

```
            break;
        }
    }

    // This finds an available Letter with the char c in bigLetters.
    // If there isn't one available, it returns null.
    Letter FindNextLetterByChar(char c) {
        // Search through each Letter in bigLetters
        foreach (Letter ltr in bigLetters) {
            // If one has the same char as c
            if (ltr.c == c) {
                // ...then return it
                return(ltr);
            }
        }
        return( null );  // Otherwise, return null
    }

    public void CheckWord() {
        // Test testWord against the level.subWords
        string subWord;
        bool foundTestWord = false;

        // Create a List<int> to hold the indices of other subWords
that are
        //  contained within testWord
        List<int> containedWords = new List<int>();

        // Iterate through each word in currLevel.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {

            // Check whether the Wyrd has already been found
            if (wyrds[i].found) {                           // a
                continue;
            }

            subWord = currLevel.subWords[i];
```

```
         // Check whether this subWord is the testWord or is
contained in
         //  it
         if (string.Equals(testWord, subWord)) {                // b
            HighlightWyrd(i);
            foundTestWord = true;
         } else if (testWord.Contains(subWord)) {
            containedWords.Add(i);
         }
      }

      if (foundTestWord) { // If the test word was found in
subWords
         // ...then highlight the other words contained in testWord
         int numContained = containedWords.Count;
         int ndx;
         // Highlight the words in reverse order
         for (int i=0; i<containedWords.Count; i++) {
            ndx = numContained-i-1;
            HighlightWyrd( containedWords[ndx] );
         }
      }

      // Clear the active big Letters regardless of whether
testWord
      //  was valid
      ClearBigLettersActive();
   }

   // Highlight a Wyrd
   void HighlightWyrd(int ndx) {
      // Activate the subWord
      wyrds[ndx].found = true;   // Let it know it's been found
      // Lighten its color
      wyrds[ndx].color = (wyrds[ndx].color+Color.white)/2f;
      wyrds[ndx].visible = true; // Make its 3D Text visible
   }
```

```
    // Remove all the Letters from bigLettersActive
    void ClearBigLettersActive() {
        testWord = "";            // Clear the testWord
        foreach (Letter ltr in bigLettersActive) {
            bigLetters.Add(ltr);     // Add each Letter to bigLetters
            ltr.color = bigColorDim; // Set it to the inactive color
        }
        bigLettersActive.Clear();  // Clear the List<>
        ArrangeBigLetters();       // Rearrange the Letters on screen
    }

}
```

a. If the i^{th} Wyrd on screen has already been found then continue and skip the rest of this iteration This works because the Wyrds on screen and the words in the `subWords` List are in the same order.

b. Check whether this `subWord` is the `testWord`, and if so then highlight the `subWord`. If it's not the `testWord`, check to see whether `testWord` contains this `subWord` (e.g., SAND contains AND), and if so, then add it to the `containedWords` List.

2. Save the WordGame script and return to Unity.

3. Set `showAllWyrds` to false in the Inspector for the WordGame Script component of _MainCamera. Then, press Play.

You should be presented with a working version of the game and a random level.

Adding Scoring

Because of the Scoreboard and FloatingScore code that we've already written and imported into this project, adding scoring to this game should be very easy.

1. Create a Canvas for the UI Text fields to use by choosing *GameObject* > *UI* > *Canvas* from the menu bar.

2. Drag Scoreboard from the _Prefab folder in the Project pane onto Canvas in the Hierarchy pane, making Scoreboard a child of Canvas.

3. Double check that the prefabFloatingScore field of the *Scoreboard (Script)* component of the Scoreboard GameObject is set to the PrefabFloatingScore prefab from the _Prefabs folder. (If you want to learn more about how the Scoreboard works, refer to Chapter 32, "Prototype 4: Prospector Solitaire.")

4. Create a new script named *ScoreManager* in the __Scripts folder and attach it to Scoreboard.

5. Open ScoreManager in MonoDevelop and enter the following code:

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

public class ScoreManager : MonoBehaviour {
   static private ScoreManager S;  // Another private Singleton

   [Header("Set in Inspector")]
   public List<float>     scoreFontSizes = new List<float> { 36,
64, 64, 1 };
   public Vector3         scoreMidPoint = new Vector3(1,1,0);
   public float           scoreTravelTime = 3f;
   public float           scoreComboDelay = 0.5f;

   private RectTransform   rectTrans;

   void Awake() {
      S = this;
```

```
        rectTrans = GetComponent<RectTransform>();
    }

    // This method allows ScoreManager.SCORE() to be called
from anywhere
    static public void SCORE(Wyrd wyrd, int combo) {
        S.Score(wyrd, combo);
    }

    // Add to the score for this word
    // int combo is the number of this word in a combo
    void Score(Wyrd wyrd, int combo) {
        // Create a List<> of Bezier points for the FloatingScore
        List<Vector2> pts = new List<Vector2>();

        // Get the position of the first Letter in the wyrd
        Vector3 pt = wyrd.letters[0].transform.position;          // a
        pt = Camera.main.WorldToViewportPoint(pt);

        pts.Add(pt); // Make pt the first Bezier point              // b

        // Add a second Bezier point
        pts.Add( scoreMidPoint );

        // Make the Scoreboard the last Bezier point
        pts.Add(rectTrans.anchorMax);

        // Set the value of the Floating Score
        int value = wyrd.letters.Count * combo;
        FloatingScore fs =
Scoreboard.S.CreateFloatingScore(value, pts);

        fs.timeDuration = scoreTravelTime;
        fs.timeStart = Time.time + combo * scoreComboDelay;
        fs.fontSizes = scoreFontSizes;

        // Double the InOut Easing effect
```

```
        fs.easingCuve = Easing.InOut+Easing.InOut;

        // Make the text of the FloatingScore something like "3 x 2"
        string txt = wyrd.letters.Count.ToString();
        if (combo > 1) {
            txt += " x "+combo;
        }
        fs.GetComponent<Text>().text = txt;
    }

}
```

a. We want the starting position of the FloatingScore to be directly over the wyrd. First, we get the 3D, world coordinates location of the 0^{th} letter of the wyrd. On the next line, we use the _MainCamera to convert it from 3D world coordinates to a ViewportPoint. ViewportPoints range from 0 to 1 in X and Y coordinates and indicate where the point is relative to the width and height of the screen and are used for UI coordinates.

b. When the Vector3 pt is added to the List<Vector2> pts, the Z coordinate is dropped.

6. Save the ScoreManager script, and then add scoring code to the CheckWord() method of the WordGame C# script by making the following bolded edits:

```
public class WordGame : MonoBehaviour {
    ...

    public void CheckWord() {
        ...
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            if (string.Equals(testWord, subWord)) {
                // ...then highlight the subWord
                HighlightWyrd(i);
```

```
            ScoreManager.SCORE( wyrds[i], 1 ); // Score the
testWord // a
            foundTestWord = true;
        } else if (testWord.Contains(subWord)) {
            ...
        }
    }

    // If the test word was found in subWords
    if (foundTestWord) {
        ...
        for (int i=0; i<containedWords.Count; i++) {
            ndx = numContained-i-1;
            HighlightWyrd( containedWords[ndx] );
            ScoreManager.SCORE( wyrds[ containedWords[ndx] ],
i+2 );// b
        }
    }
    ...
  }

  ...
}
```

a. This line calls the `ScoreManager.SCORE()` static method to score the `testWord` that the player spelled.

b. Here, `ScoreManager.SCORE()` is called to score any smaller words contained within the `testWord`. The second parameter (`i+2`) is the number of this word in the combo.

7. Save the WordGame script, return to Unity, and press Play.

You should now get a score for each correct word you enter, and you get a multiplier for each additional valid word contained in the word you type. However, the white scores are a bit difficult to see over the white Letter tiles. We'll fix this a little later when we add more color to the game.

Adding Animation to Letters

In a similar manner to scoring, we can very easily add smooth animation of Letters by taking advantage of the interpolation functions that we imported in the Utils script.

1. Add the following code to the Letter C# script:

```
public class Letter : MonoBehaviour {
    [Header("Set in Inspector")]
    public float          timeDuration = 0.5f;
    public string         easingCuve = Easing.InOut;// Easing from
Utils.cs

    [Header("Set Dynamically")]
    public TextMesh       tMesh; // The TextMesh shows the char
    public Renderer       tRend; // The Renderer of 3D Text. This will
                          //   determine whether the char is visible
    public bool           big = false; // Big letters act a little differently
    // Linear interpolation fields
    public List<Vector3>    pts = null;
    public float          timeStart = -1;

    private char          _c;    // The char shown on this Letter
    ...

    // Sets the position of the Letter's gameObject
    // Now sets up a Bezier curve to move to the new position
    public Vector3 pos {
        set {
            // transform.position = value; // This line is now
commented out

            // Find a midpoint that is a random distance from the
actual
            //  midpoint between the current position and the value
passed in
```

```
            Vector3 mid = (transform.position + value)/2f;
            // The random distance will be within 1/4 of the
magnitude of the
            //  line from the actual midpoint
            float mag = (transform.position - value).magnitude;
            mid += Random.insideUnitSphere * mag*0.25f;
            // Create a List<Vector3> of Bezier points
            pts = new List<Vector3>() { transform.position, mid,
value };
            // If timeStart is at the default -1, then set it
            if (timeStart == -1 ) timeStart = Time.time;
        }
    }

    // Moves immediately to the new position
    public Vector3 posImmediate {                              // a
        set {
            transform.position = value;
        }
    }

    // Interpolation code
    void Update() {
        if (timeStart == -1) return;

        // Standard linear interpolation code
        float u = (Time.time-timeStart)/timeDuration;
        u = Mathf.Clamp01(u);
        float u1 = Easing.Ease(u,easingCuve);
        Vector3 v = Utils.Bezier(u1, pts);
        transform.position = v;

        // If the interpolation is done, set timeStart back to -1
        if (u == 1) timeStart = -1;
    }

}
```

a. Since setting `pos` now creates an interpolation to the new position, `posImmediate` was added to allow us to jump this Letter immediately to another position.

2. Save the Letter script, return to Unity, and press Play. You'll now see the Letters all interpolate to their new positions. However, it looks a little strange for all the Letters to move at the same time and start from the center of the screen.

3. Let's add some small changes to the `WordGame.Layout()` method to improve this:

```
public class WordGame : MonoBehaviour {
    ...

    void Layout() {
        ...
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                ...
                // The % here makes multiple columns line up
                pos.y -= (i%numRows)*letterSize;

                // Move the lett immediately to a position above the
screen
                lett.posImmediate = pos+Vector3.up*
(20+i%numRows);
                // Then set the pos for it to interpolate to
                lett.pos = pos; // You'll add more code around this line later
                // Increment lett.timeStart to move wyrds at different
times
                lett.timeStart = Time.time + i*0.05f;

                go.transform.localScale = Vector3.one*letterSize;
                wyrd.Add(lett);
```

```
            }
            ...
        }
        ...
        // Create a big Letter for each letter in the target word
        for (int i=0; i<currLevel.word.Length; i++) {
            ...
            // Set the initial position of the big Letters below screen
            pos = new Vector3( 0, -100, 0 );

            lett.posImmediate = pos;
            lett.pos = pos;     // You'll add more code around this line later
            // Increment lett.timeStart to have big Letters come in
last
            lett.timeStart = Time.time +
currLevel.subWords.Count*0.05f;
            lett.easingCuve = Easing.Sin+"-0.18"; // Bouncy easing

            col = bigColorDim;
            ...
        }
        ...
    }

    ...
}
```

4. Save the WordGame script, return to Unity, and press Play. The game should now layout with nice smooth, progressive motions.

Adding Color

Now that the game moves well, it's time to add a little color:

1. Add the following bold code to WordGame to color the wyrds based on their length:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Singleton

    [Header("Set in Inspector")]
    ...
    public Vector3          bigLetterCenter = new Vector3(0, -16, 0);
    public Color[]          wyrdPalette;

    [Header("Set Dynamically")]
    ...

    void Layout() {
        ...
        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                ...
            }

            if (showAllWyrds) wyrd.visible = true; // This line is for testing

            // Color the wyrd based on length
            wyrd.color = wyrdPalette[word.Length-
WordList.WORD_LENGTH_MIN];

            wyrds.Add(wyrd);

            ...
        }
        ...
    }

    ...
}
```

These last few code changes have been so simple because we already had supporting code in place (for example, the Wyrd.color and Letter.color properties as well as the Easing code in the Utils class).

Now, you need to set about eight colors for wyrdPalette. To do this, we'll use the Color Palette image included in the import at the beginning of the project. We're going to be using the eye dropper to set color, which may leave you wondering how to see both the Color Palette image and the _MainCamera Inspector at the same time. To do that, we'll take advantage of Unity's capability to have more than one Inspector window open at the same time.

Figure 34.5. Using the pane options button to add an Inspector to the Game pane



2. As shown in Figure 34.5, click the pane options button (circled in red) and choose *Add Tab* > *Inspector* to add an Inspector to the Game tab.

Figure 34.6. The lock icon on one Inspector (circled in red) and the eye dropper in the other inspector (circled in light blue)



3. Select the *Color Palette* image in the *Materials & Textures* folder of the Project pane. It will appear in both Inspectors. (You might need to drag the edge of the image preview part of the Inspector to make it look like the Figure 34.6.)

4. Click the lock icon on one inspector (circled in red in the Figure 34.6)

5. Select _MainCamera in the Hierarchy pane. You'll see that only the unlocked Inspector changes to _MainCamera, while the locked one is still showing the Color Palette image.

6. Expand the disclosure triangle next to `wyrdPalette` in the _MainCamera Inspector and set its size to 8.

7. Click the eye dropper next to each `wyrdPalette` element (circled in light blue), and then click one of the colors in the Color Palette image. Doing this will give you the eight different colors of the Color Palette image, but they will all default to having an alpha of 0 (and therefore being invisible).

8. Click each color bar in the `wyrdPalette` array and set each one's alpha (or A) to 255 to make it fully opaque.

Now when you play the scene, you should see something that looks like the screen shot from the beginning of the chapter.

9. And, as always, save your scene.

Summary

In this chapter, you created a simple word game and added a little flair to it with some nice interpolated movement. If you've been following these tutorials in order, you may have realized that the process of making them is getting a little bit easier. With the expanded understanding of Unity that you now have and the capabilities of readymade utility scripts like Scoreboard, FloatingScore, and Utils, you're able to focus more of the coding effort on the things that are new and different in each game and less on reinventing the wheel.

Next Steps

In the previous prototypes, you saw examples of how to set up a series of game states to handle the different phases of the game and transition from one level to the next. Right now, this prototype doesn't have any of that. On your own, you should add that kind of control structure to this game.

Here are some things to think about as you do so:

• When should the player be able to move on to the next level? Must she guess every single word, or can she move on once she has either reached a specific point total or has guessed the target word.

• How will you handle levels? Will you just pick a completely random word as we are now, or will you modify the randomness to make sure that level 5 is always the same word (therefore making it fair for players to compare their scores on level 5)? Here's a hint if you decide to try for a modified randomness:

```
using UnityEngine;
using System.Collections;

public class LevelPicker : MonoBehaviour {
   static private System.Random   rng;

   [Header("Set in Inspector")]
   public int      randomSeed = 12345;

   void Awake() {
      rng = new System.Random(randomSeed);
   }

   static public int Next(int max=-1) {
      // Returns the next number from rng between 0 and max-1.
      // If -1 is passed in, max is ignored
      if (max == -1) {
         return rng.Next();
      } else {
         return rng.Next( max );
      }
   }

}
```

• How do you want to handle levels with too many or too few subWords? Some collections of seven letters have so many words that they extend off the screen to the right, whereas others have so few that there's only one column. Do you want to make the game ask for the next word in this case? If so, how do you then instruct something like the PickNthRandom function to skip certain numbers?

You should have enough knowledge of programming and prototyping now that you can take these questions and make this into a real game. You've got the skills, now go for it!

Appendix A. Standard Project Setup Procedure

Many times throughout the book, you are asked to create a new project and then given code to try. This is the standard procedure that you should follow each time to create a new project, set up a scene, create a new C# script, and attach that script to the Main Camera of the scene. Instead of repeating these instructions throughout this book, they are collected here.

Setting Up a New Project

Follow these steps to set up a new project. The screenshots show the procedure on both OS X and Windows:

1. When you first launch Unity, you are presented with the start window shown in Figure A.1. Here, you can click the New button to create a new project. Alternatively, if you are already running Unity, you can choose *File* > *New Project...* from the menu bar.

Figure A.1. Choosing New Project from the Unity start window



2. This opens the Unity new project screen shown in Figure A.2. Unity will create a new project folder with the name you set in the *Project name** field at the location you set in the *Location** field. Clicking the ellipses on the right side of the Location* field will allow you to choose a location for your project using your standard system file dialog box. In general, for this book, you should choose the 3D radio button and set Enable Unity Analytics to Off. For more information on the options, check out the New Project Options sidebar.

Figure A.2. The New Project Screen



As an example, with the settings in Figure A.2, Unity would create a project folder named *ProtoTools Project* on the Desktop of my Mac that was defaulted to a 3D layout.

New Project Options

Unity gives you several options on the new project screen.

3D / 2D (Choose 3D) – The 3D / 2D radio button sets up a default camera in your project that is either perspective (3D) or orthographic (2D). That's it.

Enable Unity Analytics (Choose Off) – Unity Analytics is a way to get information on how many people are playing your game, what they're doing, etc. It's a fantastic tool, but it's not needed for this book.

Add Asset Package (Don't) – Unity comes with a number of Asset Packages that provide various things like terrain tools, particle effects, etc. Many of them are pretty cool, but for this book, there's no reason to add them to your project during project creation. I generally avoid adding them for three reasons:

• **Project bloat:** If you import every possible package, the size of the project will bloat to 1,000 times its original size (from ≈300Kb to ≈300MB)!

• **Project pane clutter:** Importing all the packages will also add a huge number of items and folders to your Assets folder and Project pane.

• **You can always import them later:** At any time in the future, you can choose *Assets* > *Import Package* from the menu bar to import

any of the packages listed in the Project Wizard.

---

3. On the new project screen, click the *Create Project* button (shown in Figure A.2). Unity will appear to close and relaunch, presenting you with the blank canvas of your new project. This relaunch may take a few seconds, so be patient.

Getting the Scene Ready for Coding

The new project you just created comes with a default scene. To get ready for coding, follow these instructions:

1. **Save the scene.** The first thing you do in a project should *always* be to save the scene. Choose *File* > *Save Scene As...* from the menu bar and choose a name. (Unity will automatically navigate to the correct folder in which to save the scene). I tend to choose something like *_Scene_0*, which is easily enumerable as I create more scenes in the future. The underscore at the beginning of the name will sort the scene to the top of the Project pane.

2. **Create a new C# script (optional).** Some chapters ask you to create one or more C# scripts before beginning the project. To do so, click the *Create* button in the Project pane and choose *Create* > *C# Script*. A new script will be added to the Project pane, and its name will be highlighted for you to change. Name this script whatever you like as long as there are no spaces or special characters in the name, and then press the Return or Enter key to save the name. In Figure A.3, the script is named *HelloWorld*.

Figure A.3. Creating a new C# script and viewing that script in MonoDevelop



Warning

CHANGING A SCRIPT NAME AFTER IT HAS BEEN CREATED CAN CAUSE PROBLEMS When you set the name of a script as part of the creation process, Unity will automatically change the name in the class declaration as well (on line 4 in Figure A.3). However, if you choose to change the name of your C# script after that initial process, you need to change its name not only in the Project pane but also in the class declaration line of the script itself. In Figure A.3, this class declaration is on line 4, where *HelloWorld* would need to be changed to the new script name.

---

3. **Attach the C# script to the scene's Main Camera (optional).** Some chapters request that you *attach* one or more of the new scripts to the Main Camera. Attaching a script to a GameObject like Main Camera will make that script a *component* of the GameObject. All scenes will start with a Main Camera already included, so that's a fantastic place to attach any basic script that you want to run. Generally, if a C# script is not attached to a GameObject in the scene, it will not run.

4. Attach the script to the Main Camera (optional). Some projects will ask you to attach the script you just created to the main camera. This is a bit tricky, but you'll soon be used to it because it is so frequently done in Unity. Click the name of the new script, drag it over on top of the Main Camera in the scene, and release the mouse button. It should look like what is shown in Figure A.4.

Figure A.4. Dragging the C# script onto the Main Camera in the Hierarchy pane



The C# script is now attached to the Main Camera and will appear in the Inspector if the Main Camera is selected. You're now all set to start work on any of the projects in the book.

Appendix C. Online Reference

Whereas many online references would just give you a list of websites to check out, I thought it would be more useful for me to use this appendix to show you how I tend to look for answers online when I need them. Appropriately, this includes a few basic links, but it also includes strategies for tracking down information and answers to problems that you might encounter.

I recommend reading this straight through once (it's very short) and then returning to it when you encounter an issue.

Tutorials

Unity has created a series of tutorials over the years that can be very useful to check out. This book focused on short gameplay tutorials to help you understand how to program game mechanics, whereas the tutorials created by Unity tend to spend an equal amount of time on art assets, animation, building scenes, and visual effects in addition to scripting. This book is about you learning how to design and prototype games; their tutorials are about learning all the different features of the Unity engine.

Be aware when looking at these that many were made with older versions of Unity, and they sometimes don't update the tutorials to match the new version of the engine (meaning that sometimes the elements of the Unity interface that the tutorial describes or some code libraries that they use may have changed). In addition, some of the very old tutorials were written in JavaScript rather than C#. This should be fine for you—especially once you've read and understood the code in this book—but in general, I'd recommend looking for tutorials written in C# to make sure you're finding more recent ones.

Unity Website

On Unity's website, there is a *Learn* section that is meant to introduce you to Unity through several different tutorials. This link will take you to that page. Choose a topic that you would like to learn about, and you can view a video tutorial to help you do so:

• **Learn section—Tutorials:** http://unity3d.com/learn/tutorials

Unite Archive

Once you have some experience with Unity under your belt, you may want to look at some more advanced resources. A great place to do this is to look at videos of various talks from the Unite conference over the years. Unity holds versions of Unite all over the world every year, and they record many of the talks. You can find them all here:

• https://unite.unity.com/archive

Programming

As you delve further into programming Unity, you'll find that the documentation for programming Unity with C# is primarily located in two places: Unity's scripting documentation and the Microsoft C# reference. The Unity scripting documentation does a fantastic job of documenting Unity-specific features, classes, and components, but it doesn't cover any of the core C# classes (such as List<>, Dictionary<>, and so on). For these, turn to Microsoft's C# documentation. I recommend first looking for something in the Unity documentation available on your computer, and if it's not there, then look in the Microsoft docs.

Unity Scripting Reference

Unity scripting references include the following:

• **Online:** http://docs.unity3d.com/Documentation/ScriptReference/

• **Local:** From within Unity, choose *Help* > *Scripting Reference* from the menu bar. This brings up a version of the reference that is stored locally on your computer. Even if you don't have an Internet connection, this reference is available. (I use it while traveling all the time.)

Microsoft C# Reference

Search Bing.com for *Microsoft C# Reference.* The first hit should be what you're looking for. As of the time of writing this book, the URL is http://msdn.microsoft.com/en-us/library/618ayhy6.aspx.

Stack Overflow

Stack Overflow is an online community of developers helping developers. People post questions, and other members of the site answer them. In a bit of gamification, those who give the best answers (as voted by other members) earn experience points and prestige on the site:

• http://stackoverflow.com

Often, when I'm trying to figure out how to do something new or unusual, I'll end up finding a good answer on Stack Overflow. For instance, if I want to know how to sort a List<> using LINQ, I enter *c# LINQ sort list of objects* into Google, and as I write this, the top eight hits are Stack Overflow questions. I usually find myself there via a Google search rather than starting on Stackoverflow.com, but when a hit comes up on the site, it's my first choice for finding good answers.

Learning More C#

I highly recommend two additional books for learning more about C#:

• **For beginners:** Rob Miles's *C# Programming Yellow Book*, http://www.csharpcourse.com

Rob Miles, a lecturer at the University of Hull, has written a fantastic book on C# programming that he updates often. You can find the current version on his website. It is witty, clear, and comprehensive.

• **For reference:** *C# 4.0 Pocket Reference, 3rd Edition,* http://shop.oreilly.com/product/0636920013365.do

Although there is now a C# 5.0 version of this reference, Unity is still using the C# 4.0 standard (well, it's most of C# 4.0; there are actually a few bits missing), so this is the reference for you. Any time I have a C# question, this is the first place I turn. It's a truncated version of the information in O'Reily's *C# in a Nutshell* book, but I actually find the pocket reference more useful. It also includes a ton of LINQ information.

Searching Tips

Any time you want to search for something having to do with C#, make *C#* the first term in your search. If you just search for *list,* the first thing to come up has nothing to do with coding. Searching for *C# list* will get you to the right place immediately.

Similarly, if you want to find anything related to Unity, be sure to make *Unity* your first search term.

Finding Assets

The following sections provide advice on finding various art and audio assets.

The Unity Asset Store

The Asset Store is accessed by opening the Asset Store window in Unity (choose *Window > Asset Store* from the menu bar) or by going to the following website. The Asset Store has a huge collection of models, animations, sounds, code, and even complete Unity projects

that you can download. Most of the assets are available for a small fee, but some of the assets on the site are even free. Some things are very pricey, but they are often worth it and can save you hundreds of hours of development:

• https://www.assetstore.unity3d.com/

Models and Animations

These sites are some places to look for 3D models. Some will be free, but many will be paid. Also, be aware that many of the free ones are for noncommercial use only:

• **TurboSquid:** http://www.turbosquid.com/

• **Google 3D Warehouse:** http://3dwarehouse.sketchup.com/

Be aware that nearly all the assets on the Google 3D Warehouse site are in SketchUp or Collada formats, neither of which import into Unity very well. You can watch this Unite 2013 talk for information on how to make the imports work better:

http://www.youtube.com/watch?v=Zj727ov9Pe0.

Fonts

Nearly all the fonts on these sites are free for noncommercial use, but you will often need to pay to use them on commercial projects:

• http://www.1001fonts.com/

• http://www.1001freefonts.com/

• http://www.dafont.com/

• http://www.fontsquirrel.com/

• http://www.fontspace.com/

Other Tools and Educational Discounts

If you are a student or faculty member of a university, you qualify for many discounts on software.

• **Adobe:** Adobe offers the entire Creative Cloud suite of their tools to students at a discounted monthly rate. This includes Photoshop, Illustrator, Premier, and others.

http://www.adobe.com/creativecloud/buy/students.html

• **AutoDesk:** AutoDesk gives students and educators a free 36-month license for almost any of their tools, including 3ds Max, Maya, Motionbuilder, Mudbox, and more.

http://www.autodesk.com/education/free-software/featured

• **Blender:** Blender is a free, open source tool for modeling and animation. It includes many of the capabilities of software like Maya and 3ds Max but is entirely free and can be used for commercial purposes. However, its interface is quite different from what you may be used to from other modeling and animation software.

http://www.blender.org/

• **SketchUp:** SketchUp is another free tool for modeling. It has very intuitive modeling controls and is updated frequently. The basic version, *SketchUp Make*, is completely free, and *SketchUp Pro* is heavily discounted for students and educators. In newer versions of SketchUp, it's possible to export obj and fbx format files, which Unity can import easily. If you're making your own models in SketchUp, consider checking the *Triangulate all faces*, *Export two-sided faces*, and *Swap YZ coordinates* options and setting your Units to *Meters* when exporting files.

http://www.sketchup.com/