

Introduction to **GAME SYSTEMS DESIGN**



Dax **GAZAWAY**

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Introduction to Game Systems Design

Introduction to Game Systems Design

Dax Gazaway

 Addison-Wesley

**Boston • Columbus • New York • San Francisco • Amsterdam • Cape
Town**
**Dubai • London • Madrid • Milan • Munich • Paris • Montreal •
Toronto • Delhi • Mexico City**
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

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

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

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services. The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screenshots may be viewed in full within the software version specified.

Microsoft® Windows and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Cover image: Vladimir Vihrev/Shutterstock

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover

designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021940285

Copyright © 2022 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-744084-9

ISBN-10: 0-13-744084-7

ScoutAutomatedPrintCode

Editor-in-Chief

Mark Taub

Acquisitions Editor

Malobika Chakraborty

Development Editor

Chris Zahn

Managing Editor

Sandra Schroeder

Senior Project Editor

Lori Lyons

Copy Editor

Kitty Wilson

Production Manager

Vaishnavi Venkatesan/codeMantra

Indexer

Timothy Wright

Proofreader

Betty Pessagno

Compositor

codeMantra

This book is dedicated to my game family. This includes those who raised me as a gamer, those who have been with me through this journey, and those who took me under their wing as I learned the professional trade.

Thank you to everyone.

Contents at a Glance

- Preface
- Acknowledgments
- About the Author
- 1 Games and Players: Defined**
- 2 Roles in the Game Industry**
- 3 Asking Questions**
- 4 System Design Tools**
- 5 Spreadsheet Basics**
- 6 Spreadsheet Functions**
- 7 Distilling Life into Systems**
- 8 Coming Up with Ideas**
- 9 Attributes: Creating and Quantifying Life**
- 10 Organizing Data in Spreadsheets**
- 11 Attribute Numbers**
- 12 System Design Foundations**
- 13 Range Balancing, Data Fulcrums, and Hierarchical Design**
- 14 Exponential Growth and Diminishing Returns**
- 15 Analyzing Game Data**
- 16 Macrosystems and Player Engagement**
- 17 Fine-Tuning Balance, Testing, and Problem Solving**

18 Systems Communication and Psychology

19 Probability

20 Next Steps

Index

Contents

Preface

Acknowledgments

About the Author

1 Games and Players: Defined

Defining *Game*

Agreed Upon, Artificial Rules

Players Have an Impact on the Outcome

People Can Opt Out

Game Sessions Are Finite

Intrinsic Rewards

Game Attributes Summary

Finding the Target Audience for a Game: Player Attributes

Age

Gender

Tolerance for Learning Rules

Interest in Challenge

Desired Time Investment

Pace Preference

Competitiveness

Platform Preference

Skill Level

Genre/Art/Setting/Narrative Preference

Value Gained from Players

Payment

Other Forms of Value

Target Audience Value

Target Audience Composite

Chess

Galaga

Mario Kart

The Battle for Wesnoth

Bejeweled

What to Do with a Target Audience Profile

Further Steps

2 Roles in the Game Industry

Core Management Team

Vision Holder

Lead Engineer

Lead Artist

Lead Designer

Producer

Lead Sound Designer

Team Subdisciplines

Art

Engineering

Production

Design

Sound Team

QA Team

Narrative Designer

Additional Roles

Further Steps

3 Asking Questions

How to Ask a Theoretical Question
Steps of the Scientific Method
Defining a Question for Data Analysis

How to Ask for Help with a Problem
Why How You Ask Matters
Steps to Writing a Good Question

Further Steps

4 System Design Tools

What Is Data?
Game Industry Tools

Documentation Tools
Image Editing Tools
3D Modeling Tools
Flowchart Tools
Databases
Bug-Tracking Software
Game Engines

Further Steps

5 Spreadsheet Basics

Why Spreadsheets?
What Is a Spreadsheet?
Spreadsheet Cells: The Building Blocks of Data

Cells
The Formula Bar
Spreadsheet Symbols

Data Containers in Spreadsheets
Columns and Rows
Sheets

Workbooks

Spreadsheet Operations

Referencing a Separate Sheet

Hiding Data

Freezing Part of a Sheet

Using Comments and Notes

Using Formfill

Using Filters

Data Validation

The Data Validation Dialog

Time Validation

List Validation

Named Ranges

Further Steps

6 Spreadsheet Functions

Grouping Arguments

Function Structure

More Complex Functions

Functions for System Designers

SUM

AVERAGE

MEDIAN

MODE

MAX and MIN

RANK

COUNT, COUNTA, and COUNTUNIQUE

LEN

IF

COUNTIF

VLOOKUP

FIND

MID

NOW

RAND

ROUND

RANDBETWEEN

Learning About More Functions

How to Choose the Right Function

Further Steps

7 Distilling Life into Systems

An Abstract Example

Throwing

Sticks

Running

Teamwork

Putting Together the Mechanics

Story in Games

Further Steps

8 Coming Up with Ideas

Idea Buffet

Sample Idea Buffet

Running a Brainstorming Session

Having Goals

Gathering the Troops

Giving Yourself a Block of Time

Don't Accept the First Answer

- Avoiding Criticism
- Keeping on Topic (Kind Of)
- Capturing the Creativity
- Keeping Expectations Reasonable
- Percolating

Methods to Force Creativity

- Bad Storming
- Jokes
- Building Blocks
- Future Past
- Iterative Stepping
- Halfway Between
- Opposite Of
- Random Connections
- Stream of Consciousness Writing

Further Steps

9 Attributes: Creating and Quantifying Life

Mechanics Versus Attributes

Listing Attributes

- Initial Brainstorming
- Blue-Sky Brainstorming
- Researching Attributes
- Referring to Your Own Personal Attribute Bank

Defining an Attribute

- Considerations When Defining an Attribute

Grouping Attributes

Further Steps

10 Organizing Data in Spreadsheets

Create a Spreadsheet to Be Read by an Outsider

Avoid Typing Numbers

Label Data

Validate Your Data

Use Columns for Attributes and Rows for Objects

Color Coding

Avoid Adding Unneeded Columns or Rows or Blank Cells

Separate Data Objects with Sheets

 Reference Sheet

 Introduction Sheet

 Output/Visualization Sheets

 Scratch Sheet

Spreadsheet Example

Further Steps

11 Attribute Numbers

Getting a Feel for Your Attributes

Determining the Granularity for Numbers

 Numbers Should Relate to Probability

 Some Numbers Need to Relate to Real-World Measurements

 User Smaller Numbers for Easier Calculations

 Use Larger Numbers for More Granularity

 Very Large Numbers Are Confusing

 Humans Hate Decimals and Fractions, but Computers Don't Mind Them

 Numbering Example

The Tension Trick

Searching for the Right Numbers

Further Steps

12 System Design Foundations

Attribute Weights
DPS and Intertwined Attributes
Binary Searching
 How Binary Searching Works
 Lacking a Viable Range
Naming Conventions
Naming Object Iterations
 The Problem with “New”
 Iteration Naming Method 1: Version Number
 Iteration Naming Method 2: Version Letter and Number
 Special Case Terms
Using the Handshake Formula
Further Steps

13 Range Balancing, Data Fulcrums, and Hierarchical Design

Range Balancing
 How Range Balancing Works
 Who Adjusts What
Data Fulcrums
 What Is a Fulcrum?
 Creating a Fulcrum
 Testing a Fulcrum
 Locking a Fulcrum
 Using a Fulcrum for Data Creation
 Unavoidable Cross-testing
 Fulcrum Progression
Hierarchical Design
 Starting the Hierarchy
 Advantages of Hierarchical Design
Further Steps

14 Exponential Growth and Diminishing Returns

Linear Growth

Exponential Growth

Parts of the Basic Exponential Growth Formula

Building Blocks of the Exponential Growth Formula

Tweaking the Basic Exponential Growth Formula

A Note on Iterations

Exponential Charts and Game Hierarchy

Further Steps

15 Analyzing Game Data

Overview Analysis

Next-Level Deep Analysis

Practicing Data Analysis

Comparison Analysis

Canaries

Further Steps

16 Macrosystems and Player Engagement

Macrosystem Difficulty Adjustment

Flat Balancing

Positive Feedback Loops

Negative Feedback Loop

Dynamic Difficulty Adjustment

Layered Difficulty Adjustment

Cross-Feeding

Balancing Combinations

Further Steps

17 Fine-Tuning Balance, Testing, and Problem Solving

Balance

- Why Balance Matters
- General Game Balance
- Breaking Your Data
- Problems with Balancing Judged Contests
- How to Start Balancing Data
- Performing Playtests
 - Minimum Viability Testing
 - Balance Testing
 - Bug Testing
 - User Testing
 - Beta/Postlaunch Telemetry Testing
- Solving Problems
 - Identify the Problem
 - Eliminate Variables
 - Come Up with Solutions
 - Communicate with the Team
 - Prototype and Test
 - Document the Changes
- Further Steps

18 Systems Communication and Psychology

- Games as Conversations
- Word Meanings
- Noise
- Reciprocity
 - Overstepping Bounds
 - Shallow Relationship
 - Right Balance
- Reward Expectations
- Further Steps

19 Probability

Basic Probability

Probability Notation

Calculating One-Dimensional Even-Distribution Probability

Calculating One-Dimensional Uneven-Distribution Probability

Calculating Compound Probability

Calculating 2D6 “Or Higher” Cumulative Probability

Calculating the Probability of Doubles

Calculating a Series of Single Events

Calculating More Than Two Dimensions

Calculating Dependent Event Probability

Calculating Mutually Exclusive Event Probability

Calculating Enumerated Probability with an Even Distribution

Calculating Enumerated Probability with an Uneven Distribution

Calculating Attributes Weights Based on Probability

Calculating Imperfect Information Probability

Perception of Probability

Probability Uncertainty

Mapping Probability

Attributes of a Random Event

Mapping Probability Examples

Measuring Luck in a Game

Testing for Pure Luck

Testing for Luck Dominant

Testing for Luck Influenced

Adjusting the Influence of Luck

Chaos Factor

Further Steps

20 Next Steps

Practice

Analyze Existing Games

Play New Games

Modify Existing Games

Work on Your Game

Keep Learning

Index

Preface

This book covers the basic aspects of game system design in plain English. It uses numerous examples and analogies to help guide you through topics that might seem intimidating at first but are totally within your reach. The book focuses on learning how to use spreadsheets for system design. It covers the basics and best practices for using spreadsheets to make complex game data more manageable.

Who This Book Is For

The primary audience for this book is aspiring game designers who are new to doing system design and interested in learning more. It is assumed that anyone starting this book already understands basic mathematics. But, beyond that, there are no presumptions for prior game design learning. This book is made to guide someone with a basic high school education from being a complete novice to becoming a practicing system designer.

The following are some of the groups of people who could benefit from the methods described in this book:

- Aspiring professional video game system designers
- Game masters/dungeon masters
- Hobbyist video game designers
- Designers of pen-and-paper RPGs and other analog games
- Experienced level designers who want more system design knowledge
- Programmers/engineers who will be working with system designers
- High school educators who want to connect games with math for students
- Producers/lead designers who want to better understand systems

How To Use This Book

This book is written to be read from beginning to end if you are starting fresh, without much prior knowledge of game systems. It's also made to be a reference book that you can jump around in and pick up useful bits of information, even if you are an experienced system designer. The best method for absorbing the information would be to read through the book once, working in a spreadsheet as you go, and then come back to the book as you create your next game for guidance on the complex tasks required to fully realize your game.

This book discusses and refers to a number of existing games, and it would be helpful for you to understand these games to some extent. Before you read the rest of this book, familiarize yourself with the following games by at least watching video reviews online or finding free web apps and playing the game a few times:

- Play backgammon, chess, and the Royal Game of Ur. Pay attention to the kinds of dice rolls you make in these games, how pieces are moved, and how the mechanics of each game interact with the game objects.
- Play The Battle for Wesnoth to get a better idea of what a turn-based game is and what an RPG is. Wesnoth has attribute-driven data objects and game mechanics that illustrate many of the concepts covered in this book. Further, it is supported by an active community that keeps the game well documented and up to date.
- Play or at least watch video reviews of *Pac Man*, *Galaga*, and other classic arcade games.

The games used as examples in this book were purposefully chosen because they are easily accessible.

This book describes many methods of working with game systems in great detail. It might seem that the methods in this book are being exclusively recommended, but this is not the case. Game system designers use an infinite number of methods, tricks, and techniques to do their work. They use so many, in fact, that they could not fit into a single book. This book is designed to provide a starting point that shows a small number of sample

methods that are useful for all system designers. I expect and encourage you to continue to learn more techniques from other books, colleagues, and your own personal experiences. There are as many different ways to design game systems as there are system designers, and experimenting will help you find your own style.

What This Book Covers

Here is a rundown of what each chapter in this book covers.

- **Chapter 1: Games and Players: Defined**

This chapter defines some of the important terms used in this book and provides some clarity on some important topics.

- **Chapter 2: Roles in the Game Industry**

The game industry includes a wide variety of disciplines and subdisciplines that can be confusing to those who are new to game design. This chapter describes the common roles in the industry.

- **Chapter 3: Asking Questions**

Game designers must ask questions and interpret answers in unique ways, and this chapter helps you rethink how we go about it.

- **Chapter 4: System Design Tools**

The game industry is, as you would expect, full of computer software tools. This chapter covers the kinds of tools you are likely to use and some of the most popular tools in each category.

- **Chapter 5: Spreadsheet Basics**

Spreadsheets are ubiquitous in most work, and they are especially useful to game system designers. This chapter covers spreadsheet basics.

- **Chapter 6: Spreadsheet Functions**

This chapter continues the exploration of the power of spreadsheets by focusing on functions.

- **Chapter 7: Distilling Life into Systems**

When you really look in detail at the mechanics that compose any game, you find that they are analogs for aspects of real life, even if they are abstracted. This chapter explains how you use those abstractions to create the building blocks of games.

- **Chapter 8: Coming Up with Ideas**

This chapter helps you develop your skills around being creative, specifically in regard to coming up with new ideas for games.

- **Chapter 9: Attributes: Creating and Quantifying Life**

One of the most common early tasks system designers perform is creating attributes for game objects. This chapter covers what attributes are and how to get started creating them for a game.

- **Chapter 10: Organizing Data in Spreadsheets**

Once you have started creating attributes for your game objects, you will need to organize them and eventually analyze them. The best place to do this is in a spreadsheet. This chapter covers how to organize your ideas in a usable format.

- **Chapter 11: Attribute Numbers**

This chapter discusses how to quantify attributes into numbers, including a scale of numbers xxiiiand what kind of number granularity best fits a game.

- **Chapter 12: System Design Foundations**

This chapter covers attribute weights, considerations for intertwined attributes, binary searching for the correct number, and naming conventions.

- **Chapter 13: Range Balancing, Data Fulcrums, and Hierarchical Design**

This chapter discusses methods of turning a small number of data objects into a fully fledged set of game data.

- **Chapter 14: Exponential Growth and Diminishing Returns**

Exponential growth is one of the most powerful methods of balancing modern games. This chapter covers why we use this method and

explains a formula you can use to quickly create a nearly infinite number of varieties of exponential growth in games.

- **Chapter 15: Analyzing Game Data**

An important step in understanding a game as a whole is to evaluate all of its objects together, whether it's a small set of 10 objects or tens of thousands of objects. This chapter covers how to collect data in a spreadsheet and get started doing basic analysis.

- **Chapter 16: Macrosystems and Player Engagement**

You can use several different styles of difficulty adjustment to make a game harder or easier or to adjust a game to a player's particular needs. This chapter provides a high-level overview of various methods and gives examples of how these methods can be used in a variety of situations to get the proper balance for a game.

- **Chapter 17: Fine-Tuning Balance, Testing, and Problem Solving**

Much of a game designer's time is not spent designing but balancing, testing, and problem solving. This chapter covers methods of making these important tasks easier and more productive.

- **Chapter 18: Systems Communication and Psychology**

Games can be delivered to an audience in a variety of ways. A designer must consider how a particular game gives information to players and receives information from them. This chapter covers many of the aspects of communication with players.

- **Chapter 19: Probability**

Not everything is predictable in the world or in games. However, it is possible to understand some unpredictability. This chapter introduces you to basic methods of calculating and understanding game probability.

- **Chapter 20: Next Steps**

This final chapter gives you some more direction toward further growth in the world of game system design.

Register your copy of *Introduction to Game Systems Design* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN **9780137440849** and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

First, I must thank my wife, Melanie Gazaway, who stood by me while I lived this, encouraged me to write it down, and helped me find all my worst typos before I sent in the book for review. Next, I want to thank my children, Mazzy and Jack, who had to put up with an awful lot while I was working in the game industry. From late nights at the office to missed vacations, I was not always able to be there for them when I wanted, but they never made me feel bad about it.

Next, I want to thank my parents, Armen and Michael Gazaway, who raised me as a gamer nerd. I certainly would not be where I am today without them. Michael was my dad and first dungeon master. He was the first person I knew who designed and modified games. He taught me the fundamentals of game design before most kids even knew there was such a thing. My mom, Armen, read me *Lord of the Rings* as a bedtime story and let me skip school to see *Star Wars* on opening day. Even now we discuss games, sci-fi, and fantasy movies as a normal part of conversation.

Beyond my parents, Rick Herrick was a family friend and huge gamer influence on me. Scott Stocklin and Jesse Wise were childhood friends who introduced me to even more games and were the test subjects for some of my earliest and worst attempts at making my own games.

In college, I was in the “crucible of design” where my group of friends were constantly making and playing each other’s games. It was in that time that I developed more quickly than at any other time before becoming a professional. I would especially like to thank my gaming group, including Dax Berg, Goose, Todd Meyers, Ron Mertes, Skip, Foz, the Chads, Pig Man, Sarah Lacer, Marie, Glenn, Connor, Evan, and all the Daves.

Once I became a professional, the 3DO team was a tremendous help. Special thanks in particular to the leads of the team, Jason Epps and Howard Scott Warshaw (yes, THE Howard Scott Warshaw). They guided

me from being a very fresh rookie into becoming a professional game designer.

I first heard the phrase “game system designer” with the Lucas Arts Team, and once I heard Chris Ross say it, I was hooked for life. In addition, he and Dan Connors were very supportive in letting me explore this new unofficial title to figure out what it meant. I cannot thank the Gladius team enough. They were all great, and I learned a ton of what is written in this book while working on that team. Special thanks go out to the system team of Alex Neuse, Derek Flippo, and Robert Blackadder.

The Vicarious Visions Team brought me on specifically because I was a system designer, and that was the direction they wanted to take the studio. This was a massive responsibility, and I learned an incredible amount while working there. I had more friends at that studio than I can name, so I will say special thanks to my system team of Dan Tanguay, Jonathan Mintz, Alan Kimball (programmer extreme and honorary system guy), Jay Twining, Justin Heisler, Mike Chrzanowski, Brandon Van Slyke, and Jessica Lott. Thanks to Tim Stellmach for introducing me to Bad Storming.

Row Sham Bow was the last professional studio I worked at and easily the best. Every single person there was amazing. The studio set the bar so high for me that I will only ever consider working at a studio this great in the future.

I would like to thank the Full Sail team. I love teaching and sharing my experiences with enthusiastic, motivated students who are at the beginning of their game design journey. In specific, several of my colleagues encouraged me to write this book and provided valuable feedback as I did so. These include Zack Hiwiller, Ricardo Aguiló, Fernando De La Cruz, Christina Kadinger, Andrew O’Connor, Hayden Vinzant, Paul Fix, Derek Marunowski, and Phillip Marunowski. A special thanks also goes to my interns and those wonderful students who kept coming back for game days.

Finally, I want to thank all my wonderful students. Seeing their passion and enthusiasm keeps me feeling young and passionate about this profession. I wrote this book for them specifically. It took me over 20 years to accumulate the knowledge I am presenting here, and now I am passing it along to the next generation. My greatest hope is that I can make their

journey easier than mine was, as all my mentors made my journey easier than theirs was.

About the Author

Dax Gazaway was raised in a gamer family. His parents met in a Dungeons & Dragons group, and he was surrounded with games being played and made. From a very early age, Dax was fascinated by the numbers in games. He would pour over monster manuals and board game books, dissecting the rules to figure out how the systems worked.

Dax started in the video game industry in the late 1990s. During his tenure in the industry, Dax pioneered game system design at multiple independent and AAA studios, helping to refine and define the subdiscipline. In recent years, he has become a course director at Full Sail University, specializing in teaching new students the concepts and tools of the system designer. Dax has created new curriculum and multiple classes for system design students, and he teaches introduction to system design courses.

The following is a selection of Dax's game design credits:

- *Star Wars: Obi-Wan*, System and level designer
- *Star Wars: Jedi Starfighter*, System and level designer and QA liaison
- *Star Wars: Bounty Hunter*: System and level designer
- *Gladius*: System designer
- *Syphon Filter* franchise: Lead designer and system designer
- *Spider Man 3*: Lead system designer
- *Marvel Ultimate Alliance 2*: Lead system designer
- *Guitar Hero* franchise: System designer

In addition, Dax has been the studio lead system designer for Row Sham Bow Games and a system design consultant for multiple projects.

Chapter 1

Games and Players: Defined

To begin learning any topic, it's important to agree on terminology. This chapter defines a number of terms in order to establish a common vocabulary to use throughout this book. This chapter begins by providing working definitions of *game* and *game player* and then looks at a functional definition in detail that we can use to differentiate what is a game from what is similar to a game. Furthermore, we break down the various attributes that define game players in a practical way that you can use to guide your game development.

Defining *Game*

We need to define what a game is before we can define *game systems designers* or *game data designers*. While you could philosophically argue that “Life is a game” or “The entire universe and everything in it is just a big game,” for the sake of this book, we need to more narrowly focus on the activities that are commonly seen as games, and we need to look at why those activities are different from everything else.

Games are human-made creations that have a few specific attributes. Instead of trying to sum up everything that it means to be a game, it's more valuable to look at various attribute criteria that must occur in order for us to have a game. In a way, we are creating a checklist that requires all the checks be marked in order for what we are looking at to be considered a game:

- A game has agreed upon, artificial rules.
- A player can have impact on the outcome of a game.
- A player can opt out of a game.

- Game sessions are finite.
- A game has intrinsic rewards that hold no extrinsic value.

The following sections look at these attributes one at a time to build a better understanding of the term *game*.

Agreed Upon, Artificial Rules

When one animal is hunting another, that is not a game. The goals in such a scenario are to kill or survive. There are rules in the hunt, but they are dictated by biology, physics, chemistry, and many other factors. While there are physical limits to what can be done in the hunt, there are no artificial limits; there is no such thing as cheating in the hunt.

So hunting is not a game, but we can observe many animals playing hunting-like games in the wild. Bear cubs may wrestle but not attempt to hurt each other. Wolves may chase each other as they would prey—but without attacking. These, essentially, are games. We, as humans, tend to think we have the monopoly on creating games, but we certainly do not.

Now let's consider two forms of human combat: street fighting and collegiate wrestling. In a street fight, there are no rules. Street fighting is against the law, so there is not even a semblance of any rule in the activity. If one combatant is clearly larger and more skilled than the other, the fight will likely be lopsided. If one of the combatants can grab and use a weapon, he might be able to gain the upper hand. Whatever the physical circumstances of the environment allow can happen in a street fight. So, a street fight is indeed combat, and it could even be considered some form of contest, but it is not a game.

On the other end of the human combat spectrum is collegiate wrestling. It is also a very physical contest that can resemble a fight, but it is a game. So what makes it different from street fighting? The rules. In wrestling, the players are not allowed to use weapons. They are not allowed to punch each other, even though they clearly have the means to do so, and it would almost certainly be an advantage to do so.

Now let's look at a contest that involves no real physical contact, let alone combat: checkers. When two players sit down to play checkers, the goal is for one player to capture all of his opponents checkers. Although there is nothing physically stopping the player from simply reaching over to the other side of the board and grabbing his opponent's checkers, he is not allowed, by the rules, to do so. To play the game, both players have agreed that they must create a specific set of circumstances in order to remove an opponent's checker. Only when the players have followed the artificial rules of the game and one of them has created the circumstances to remove all checkers can that player be declared the winner. In the rules of checkers, the set of circumstances needed to remove a checker is exclusive—which means that any move that does not strictly adhere to the stated rule is assumed to be against the rules.

If the rules of a game were not agreed upon, the game would quickly break down and fail to function. If there is an established set of rules for a game and a player chooses to play that game, we can implicitly assume that the player has agreed to all the rules—regardless of whether that player knows or understands them.

While all of the attributes that make a game a game are important, the most important one is that a game has agreed upon, artificial rules. Without these rules, there is no game.

Players Have an Impact on the Outcome

This attribute is actually slightly controversial. A small number of people believe that a game can be a game even if players have no impact on the game. However, for the sake of our definition in this book, if a player does not have an impact on the game, she is not a player at all, and the activity is not a game. Let's take a look at a couple of examples.

In the game Candy Land, players take turns drawing a card and then moving an avatar along a linear track toward a goal. Is this a game? According to the criteria we are putting forth—that a player must have an impact on the game—it is not. A player does not have any agency in Candy Land. Imagine that one person were to play Candy Land as all four avatars. Could this player have enough influence to make a preferred avatar win?

No. Candy Land would proceed in exactly the same way with one person doing all the moves for all the pawns as it would if four people were making moves individually for each pawn. It would proceed identically regardless of the desire, skill, or intent of the player.

In contrast, with chess, if a single player plays the turns for both sides, it is easy for that player to decide which side will win and which will lose. In fact, most people find it difficult to play chess against themselves because it is too easy to favor one side winning without even trying.

People Can Opt Out

So far, we have established that a game has artificial, agreed upon rules and that people involved in a game can have an impact on it. Laws fulfill both of these requirements, but clearly, they are not games, and we would not consider lawmakers to be game designers. We have a defined word for lawmakers for a reason. But what makes the law more than a set of game rules? The people involved can't opt out. Let's consider a couple of examples.

If a person is walking with a basketball and someone calls out, "That's traveling. You must be dribbling the ball while you walk," the person might respond, "I'm not playing basketball. I'm just walking, so it's okay." This is perfectly fine. People are never obliged to play a game.

On the other hand, if a person is walking across the street through traffic where there is no crosswalk, someone might say, "That's illegal. You are not allowed to jaywalk." This person can't get away with responding "I'm not playing, so it's okay." Regardless of whether they want, care about, or even know about the law, all people are required to comply with the laws of the location they are in. This is a key difference between laws and game rules: People can't opt out of laws, but they can opt out of games.

Game Sessions Are Finite

A game must have some form of ending so that participants are sometimes "playing the game"—but not always. A massive multiplayer online game (MMO), for example, may go on indefinitely, but players do stop playing

from time to time. The rules of the game exist for those who are playing but not for someone who has finished a session.

With a social club, such as Boy Scouts or Girl Scouts, there are artificial rules, the members can have an impact on their status—and therefore the outcome of the membership—and the members may opt out (quit) if they want to. So, based on these three attributes, these kinds of organizations could be considered games. However, they do not have finite sessions.

When you join a club or another social group, you agree to follow the rules of the group at all times—and not just during specific times. This makes social clubs, music bands, theater troupes, and many other interests distinctly different from games.

Intrinsic Rewards

Based on the attributes mentioned so far, we might consider a job a game. It has artificial rules dictating what you should be doing when and where. You can opt out, and it has finite sessions. So what differentiates a job from a game? One more aspect that distinguishes games is intrinsic rewards—that is, rewards that have value only in the game. For example, when you play checkers, it is valuable to capture an opponent's checker. However, that captured checker holds no value outside the game. At the end of the game, all checkers are returned to the set to be redistributed at the beginning of the next game. At a job, on the other hand, the reward for work is usually money. The financial reward does not lose value outside the job and can be used in the larger economic system.

Let's consider a slightly more complex example: poker. We generally recognize poker as a game, although some people play it as their profession. In this game, the players play for money that has an extrinsic value, but there are significant portions of the game that hold only intrinsic value. For example, in the context of poker, having cards of the same suit and in order is considered highly valuable. Outside the context of poker, randomly drawing five cards of the same suit in order is still very rare, but it has no value. It's the addition of intrinsic rewards that changes a task, an errand, or a job into a game.

Game Attributes Summary

By bringing together all the attributes of games we have just discussed, we create a unique definition of *game*. Every game discussed in this book has all these aspects.

Determining what is a game and what isn't can get rather complex. Jobs can have games in them. Games can have more games in them. Some activities can be called games but are missing a crucial needed aspect (such as Candy Land lacking player influence). Despite these oddities, we now have a clear enough idea of the attributes needed to call something a game that we can go forward in this book with a shared definition for reference.

Puzzles and Toys

Puzzles are a large and unique game sub-genre. A puzzle has a specific correct answer. With a puzzle, you either solve it according to the rules, continue to work on it, or give up. A jigsaw puzzle, for example, has the attributes of a game, but you can't lose; you can only stop playing.

When designing a large game that contains subgames and puzzles, keep in mind this distinction: If a player can fail, it's a game; if a player can keep trying, without punishment, until he solves the challenge, it's a puzzle.

Toys don't have rules, so they are easily distinguished from games. Toys are similar to games, but with a toy, you can do what you want within the bounds of physics. By this definition, many computer games and many modded computer games are toys, rather than games, because it is entirely possible to change the rules and, therefore, the outcome of the session.

Finding the Target Audience for a Game: Player Attributes

Once you know the attributes that constitute a game, it is important to consider who you are making a game for: Who will be the game player? And, more fundamentally, *what is* a game player? It could be argued that

literally everyone in the world is either a current or former game player; most people enjoy and sometimes play some sort of game.

Note

Don't try to define the audience you want to play your game; really, you want everyone to play it. Instead, define the audience for which you are making the game. Many players' preferences are mutually exclusive. For example, some players won't play games that have sessions that last longer than a few minutes. Other players won't play games whose sessions last less than an hour. It is literally impossible to please both of these groups, and you should not try to do it. You can, however, decide which group you want to please and make your game accordingly. So, you can accept and embrace anyone who wants to play the game you are making, but you can only attempt to lure a much narrower audience as you make the game.

To define your target audience, you can think about various attributes, much as you will do for game objects later in the book. Attributes of game players run the gamut from demographic characteristics to attitudes and behaviors. The following sections describe a number of game player attributes that are just a sampling of a nearly infinite number of ways you could describe a game player. Use this information as a starting point for your own exploration rather than as a template for what is "the right way" to define a game player.

Age

Age can most effectively be split into two major groups: adults and developing children. As children develop cognitive skills, they do so (very roughly) by age. Games made for developing children have specific requirements and face legal restrictions. As a game developer, you must take into account the cognitive development of children of different ages if you want to make games for them. There are entire fields devoted to child development and making learning activities or games for children at

different stages of development. If you are planning on making a game for children, you need to do some research into the field of child development.

Adults, on the other hand, can be grouped into one large pool for the sake of game development. You can assume that all adults that you make games for can read, use a controller or keyboard, understand basic mathematics, and work out puzzles on their own. This makes the categorization of adults by specific age nearly meaningless. While there are some cultural references that generations will interpret differently, there is not much to be gained from splitting up age any further, so when defining a target audience, it is generally sufficient to say the audience is “any adult.”

Gender

There is really no need to dive into the politics of gender because gender is not a particularly effective attribute for defining a gaming target audience. While a few games do target a specific gender, most don’t because there is very little that has to do with belonging to a gender in most games. So, unless you have a good reason to include gender as an attribute in your game’s target audience, there is no need to include it at all.

Casual Versus Hardcore Gamers

What is a casual gamer? What is a hardcore gamer? It’s nearly impossible to define either one. For example, Garry Kasparov is a world-renowned chess champion. That should qualify him as a hardcore gamer, but what games would you build for him besides chess? You might know someone with a full arm sleeve of gaming tattoos, but she only plays specific RPGs from Japan. Is she a “hardcore gamer”? Culturally, yes, but again, the definition does not really help target that player while making a game. What about a player who buys a new video game every week, plays it for a week, and then moves on? This person sounds like a hardcore gamer, but this person is completely different from Kasparov and your tattooed friend, who could also both be considered hardcore gamers. And what is a casual gamer? Is this person just someone who plays games infrequently? Does it mean someone who plays a certain game genre?

As you can see, it would be pretty difficult to use the idea of casual versus hardcore when creating a target audience. The following sections describe several more measurable attributes you can use instead for defining the audience for a game.

Tolerance for Learning Rules

Many people hate learning new rules. If you doubt this, hand someone a thick rule book and ask her to read it. Many people would think this sounds awful, right? These people often decide it is not worth the effort to play any new games. Think of people who play chess on a daily basis but no other games. They are certainly gamers, but they're not the same kind of gamer who picks up a new game weekly just to give it a try.

On the other end of the spectrum are players who enjoy learning the rules of a game; they might even enjoy learning the rules as much as or more than playing the game itself. These people might belong to a local board gaming club, where they play a new board game every week, rarely replaying a game. The draw of the game for this group is learning new rules with the group, and the games themselves are secondary.

As you create a game, you need to decide what tolerance level for learning new rules your target audience will need to have. People basically fall into five groups when it comes to tolerance for learning rules:

- **Refusing:** Players in this group all but refuse to learn new rules. If you are making a game for this group, it is almost certainly a game based on a real-world game, or it is a sequel to an existing game, or at least it fits tightly into a very specific genre. For example, Wii Bowling was able to grab an audience of people who had no interest in video games but already liked bowling. It is fairly rare to design games for this group, and doing so is much more challenging than creating a game for a more accepting audience.
- **Resistant:** Players in this group do not like learning new rules, but they will do so occasionally so that they can play a specific game. Players in this group tend to stick to a small variety of games, or possibly a

narrow genre, and tend not to explore new titles beyond what they already know. Designing games for people in this group is easy if you are doing something that would be familiar to them and takes very little ramp-up. However, members of this group are nearly impossible to reach if you are doing something unique or well outside their comfort range. Think of the “match three” genre of games, like Bejeweled. Many players play only this genre because it is easy to learn and play and is available on a convenient device, such as a smartphone. If you wanted to make a new twist on a match three game, this group might be open to giving your game a chance. However, you shouldn’t expect them to wander outside of this narrow focus willingly.

- **Neutral:** Players in this group are fine with learning new rules if it is for the specific purpose of playing a new game. Many casual video game players and board game players fall into this category. This group is unlikely to read or understand all the rules; instead, they learn as they go, and they may be interested in only the rules that are needed to play the game. The neutral category is a large segment of the market. Many, many people fall into this category, and it may be worthwhile targeting them, even though getting the rules through to them can be challenging. Players in this group are the kinds of players who follow several specific franchises and buy each new installment of those games. They are also likely to buy similar competitors within the same game genre, but they are unlikely to wander outside their preferred genre to try something completely new. For example, think about players who buy every variant of first-person shooter (FPS) that is released but rarely if ever buy games from completely different genres. This kind of player might be convinced to try something new to play with a friend but would not seek the experience without that kind of prompting.
- **Accepting:** Players in this group like learning new games and new rules. They often learn more rules than the bare minimum needed to play. They sometimes seek out rules exploits and use them to their advantage. This group is also likely to have a favorite genre and stick to it mostly but also explore other genres. The accepting category is a significant market and makes up a good share of professional game developers’ target market. There are far fewer gamers in this category

than in the three more resistant categories, but there are enough to warrant creating large-budget console and PC games for them. This is the group that is most targeted and valued by the gaming industry, and accepting players are often flooded with a variety of games made for them. This can have an oversaturating effect on the group, and each individual game created for the accepting crowd is likely to face challenges in terms of standing out in the crowded field.

- **Enthusiast:** Players in this group love to learn new rules. They often jump from game to game, and many of them drop a game as soon as they figure out the rules completely. While it is easy to get a rules enthusiast to play your new game, it is just as hard to hold on to that player. This group is also willing to accept many more rules and much higher barriers to entry than are the other groups. While there are no exact numbers on the size of each group, sales of games with many complex rules provide significant anecdotal evidence that this is the smallest of the five groups.

Interest in Challenge

There are many varieties of challenge, but we can simplify this attribute to tolerance for failure. Most games have mechanics that allow the player to both succeed and fail. We can quantifiably measure the frequency of success versus failure and use it as a metric that defines challenge. To some players, failure is almost—or even completely—unacceptable. They want constant rewards and play games simply to feel good or pass time—and not for challenge. Other players accept or even crave high failure rates. It is possible to quantify this aspect by listing an accepted failure level as a simple ratio of the number of failures to the number of successes. Consider these two dramatically different ratios:

- **15/1:** This notation indicates an expected 15 failures for every 1 success. This is a very challenging game; it is so challenging that it will likely drive away many players, but it may attract a small devoted following.
- **1/100:** This notation indicates an expected 1 failure for every 100 successes. This would be a game that is not intended to frustrate players. Story-driven RPGs and simple puzzle games may have this

type of accepted failure level. The challenge of the game is not really the point of such a game, and so challenge is heavily downplayed.

Beyond describing the player you want, this player attribute can help you determine the game's success after it is released. The success and failure rate of players is one of the easiest metrics to acquire and analyze from play tests and telemetry data. This attribute allows you to compare your expectations with the reality of the game. It can also let you know whether you need to make adjustments to the game to make it more or less challenging to better target your desired market.

Desired Time Investment

Desired time investment can be broken into two subcategories: session time and total time. Some games, such as checkers, don't have a total time, only a session time. But many games have both.

Session time is how long it takes to play a standalone session of a game or, in longer video games, to have what the player would consider a satisfying session. In some games, such as poker, a single hand could be considered a discrete session; this would be a very small session, and most poker players would consider a tournament or a night of many hands to be a discrete session. Tennis is also a well-divided game. It consists of points that make up games, which make up sets, which make up a match. So, even though you can play a game of tennis in just a few minutes, it's not considered a full session. Instead, a match is considered a full session, and a match can last an hour or more. In game design, you should consider a session to be something that a player can walk away from feeling satisfied about completing.

Total time is the amount of time it takes to “complete” a game—which can be very different for different games. Some games are never completed. Think of MMOs, match three games, and older arcade games. Players may come up with their own definitions for total time, like “hit maximum level” or “earn all achievements.” So you can see that it's not necessary to have an end to game. It is, however, important to define a session and a complete game experience for the game you are making.

For each of these time investment subcategories, you need to define what you expect out of a game and what the audience wants. Some players want a quick 5-minute session with no end game; casual mobile puzzle games suit this audience. Other players want 2-hour-long sessions with a 20-hour completion time; many action and adventure video games appeal to this group. Different time investments appeal to different players. You need to know what game you are making and know how to test your game with the proper audience to get accurate and useful information about your audience.

The following are some examples of how you might quantify desired time investment for various games:

- **Checkers:** 10-minute session, no total time
- **Poker:** 3-hour session, no total time
- **Skyrim:** 2-hour session, 100-hour total time
- **Mario Kart:** 15-minute session, 50-hour total time to unlock every reward

Pace Preference

The pace of a game is difficult to quantify but is important to players. Asking someone who likes turn-based strategy games to play a high-action FPS will not likely yield good results and vice versa. You should consider multiple factors related to pace. For example, a game of speed chess is fast paced, but it is fast in a different way than a “shoot ‘em up” top-down shooter is. When you want to quantify the pace preference of players, you should specify what specific type of pace you mean. For example, some games have purposeful rhythms and go through pace changes. Stealth games, for instance, often have a slow methodical buildup, an explosive fast section, and a slower-paced cool-down section.

When describing pace preference, it is important to be as accurate and specific as possible. The following are some examples of how you might quantify pace preference for various games:

- **Chess:** Slow, turn-based, thoughtful pace
- **Ping Pong:** Very fast, athletic, reflex-based pace

- **Generic computer RPG:** Alternating between slow and mid-paced action with some slower thoughtful crafting moments throughout
- **Generic deathmatch FPS:** Fast-paced, high-dexterity, with very little downtime for breaks
- **Generic 1980s arcade game:** Moderate and dexterity-based pace to start that gradually ramps up to fast dexterity until the player can't keep up; play then starts over

Competitiveness

How much do players want to be judged or compared to one another? Some games are built specifically around competitiveness, and others avoid it completely. For example, all racing games clearly rank players based on how they finished the race. Many go further and show campaign-style rankings that accurately display how well each player is doing in comparison to other players. On the opposite end of the spectrum are sandbox games, farming simulations, and other casual games. It is often hard to determine player ability in these games at all, let alone to compare one player quantitatively against another.

As with the other attributes of a game player, it is important to know who you are making a game for and what kind of feedback to provide them. For example, if you were making a highly noncompetitive game, you would want to avoid leaderboards, ranking, and maybe even visible scoring of any sort. Players who like noncompetitive games don't want to know how they are doing and just want to enjoy the experience. If you were making a competitive game, you would need to plan well in advance how to track, measure, and display relative ability to the player.

Regardless of which way you want to go with a game, it's a good idea to keep competitiveness in mind from the outset. When creating a player profile, you can use any description of the competitiveness that suits the game you are making. The following are a few examples of how you might describe various levels of competitiveness:

- **Noncompetitive:** The game has no scoring, no ability tracking, and no player-versus-player comparison. It is meant to be easy and fun to play

and not make players feel bad about their current ability.

- **Competitive:** The game tracks a few key metrics on a per-session basis. At the end of a game session, players are shown a score and given a ranking for that session. However, each session is individual, and the scores are wiped clean at the end, allowing the players to feel a bit competitive without being judged overall.
- **Highly competitive:** Every possible aspect of play is tracked, measured, and displayed in detail for the player. Individual sessions are tracked and compared, and global leaderboards are dynamically maintained and shown to the player. Players are encouraged to constantly refine their skills and are driven by climbing leaderboards of different time scales (for example, weekly competitions, daily challenges, and best-ever scores).

Platform Preference

Games are available for many platforms, including PC, console, mobile, gambling table games, and VR games. (You could even consider tables and pen and paper to be platforms.) Knowing the specific hardware platform, or multiple platforms, for a game will help define your audience—though not in a deeply meaningful way. There are now games of all types on every platform, and there are audiences for each of those types of games on each platform. It is good for you as a game designer to know the strengths and weaknesses of the hardware for which you are developing. You should therefore always be thinking about platform, although it is not as important to think about console players versus PC players versus mobile players.

Skill Level

Many games require that players bring some skills with them or quickly develop those skills while playing. Like all the other attributes, requiring specific skills for a game will attract some players and repel others. When designing games for adults, you can assume that players have reading skills and basic motor dexterity skills. A game may require many other particular skills, and it's a good idea to list them early in your development process.

Consider these examples of the skills that some games require:

- **Tennis:** High degree of hand–eye coordination and physical endurance
- **Chess:** Ability to think out future moves and options
- **FPS:** High degree of hand–eye coordination and fast reflexes
- **Match three:** Pattern recognition
- **Sandbox building:** Spatial awareness
- **Poker:** Probability calculation

Considering Inclusivity

Not every game player is born with or currently has the same physical and mental capabilities. Color blindness, for example, is particularly common and can severely limit a player trying to learn a new game. Taking even small steps as a game designer to think about inclusivity in design can increase the potential audience while also doing something positive for society. While the subject of design for inclusivity is too large in scope for this book, it is a topic that should at least be kept in mind right from the very beginning when designing a game.

Genre/Art/Setting/Narrative Preference

This catchall category allows you some freedom to more specifically define what your audience needs to be interested in. Some players are turned away by specific art styles. Some have a strong preference for specific settings, and some may be more interested in the story than in the mechanics and gameplay of the game. This category can also include things like outside interests and hobbies your target audience will likely have. For example, if you are going to make a game based on being a stock trader, players probably need to have some interest in stock trading.

Value Gained from Players

Another important factor to consider when designing a game is what value players are going to give to you. After all, you are going to put a

considerable amount of time and effort into entertaining your audience, and you probably want some form of compensation from your audience in exchange for that experience. Money is the most obvious reward, but it is not the one answer. As discussed in the following sections, there are many types of value you can gain from players at different stages in the cycle of a game's life.

Payment

The simplest form of compensation from players is payment. This could be as simple as each player purchasing a copy of the game. Modern games, however, have much more sophisticated methods of getting payment. Each form attracts some players and repels others. Knowing the kind of payment you want to get from your players from the start is an important factor in determining how the game is built. The following sections describe a few common types of payment and some pros and cons that you should consider for each type.

One-Time Purchase

With a one-time purchase, a player buys a game and can keep it forever. This is the oldest and most straightforward method of player value. People have been making one-time purchases of board games and decks of cards for centuries—and maybe longer.

Pros

- Players pay for the entire experience up front. This is often a larger amount than is required with other types of payment.
- The game can be considered done at some point. Many modern games are in production long after they ship or are even in perpetual development. This is expensive and requires a great degree of organization. With a one-time purchase, once the player buys the game, the obligation of the developer is over.

- When the game is done, the development team can move on and fully focus on the next project.

Cons

- There is no opportunity to make more money after the initial purchase.
- There can be little incentive to fix flaws discovered after the game shipped.
- There is no opportunity for the development team to get paid to continue working on the game.
- There is no external incentive for players to return to the game for new content.

Expansions Purchase

With an expansions purchase, players buy a core game and then may buy expansions for the game at a later time to modify or expand the central game. These are usually self-contained and fairly small in number. For example, a new expansion pack may come out twice a year.

Pros

- This format potentially keeps players wanting to revisit the game, which increases other forms of value (listed below).
- It is possible to tweak balance and fix mistakes. When an expansion is released, especially in a video game, the developer has an opportunity to sneak in some bug fixes or tweak balancing flaws discovered after the game launch.
- The game can be shipped quickly and on a smaller budget since it is expected that more content will be released for the game later.
- The game can be sold for a lower price because more money is expected from players for the expansions.
- The game developers can make more money on a game they know sold well instead of taking a risk on a new game.

- Publicity is important. When an expansion for a game is released, it is a good focal point to draw in new players and to get players who have moved on to come back.

Cons

- Players may feel as though the initial game is not a full experience.
- Part or all of the team is tied up working on game expansions instead of new projects.
- Each expansion pack involves many of the same obstacles and headaches as launching a game, without the benefit of launching a new game.
- The audience is much more narrow for an expansion pack than for a new game. The audience for a new game is potentially everyone, but typically the audience for an expansion is some fraction of the audience that has already purchased the existing game.

Microtransactions

With the microtransactions model, players buy or get the game for free to start. Then, while playing the game, they can make in-game purchases of things as simple as decorative changes all the way up to new levels and mechanics.

Pros

- There is a continuing possible source of income from players.
- Games can be initially priced low, or even free, which greatly expands the potential market of players.
- Players can specifically tailor their purchases to their own taste.

Cons

- Purchasing in game can feel like cheating in a competitive game.
- Players may feel that they are being “nickel and dimed.”

- The more the developer adds break points in the game to attempt to sell items to the players, the more the flow of the game is broken.
- Much of the developed content may not be used if players don't buy it.
- The base game may feel unsatisfying. Conversely, the base game may feel good enough that players decide they don't need to buy anything and, therefore, the game does not make money.
- Some players and developers have moral issues with trying to extract numerous microtransactions from players.
- There are some legal issues that need to be investigated around the use of microtransactions, depending on where the game was made and where it will be released.

Advertising

With advertising, players pay a reduced amount for the game or no money at all, and the developer makes money from in-game advertising.

Pros

- The game can be inexpensive or free for players, leading to a potentially larger audience.
- The game has a continuing source of income.
- The payment is “fair” in that players who watch advertising are not purchasing an advantage in the game.

Cons

- The player is pulled out of the game experience by the ads. Often, the developer has limited or no control of the ads being shown. The ads may have a conflicting message to the intention of the game or may even be for a competitor.
- Ad revenue tends to be rather low on a per-player basis. It takes a whole lot of players seeing a whole lot of ads to pay a development team.

- There are a few technical issues with including ads in a game, though modern game engines and platforms have greatly reduced this difficulty.

Other Forms of Value

In the end, money is the goal of professional development, but there is not always a direct path from creating a game to piles of cash. Developers can gain other forms of value from players as well. When developing a target market, it is good to list, right from the beginning, what forms of value you want to gain from your players. When thinking about the additional forms of value, it is tempting to want them all. But you should avoid this temptation and pick only one or two to focus on. Trying to target several secondary values at once is likely to water down individual efforts and weaken the chances of any of them working.

The following are a few of the ways types of value besides money that you can gain from players:

- **Word of mouth:** When players like a game and discuss it with their friends, their friends are more likely to get the game. When a game has mechanics built into it that get players to play with each other, seek out people to play with, or just talk about the game, there are more opportunities to bring in a bigger audience. When targeting word of mouth as a method of getting value, you want individuals to talk directly to other individuals because they are likely to be trusted by the people they are talking to.
- **Social media:** Social media is an even more expansive but less personal form of getting the message out than word of mouth. It essentially allows players to act as free advertising. While social media can help a small game reach a much larger audience, there can be backlash over “spam” games in social media. If a game is constantly asking players to share scores and invitations with new people, it does reach more people but at the cost of annoying potential players.
- **Popularity contests:** “Game of the year” and similar voting contests are common across the Internet. If a small game with a motivated audience can win or even place in one of these contests, it can give the

game a large amount of exposure essentially for free. Note that this is a particularly difficult method of value to plan for with a new game. It tends to occur more organically in response to a game that is already on the market.

- **Ranking sites:** Many websites have ranking lists of new games based on player scores. Games with high scores are typically more attractive to new players. Some small games have broken out and become successful simply by having a consistent and growing following of high scores on ranking sites. It might seem like ranking sites would be an obvious target for every game, but it's really not. Making a game that consistently gets high ranking scores is often expensive in comparison to making a game that gets good scores. Or, to put it another way, there is an ever-diminishing return of ranking scores based on the input of every growing effort and money put into the development of the game.
- **Content creation:** A game may allow players to create content within the game. Players can produce more content than any single development team ever could. This model tends to create an enthusiastic and dedicated player base. There are, of course, many development hurdles in creating a game that players can manipulate and expand. These hurdles may be more than many teams want to tackle.
- **Player interaction:** MMOs and other multiplayer games are good only if people play them. Simply by being in the game and playing, a player adds value to the game. Some multiplayer games need lots of players to do adequate matchmaking. With these games, too, by simply being in the pool of players, a player is adding value to the game.
- **Market numbers:** Games that are popular and have a lot of downloads or active users get attention. For this reason, some teams are willing to take a financial loss in order to encourage the largest volume of players to play the game. One of the easiest ways to become successful while selling any product is to already be popular.

Target Audience Value

Obviously, a game developer would like to gain as much value as possible from each player, but again, many of the methods listed in this section for gaining value are mutually exclusive. Having a plan upfront about the methods a team wants to use to gain value from players will help guide the team in a single direction.

For example, if the team has a small budget and no established reputation, simply getting noticed is going to be very challenging. In this case, the team might decide to get value from players by giving away the game for free, being ad supported, and trying to gain more value from encouraging players to spread the word on social media. Conversely, if the team is more developed and wants to have a gamer-friendly reputation, it may opt for an upfront cost to the game and offer expansion packs. It might at the same time target high game ranking and possibly some online awards to gain value from players.

Target Audience Composite

Once you have determined the attributes you want and don't want in the players you are making a game for, you need to do something with this information. One thing you can do is create a target audience profile. The following sections provide examples of target audience profiles for various commonly known games. These profiles were created based on games that already exist—and they show that you can examine target audience profiles after you put out a game in order to look at the kinds of players you ended up targeting.

Chess

The target audience profile for chess is as follows:

- **Learning new rules:** Resistant. This player is willing to learn a small number of unchanging rules so they can play the game repeatedly with the same or new opponents around the world.
- **Challenge level:** Variable. The player will find the right challenge level by finding the right opponent. There is no single-player version.

- **Time investment:** Sessions are around 30 minutes; there is no total time.
- **Pace:** Very slow and thoughtful.
- **Competitiveness:** Highly competitive. One player will win or lose every session. The drive to beat the current opponent is a motivating factor for playing the game.
- **Platform:** Analog board and pieces.
- **Skills required:** Very little. Only the ability to comprehend the basic rules.
- **Genre/art/setting/narrative:** None.
- **Value gained from player:** Word of mouth and player pool.

Galaga

The target audience profile for the arcade game *Galaga* is as follows:

- **Learning new rules:** Resistant. The player is not interested in learning complex controls and systems but wants to get right into the game immediately and figure out the mechanics through trial and error.
- **Challenge level:** Very high. Failure is assured with every session. The player is happy to start playing knowing that every session will end in failure.
- **Time investment:** Low. Sessions are around 5 minutes; there is no total time.
- **Pace:** Players likes to start at a moderate pace and keep playing as the pace increases to the point that they can't keep up.
- **Competitiveness:** High. Players can see their score and try and do better. A public leaderboard enhances competition at the arcade machine. Players are driven to beat their own high scores and get their initials on the public leaderboard.
- **Platform:** Arcade cabinet.
- **Skills required:** Hand–eye coordination and reflexes.

- **Genre/art/setting/narrative:** Spaceships and aliens.
- **Value gained from player:** Direct payment. A quarter gets a short one-play session.

Mario Kart

The target audience profile for *Mario Kart* is as follows:

- **Learning new rules:** Resistant. The mechanics and controls are in line with those of other racing games, so a player who has ever played any racing games will be able to quickly transition in. Further, a player who has driven a car knows the rough analogy the game is using.
- **Challenge level:** High. 5/1 failure rate. When players start, they lose against artificial intelligence (AI) most of the time. As players get better, the AI and challenge level increase consistently to keep players losing frequently. This ratio of loss motivates players to come back for more and improve their skills.
- **Time investment:** Low. Sessions are around 15 minutes. A total time of 100+ hours is required to unlock all rewards.
- **Pace:** Very high. There is no time for contemplation in any moves.
- **Competitiveness:** Very high. Players are ranked per race, and there are multiple leaderboards.
- **Platform:** Nintendo game systems exclusively.
- **Skills required:** Basic racing game skills.
- **Genre/art/setting/narrative:** Fun, bright, colorful artwork that appeals to children. Some adults may be turned off by the cartoon style.
- **Value gained from player:** Payment is primary, player interaction is secondary.

The Battle for Wesnoth

The target audience profile for *The Battle for Wesnoth* is as follows:

- **Learning new rules:** Accepting. The game is similar to many traditional turn-based strategy games but also has many unique mechanics. There are many different character classes and items to learn, and a wide variety of attributes. The player enjoys having a basic set of knowledge about the genre but also enjoys the twists and surprises the game brings.
- **Challenge level:** 1/1. Players tend to fail the first time and succeed the second on any given level. Players feel motivated by this pattern to replay a scenario multiple times and also explore new content in the game.
- **Time investment:** High. Sessions last around an hour, and there is enough content to play new sessions for hundreds of hours.
- **Pace:** Slow, thoughtful, turn-based pacing.
- **Competitiveness:** Low. While multiplayer mode is an option, most of the content is for single player mode. There is little emphasis on comparing players.
- **Platform:** PC, mobile.
- **Skills required:** Lots of reading, probability calculation, and strategic thinking.
- **Genre/art/setting/narrative:** Classic high fantasy.
- **Value gained from player:** Word of mouth, player content creation, and player interactions.

Bejeweled

For this final example, I have given the player a name and personality description to illustrate how such simple changes can humanize the target audience and help you better imagine how to make a game for those players. You can also add a few more details to help flesh out the character of your target audience. While this is not essential to get started, it can certainly help with visualization.

The target audience profile for *Bejeweled* is as follows:

- **Player name:** Chris.
- **Learning new rules:** Resistant. Chris does not play a lot of games, but she can easily figure out the basics of a match three game. She is motivated to find a few new combos and strategies as the game progresses, but she is not willing to go on forums or research a better strategy.
- **Challenge level:** Moderate. 1/10. Chris wants to be able to clear several boards of the game before the challenge starts to grow. Even at the highest challenge level, Chris can usually eke out a win and move on.
- **Time investment:** Low. Session length is around 15 minutes; there is no total length. Chris wants a game she can pick up and quickly get into and then complete a small session in just a few minutes. Chris does not want to invest large amounts of time in campaign modes or expanding mechanics.
- **Pace:** Chris enjoys slow and thoughtful turn-based moves. She does not want a game to stress her out.
- **Competitiveness:** Moderate. Chris mostly likes to play the game to kill a bit of time, but she enjoys occasionally checking the high score boards to see how she's doing compared to her friends.
- **Platform:** As many as possible, so Chris can play on any device available.
- **Skills required:** Chris is likely not color blind, but if she were, she might have a hard time with the core mechanics.
- **Genre/art/setting/narrative:** Chris likes the universal appeal of abstract decorations.
- **Value gained from player:** Chris has bought the game on some devices but plays with ads on others. Chris also likes to chat with friends about the game.

Do you feel like you can envision Chris? Maybe you know Chris or even are Chris! If you can envision Chris, and clearly see what will make her

happy or upset, it's much easier to make a game that will make her want to buy the game and continue to play.

What to Do with a Target Audience Profile

You spend a considerable amount of time determining who you want to make a game for and creating a target audience profile. You can use this profile as a guide through the rest of the game-making process.

You should refer to the target audience profile often as the game is being made. Each time there is a suggestion for a new feature, system, or data, the team should consider: Will this make the audience happy? If the team is debating multiple different routes for the game, it should think about which of them the target audience might prefer. By constantly coming back to the target, the process of making the game becomes more grounded. You are not making a game just for yourself, or for some unknown people. Instead, you are making it for a person. For humans, getting this granular makes concepts make more sense.

Using a target audience profile is also a great way to ramp up new team members. As we will discuss later in this book, the game development team is rarely static from start to finish. When the target audience profile is created, it's likely that the team will still be small and in a preproduction phase. New people who are added to the team need to understand not only how the game is being made but who it is being made for. By having a single, defined target audience, it's much easier to get new people on the team to buy into the vision of the team quickly and accurately.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the definitions of games and their players:

- Analyze several simple older games and decide for yourself if they meet the definition of a game and how. List the various attributes that make up the games specifically.

- Build several target audience profiles for your own games and other popular games. See if you can figure out why the game creators chose to target those specific target audiences.

Chapter 2

Roles in the Game Industry

The video game industry is relatively young compared to other creative industries. The breadth of things considered video games is also quite large: It can encompass everything from mobile games made by individuals, to AAA online video games created by teams of more than 100 people, to military simulation games, to an infinite number of other variations. Games more broadly span everything from board games, to card games, to sports, to all the other variations of games. Different games are created in different ways and require different team members.

A *game developer* (or *game dev*) is anyone who directly works on a video game, whether an engineer, a designer, or a sound tech. In some companies, only programmers are called game developers. However, this is a throwback to the early days of the video game industry, when all those who made games were programmers. Today it is generally accepted that more than just programmers are game developers.

A large game development team is likely to have a number of subdisciplines. On smaller teams, the same jobs typically need to be done, but a number of team members typically wear multiple hats. A solo developer, of course, needs to wear all the hats and do all the jobs.

A team—especially a large one—is almost never a static group whose members all work on a game from beginning to end. Throughout the course of a game development project, members are brought onto the team and moved off it. This chapter discusses a few of the roles that are common across the industry, discussing roles in roughly the order in which they come onto the team.

Core Management Team

The core management team includes the vision holder, the lead engineer, the lead artist, the lead designer, the producer, and the lead sound designer. This team tends to be consistent from the start of the project to completion.

Vision Holder

On each game development team, there is someone who is ultimately responsible for the game concept. This is the vision holder. This person either had the initial idea for the game, was assigned an idea by an outside group, or was put in charge by the team. This person must hold on to the core of the creative vision. This role is often filled by a company owner, a CEO, an executive, a producer, a creative director, or a director. The vision holder is responsible for protecting and guiding the creative vision and settling disputes in the team about the direction of the game as a whole. The vision holder is almost always the first person brought onto a team. A game may sit with the vision holder for months or years before anyone else is brought onto the team.

Especially in large studios/teams, the vision holder is the person involved the most in communication, meetings, traveling, and negotiation; this person is usually one of the team members least involved in the actual day-to-day making of the details in the game. While the vision holder owns the vision of the project as a whole, they actually have little insight into the minutia and must work with and trust the team to make a great game.

Vision holder is also one of the most senior positions in the game development industry. It takes years or even decades to climb the ranks from being a game developer on staff to being the vision holder. This position is often responsible for millions of dollars of invested money and the livelihood of an entire development team and maybe an entire company. This position would only be trusted to someone with an immense amount of experience.

Lead Engineer

Especially for video games, it's likely that one of the first people to join the vision holder is the lead engineer. The job of the lead engineer is to figure out technically how the game is going to be made. Will it be made using a

third-party engine, or an engine the company has already used, or a brand-new engine? What are the scope requirements needed to make the vision come to life? Who will the engineering team need to include to create this project? These are just a few of the thousands of challenges that the lead engineer faces while creating a game.

Lead Artist

Once a game has a creative vision and has groundwork being laid in the technical department, the team needs to start establishing a look for the game. This is where a lead artist comes in. This position faces many of the same challenges as the lead engineer when it comes to determining scope, staffing, and scheduling for the art portion of the project. In addition, the lead artist sets the tone for the look of the entire game.

Lead Designer

A game needs a creative vision, and groundwork must be laid for how the game is going to be technically built and what it will look like, but none of these things actually make it a game. The team needs a lead game designer, who is responsible for determining the makeup of the design team, scoping, and scheduling. In addition, the lead designer is at the heart of creating the rules and play of the game.

Producer

Producers are the coordinators and businesspeople of the creative world. The job of a producer is to make sure the game is made on budget, on time, and with the right staffing. Producers always work with the leads, subdiscipline leads, and the development team to ensure that the whole group is working as a team.

Lead Sound Designer

The control of everything audio often is led by a single person. While a small game may have one sound designer, a large game may have an entire sound team, led by a lead sound designer.

Team Subdisciplines

Subdisciplines on a team have more specific roles than the core management team and do more hands-on work. In most cases, each subdiscipline lead (commonly called a sublead) also reports to a member of the core management group. The following sections only briefly cover the disciplines outside of design. This is, after all, a book about design.

Art

Art roles include animation, character art, environmental art, concept/2D art, interface art, and technical art.

Animation

Animators make a game move and make static objects come to life. They start with a static mesh character and manipulate it to create animations. Every action that every character takes visually is the result of an animator's work.

Character Art

Character artists specialize in character models, which are called *character meshes* in the game industry. On big games with big teams, there are artists who are so specialized that their sole job on the project is to create character meshes—and there is plenty of work for them to do.

Environmental Art

Environmental artists are, as the name suggests, responsible for the look of the environment of a game. How much they do varies depending on the needs of the game. On some teams environmental artists work closely with level designers, on other teams they have free rein with a level, and on other teams they create the building blocks that level designers use to make the levels.

Concept/2D Art

Concept artists are most closely related to traditional 2D artists in the industry. Their job can start early in the development cycle. They work with both artists and designers to come up with visualizations of conceptual ideas. They may make images to guide the art team or to help a designer express an idea; they may also make posters and promotional art for the game.

Interface Art

Interface artists specialize in the interactive interface that players use to get deeper information from the game. They work on the heads-up display (HUD) and the user interface (UI).

Technical Art

Technical artists don't just straddle the line between art and programming; they must be experts at both. This makes the position difficult to fill. Technical artists need to understand the art pipeline and must be able to program well enough to create or improve the methods artists use to get their work into the game engine.

Engineering

Engineering roles include tools engineer, gameplay engineer, scripter, network engineer, graphics engineer, and audio engineer.

Tools Engineer

Tools engineers don't work on the game directly but instead focus on building the tools that all the other disciplines use to create the game. In recent years, many more engines have been built by independent third-party companies, reducing the need for tools engineers on game development teams.

Gameplay Engineer

Gameplay engineers are mostly focused on making the game work. They work closely with the system designers and all other designers to implement features needed to bring the game to life.

Scripter

Scripters use an internal scripting language to add polish and detail to the game. Scripters sometimes fall under the engineering team and sometimes under the design team, but in either case, they do basically the same job.

Network Engineer

Network engineers focus specifically on the connectivity of computers to each other and to servers. This is an immensely difficult role that requires specific training.

Graphics Engineer

As the name suggests, graphics engineers focus on the programming that supports the visuals of the game. Whether it's making water look more realistic, creating long vista views, working out how to achieve a good frame rate, or using a new shader to create an interesting effect, graphics engineers are responsible for the underlying code.

Audio Engineer

Audio engineers do the programming that powers all of the audio in the game.

Production

Production roles are a little different from the roles discussed so far in this chapter. Under the umbrella of production, there are many roles, including assistant producers, associate producer, and production assistant. There are many variations of the titles and responsibilities in production. To add to the confusion, different studios use a variety of titles for people in production. However, production roles fall roughly into three categories: management, coordinators, and assistants.

Management

Managers are in charge of money and often staffing.

Coordinators

Coordinators are involved in the day-to-day workings of the team. They often run meetings, keep track of tasks, and check up on bugs. They also facilitate communication within the team.

Assistants

Assistants are the lowest-level producers, and they do a lot of the daily chores to free up the senior-level employees. An assistant position is usually a starting position and provides excellent insight into how games get made.

Design

The designer roles include level designer, game system designer, data designer, scripter, and technical designer.

Level Designer

Level designers create the physical space that players traverse. Depending on the team, this may be laying out rough pathways, or it could be arranging preexisting environmental assets or even roughing out the world with geometry primitives. Level designers also often do scripting and other puzzle design for their levels. They may even contribute to individual story elements. In large, open-world games, a level designer might be called a scenario designer.

Game System Designer

A game system, simply put, is a set of rules that makes a game run. In any game, the rules are what really make it a game. Game system designers deal with these rules. It is their job to create the rules of the game, organize those rules so that they are all compatible with each other, and explain those rules to the player. A very simple old game is likely to have a simple system that can be explained with a page or so of rules. Modern games often involve thousands of data objects and tens of thousands of lines of computer code that function as the rules of the game. Ensuring that all this information is organized and logical—and making sure players can digest it—is the primary responsibility of the system designer. A system designer creates AI,

weapons, vehicles, and any other objects that are used throughout the game. System designers decide what attributes are assigned to the various objects and how all the systems in the game interact with each other.

Data Designer

Data designers populate the game world with characters, objects, weapons, vehicles, and every other game object. Whereas system designers decide what attributes each object will have, data designers decide the value of each of those attributes.

Scripter

As mentioned earlier in this chapter, scripters use an internal scripting language to add polish and detail to the game. Scripters sometimes fall under the engineering team and sometimes under the design team, but in either case, they do basically the same job.

Technical Designer

The term *technical designer* is currently used a couple ways in the game design industry:

- A technical designer may be an implementer who is strong at scripting/programming but does not do much of the documentation or the high-level creative design. This could potentially be a junior designer who is mostly taking direction from a senior designer.
- A technical designer may be a bridge between the engineering team and the design team, making tools or systems and using them to make the lives of the other designers easier. This would almost never be a junior designer as the skills required to do the job are simply too advanced and require too much experience.

Sound Team

The sound team is in charge of making the game sound amazing. While this seems fairly straightforward, it's one of the most difficult positions in the industry. Sound designers often need to wait until much of the rest of the

game is nearing completion before they can start their job. They have to account for every bit of audio feedback the player will need, which is often vast. They also often create soundtracks, and they are increasingly adding dynamic music based on the action in the game. These are difficult, technically challenging skills to learn. Members of the sound team must have background in sound effects and music, and they are also typically much more technically savvy than traditional musicians.

QA Team

The quality assurance (QA) team, also known as testers, is in charge of ensuring the quality of the game when it is released. There is a common misconception that members of the QA team are playtesters only. This is incorrect. Playtesters are volunteers who play the game as a typical end user would and give unquantified feedback on how the game made them feel, how much fun it was, and so on. The QA team does very often play the game, but it does a lot more: It tests the game. This means doing things like bumping into every wall in a level dozens of times and from every direction to make sure the player character can't slip out between gaps in geometry and flipping back and forth between menu pages for hours to ensure that graphics and code are being properly loaded and unloaded. An average player would never do these types of activities while playing a game, but the industry has standards set high enough that all fringe cases must be accounted for. Members of the QA team include those with diverse educational backgrounds. Positions on this team are often seen as stepping stones into other career paths, most notably production and design.

Narrative Designer

The narrative designer, sometimes called an author, is in charge of the game's story. There are typically a very small number of narrative designers in a studio, and very often the role is fulfilled by those on the senior design or production team. Narrative designers have training in creative and technical writing, as well as a solid game design foundation.

Additional Roles

From human resources, to game writers, to IT, there are many more people and positions that run in the background of the game industry, making the process work. These other roles may not get a lot of press or be the face of a game at a convention, but they are every bit as important as the person credited as having “created the game.”

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the roles in the game industry:

- For some of your favorite large studio games, find out what game developer roles were involved in game creation. Note how the scope of the game drastically affects the number of developers and the variety of roles on the team.
- Do some research on the websites of a few small, independent game studios and take note of what game developer roles were involved in game creation.

Chapter 3

Asking Questions

Asking questions seems like a topic we all know intuitively. After all, people ask questions all the time, so why bother learning how to do this task for game development? Game designers need to ask unique questions in unique ways, and they interpret the answers to questions in a unique way. This chapter looks at how to craft theoretical questions that yield meaningful answers. For example, “Is the game fun?” is not a quantifiable or answerable question because everyone has a different and personal interpretation of what fun is. This chapter also talks specifically about how to write technical questions that seek help with problems. For example, rather than ask the lead engineer “Why does the game engine crash when I load this asset?” you need to get more granular in order to reach the heart of the issue.

How to Ask a Theoretical Question

System designers often need to ask questions about games. Specifically, they ask questions about how the game feels, how the game is balanced, and how new data objects fit into an existing system. The main goal with all of these questions is to quantify feelings. To do that, it is important to take a somewhat scientific approach, and one good way to do that is to use the scientific method.

Steps of the Scientific Method

The scientific method, as applied to game development, proceeds through the following steps:

1. Define a question for playtesting.
2. Gather information and resources.

- 3.** Form an explanatory hypothesis.
- 4.** Test the hypothesis by performing an experiment and collecting data in a reproducible manner.
- 5.** Analyze the data.
- 6.** Interpret the data and draw conclusions that serve as a starting point for new hypotheses.
- 7.** Publish results.
- 8.** Retest.

Step 1: Define a Question for Playtesting

If you want to know if a game is fun for a playtester, what questions do you need to ask? You could simply conduct a survey and ask players if they enjoy the game, but would that give you usable answers? Research has shown that when people agree to complete a survey, they enter the process predisposed to answer positively. After all, they were excited enough about the project to be willing to take the survey. This predisposition is called *acquiescence bias*. So by posing a question through a survey, you are unlikely to get unbiased results. Another issue with this approach is that different people have different definitions of “fun,” and often these definitions run exactly counter to each other. For example, one person might find having “jump scares” in a game fun and exciting, whereas another may find jump scares to be stressful and not fun at all. So asking each of these players if a game centered around “jump scares” is fun would yield contrary results. To get meaningful results, you need to figure out how to define the feelings you want players to feel. Then you need to figure out how you can define those feelings in metrics that you can measure.

It’s important to avoid asking questions that are not focused enough. Asking something as broad as “Is the whole game fun?” or “Is the entire game balanced?” or “Is the game too easy?” will not give you actionable results. Parts of the game might be fun, and others might be boring or frustrating.

Asking overarching questions of large breadth leaves too much room for interpretation. When asking a question about a game, think about how focused you can make the question—for example, “Is this individual enemy character too challenging for our target player?”

Once you have narrowed your question down to the smallest unit that is practical, the next step is to quantify the question so that you can get a measurable answer. At first, it may seem that it’s impossible to do this with a lot of questions. Emotions like fun, exciting, scary, and rewarding can’t be directly measured. However, you can try to backward engineer a feeling based on observed data. For example, what metrics could you observe in a player who is having fun versus in a player who is bored? Play time might be a good indicator. Players who are having fun might play longer sessions, or play through a higher percentage of the game, or replay the game more often, or have a higher total time played—or maybe even all of the above. Some players find failure frustrating, so you could also track the number or ratio of failures at a given task. For example, you could say that a task is “frustrating” if the player fails at it more than 70% of the time. That is a perfectly quantified and measurable metric.

The following sections look at two more examples of transforming vague, broad questions into useful, measurable questions.

Example 1

Original question: Is level 3 exciting enough?

Quantification: You want the level to go by fast, and the designer is saying he designed it to be around 1:45 long. If players are in the level longer than 2 minutes, they are likely missing something, meaning that frustration or boredom could soon set in. However, you don’t want the player breezing through because then there would be no excitement, so you want to make sure players are failing at the level and restarting it at least twice before they learn the required skills to succeed and pass on to the next level. Given these factors, you can’t find out whether level 3 is exciting enough for all players. Some players will have a tougher time than others, but you do want this level to be fairly controlled. You want to target your preferred metrics

for a fairly high majority; for example, you might decide that an 80% success rate indicates that you have achieved your goal.

Revised question: Can 80% of players complete level 3 in under 2 minutes, while also restarting the level at least two times but not more than five times?

The revised question is now completely quantified. While at first glance it would not appear that this question yields an answer about whether the game is exciting, through your testing, you determine that it can. You might also determine through testing that your presumed definition of exciting is incorrect. If this happens, you can revise your quantified definition.

Example 2

Original question: Is this boss battle difficult enough?

Quantification: This is a late-game boss fight, so it should have a small probability of success on the first encounter—such as 15%. The player should be using most of their acquired weapons to beat the boss; let's say 70% of weapon variations have been triggered at least once during the encounter. The battle should take a while; for example, you might want players to spend at least 5 minutes in the battle before the boss is defeated. Keep in mind that you can adjust all these numbers at a later time if you find that they don't actually represent the feeling you want.

Revised question: Do 85% of players fail the boss battle at least one time? When they are successful at the battle, do players use at least 70% of weapons available and take at least 5 minutes to complete the encounter?

Determining What Numbers to Use When Quantifying a Question

How do you know what numbers to use to quantify a question like this? Sometimes, you already have an idea for a metric in your head before you even start. For example, you might think that a challenge that is “easy” should not be failed more than 50% of the time. Other times, you don’t know what metric to use. In such a case, if you already have a game to test, you can observe testers anecdotally and watch what metrics correlate with

the testers' feelings. For example , if you observe a tester getting bored with a level, you want to note the time at which the player got bored. You could also observe any other metric that correlates with the tester's current observed attitude. If you don't have anything to go on, you will have to make a few guesses as you define the question and assume that you will need to adjust the numbers as you go forward with testing. At this point, it is important to do testing in which you watch and observe players while they play the relevant portions of the game.

Note that on a small scale or with individual playtesting, you can still use some subjective questions with your testers, like "How fun is the game?" You just need to acknowledge that the results you get from such a question likely won't have directionally actionable metrics. However, you can use such questions to start conversations with playtesters and eventually move toward gathering less subjective data.

Step 2: Gather Information and Resources

Once you have come up with quantifiable questions, the next step is to make observations of your players and gather information. Just as in any other field, there are several ways to do this right, and there are an enormous number of ways to do it wrong. Setting up a test incorrectly can result in ambiguous results or, even worse, misleading results. (You'll learn more about conducting tests on data in [Chapter 17, "Fine-Tuning Balance, Testing, and Problem Solving."](#))

Step 3: Form an Explanatory Hypothesis

In this step, you need to start tying the quantified metrics you've recorded to the feelings and behaviors you observe in testers. By collecting data, you can get a much clearer definition of what characteristics such as *challenging*, *boring*, and *fun* mean for your specific game.

Step 4: Test the Hypothesis by Performing an Experiment

Once you feel that you understand how to quantify the desired emotions from players, the next step is to try to purposefully manipulate those emotions. For example, if you determine that a failure rate of greater than

70% would be frustrating, then test that. Start by performing a control test with your current data to get a baseline reaction. Next, tune the game so that testers are failing more than 70% of the time and then ask them (or observe) whether they are getting frustrated. Then do the converse and reduce the difficulty to ensure that it has a correlated effect. Through experimentation like this, it is possible to dial in on tuning to get accurate adjustments not only in numbers but also in feelings.

Step 5: Analyze the Data

This step, which needs to occur regularly throughout the process of testing, should be done using a database or a spreadsheet. This process is covered in detail in [Chapter 15, “Analyzing Game Data.”](#)

Steps 6 and 7: Interpret the Data, Draw Conclusions, and Publish Results

Once you have tested, tuned, analyzed, and begun to understand the data for your game, it’s time to write it all down. Ideally, you should be keeping notes throughout the entire process. It is critical to write a guide when you understand what is going in order to explain the process to teammates and your future self.

Step 8: Retest

Game testing never stops. Often games remain in a testing phase and are balanced long after the game is completed and shipped out to players.

Defining a Question for Data Analysis

When you are trying to convert players’ feelings into metrics, you often have to do some guesswork and interpretation on what those feelings mean. Sometimes you need to dig through existing data and answer some questions, such as “What is the best weapon?” or “Which is the toughest enemy character?” or “How many shields could this sword defeat?” These kinds of questions don’t have anything to do with the feelings of a player and often don’t require any tester observation. With this type of information, you can use a spreadsheet or other calculations to determine

the equation mathematically. When you ask questions like these, it is important to quantify each and every aspect of the question so that you can then write a formula to find the desired result.

The following sections look at two examples of defining questions for data analysis.

Example 1

In the game of backgammon, rolling doubles gives the player more movement. Say that you want to know the chances of a player getting the extra movement. To quantify this, you would use the question “What is the player’s percentage chance of rolling doubles on two six-sided dice?”

Example 2

Say that your game has a gun that can fire a laser round 100 meters, and it loses no ability

to cause damage over that entire range. You have another gun that fires bullets up to

60 meters. The bullets do more damage than lasers at point blank, but the damage the bullets can do decreases every meter they travel. To find quantitative results, you could ask, “Which is the best gun to use, from the perspective of damage, at 40 meters?” This again, would require no playtesting but would instead be done directly in a spreadsheet.

Note

Methods for answering questions such as the ones posed in these examples are discussed in [Chapter 15](#). At the early stage of development, it’s more important to get comfortable asking questions than it is to find immediate answers.

How to Ask for Help with a Problem

Asking for help with a problem requires very different skills from asking a data-driven question. As discussed earlier in this chapter, you answer data

questions with data results, using multiple opinions, analysis, and testing. With theoretical questions, it is likely that no one currently knows the answers, and you discover information as you pursue answers. Technical questions are much more straightforward than data-driven questions. You ask a technical question when you are unsure of something but think someone else might know the answer already. Hopefully, someone already knows the information, and they just need to know that you need it and give it to you. While this sounds straightforward, it's also easy to get it wrong, so this section examines one method you can use to increase your likelihood of getting information from someone else.

Why How You Ask Matters

In the game industry, there are constant deadlines and generally more work to go around than people or time to do it. The more responsibility a person has, generally, the more pressed for time that person is. Leads, producers, executives, and other people with big titles often have very tight schedules. Sadly, these are also the people who most often have the answers to questions you have. To complicate the situation further, the game industry is filled with many intense people who thrive under harsh deadlines and rigorous work schedules. These attributes make great workers—but not always the most patient or friendly people. A very busy, intense person might see a poorly written question as a time waster and become aggravated. Nearly every game designer in the industry would be able to tell you a story about a lead engineer giving them an earful in response to a question.

Questions that are well written, well researched, and well organized are more likely to get the attention and answers you need. Questions that are poorly written are less likely to get answered, or taken seriously, and they are much more likely to be dismissed or not answered in a constructive way. This does not mean you should keep your questions to yourself. Asking questions when you need help is vitally important. When you find that you need help, take the time and care to write a solid question, and you will have a better chance at getting the information you need.

Steps to Writing a Good Question

There are a few important steps involved in writing a good question:

1. Do your due diligence and write about the issue. When you're clear on the issue, write down your question. Try to answer your own question for 10 intense minutes. Do Internet searches, read manuals, dig through similar examples, and see if you can figure out the answer for yourself. Let the recipient of your question know the sources you have already exhausted.
2. Write reproduction steps. What exactly did you do, step by step, and how exactly did it go wrong?

Tip

“What went wrong” is never “everything” or “something”.

3. Write down attempted fixes. The person you are asking doesn't know what you have or have not tried already, so list all the things you have already tried.
4. Include a thank you with your question.
5. Before sending it, read your question as if it were sent to you and you had no knowledge of the issue until now. Can you understand it? Is all the important information there? If so, go ahead and send the question.
6. When you get an answer, assume that the answer is right, even if it seems odd.
7. Follow the answer you get *exactly*.
8. If the answer does not work, go through all these steps again and follow up. It is important to do this to prevent the person answering the question from falsely thinking that the problem has been solved by a potentially wrong answer.

As you carry out these steps, consider these additional tips:

- **Get to the point.** There is plenty of other time to chat. When you need a question answered, get right to the point without a lot of fluff.
- You may be very frustrated and angry when writing a question but remember that you are asking someone to help you. Put in extra effort to **be nice**. Also, don't act entitled or owed by the recipient, even if you are. Doing so can cause a hostile exchange, which does not set you up for getting the answers you need.
- Tell the recipient what you are trying to do and the **exact problem** you are trying to solve—not just a symptom of the problem. For example, say that you are trying to calculate the average time it takes a player to complete a game level, but each time you type your formula in Excel, the whole workbook crashes. Saying “Excel is crashing on me. Do you know why?” is not nearly as useful as saying “I am trying to calculate average player time in my level, but my formula is crashing Excel. Attached is a copy of the workbook, and this is the formula....” Knowing what you are actually attempting to do is vital information for the reader. In this example, the answer to your question might be to not use Excel because it is not the right tool for the job. But this answer can only be given in answer to a question that is trying to solve the actual problem and not a symptom.
- Be **very specific** and use detail. Consider the following:
 - When writing a technical or other difficult question, avoid using pronouns in order to eliminate one possible point of ambiguity. For instance, when you say something like, “The hero barges in on the bad guy, then he dies,” how does the reader know which one died? Instead, try “The hero barges in on the bad guy, and the hero dies.” This leaves no question about what exactly happened.
 - It is inadvisable to use words like *stuff*, *things*, *whatever*, *junk*, *you know*, *everything*, and *crazy*.
 - “The player character” and “the player” are different. This distinction is unique to the game industry, but it is very important. The player sits in a chair and pushes buttons. The player character is the in-game representative of the player. For example, “The

player character can't see the objective" is a different problem than "the player can't see the objective."

- **Don't pretend** you know information that you don't. You may feel embarrassed about having to ask a question when you know that you should already know part of the information. It's time to come clean. Admit your lack of knowledge and just ask the question. It's not fun to do this, but it is much better than not getting the information you need.
- Use **bold wisely**. By highlighting a few key words, you can help direct a skimmer to the most important parts of your question. Don't go overboard with bold, or you will make the question look noisy and like you are screaming. No one likes to have questions screamed at them. Almost never use all caps, which tend to look like shouting. (Go back through this list and just look at the bold words. Just based on these words, you can quickly get an idea of what is being communicated.)
- It can help to let someone know **why** you are asking them. Prefacing a question with "I know you have years of experience with C#, so I thought I would ask you how to get this script working" gives the recipient more information on what you are hoping to get. If the reason is obvious, however, it is not necessary to explain why.

Good Question Example

Take a look at the following example of an email that demonstrates the preceding advice for writing a good technical question:

Subject: Trouble adding a new row to the Time Logging Excel
Spreadsheet

Hello, *Developer*,

My team and I have not been able to get the time logging spreadsheet to work properly when trying to enter new hours.

Here are the reproduction steps:

1. Open the spreadsheet in Excel 2010 on a MacBook Pro running OS X Version 10.7.5, select row 56 on Tab 1.

2. Right-click the selected row and try to select Insert New Row.

Issue: The Insert New Row option is gray, and I can't select it.

I have tried looking through the Excel help files for “Insert new row,” and I have done a search on the Internet using the search term “Excel gray select new row.” I read the first five search results, but none of them showed a solution for my specific problem.

I tried closing Excel and opening it again. I tried creating a blank workbook, and I could use the Insert New Row option in the new workbook.

Is there a way for me to insert a new row 56 into my workbook, or is there another workaround?

Thanks,

My Name

My Game Title, My Team Name

My Contact Info

Bad Question Examples

The following are some examples of bad questions:

- Everything in my level is broken. Can you fix it?

This question provides no information and makes the asker sound helpless. This kind of question makes a lead engineer weep a single tear.

- “Something is wrong with my .png file, but I don’t know what. What should I do?”

This question provides too little detail. You need to tell the recipient what is wrong with the file. Does it fail to open with Photoshop CS6? Does it crash the computer when you attempt to open it with Microsoft Paint? Do the contents of the file not match your expectation? Is it something else?

- “I did what you said, but now things are different. How do I get them back?”

There are several problems here. This question looks like it is trying to shift blame onto the person who answered it. That's not polite or constructive. The question is also lacking in specifics. What is different? Is it different in a bad way? How so?

Here is a better way to ask this question:

Subject: Class project Level 10 is crashing on load

As you know, I have been working with Unity to create a level with my team. After adding a new static mesh last night, my level has been crashing.

Here are the steps I have gone through:

1. Open Unity and load my level on Windows 10 64-bit machine.
2. Select the option to play my game.

Issue: Unity crashes with no debug output or warning. The program closes completely and immediately.

I have tried deleting the static mesh that I added last night. I have tried closing and reopening Unity. I am now frustrated and confused. Can you suggest steps for me to take to debug this issue?

Thanks,

My Name

My Game Title, My Team Name

My Contact Info

When someone's answer to a question you have asked really is your problem, say a short but polite “Thank you” either in person or via email. They need to know if it worked just as much as they need to know it didn't work, and closing the communication loop is important to show you can be trusted as an ally in finding and solving problems.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to practice asking questions:

- Write some data-driven questions for a simple game you have made or played and then answer them. Practicing this skill takes a surprisingly large number of iterations. As you write the questions, focus on how others might misread them. You can even go so far as to try to think of all the ways the question could be misinterpreted.
- Take extra time to think about writing questions in your everyday life, as you would as a game designer. One benefit of living in a world that is full of opportunity for asking questions is that you can use these opportunities to practice. Posting questions on technical forums is a great way to get feedback from strangers and see if you are communicating your questions clearly.
- Browse through forums for examples of bad questions. This should be very easy as public Internet forums are bursting at the seams with bad questions. With subjects you understand, try to respond to questions in ways that help clarify the questions. Not only will this give you good practice writing questions, but it will also give you a chance to try to help others.

Chapter 4

System Design Tools

The game industry is, as you would expect, full of computer software tools. The needs of each team, project, and game are different, and a variety of tools are available to meet these needs. Some tools are used more often than others. This chapter discusses some of the most common categories of tools used in the game industry after taking a close look at exactly what the term *data* refers to.

What Is Data?

Throughout this and following sections, you will often see references to tools and concepts that deal with data. But what exactly is data? For the sake of this book, *data* is numbers or text used to describe something. Importantly, data is acted upon; it does not take action. Let's look at a couple of examples.

Say that a hero character has an attribute called health that has rules associated with it. Combat rules determine the amount of health that is lost during an attack or gained during healing. The hero's health score is currently 100. In this case, 100 is data. It is a number attached to the health attribute that is used by the attribute and the rules to do something. In [Table 4.1](#) you can see several more examples of data that are attached to different attributes and rules.

Table 4.1 Examples of Data Attached to Attributes and Rules

Rule/Attribute	Data
Health	100
Weapon	Sword

Strength	50
Fuel capacity	20
Engine type	Gasoline
Is alive	True
Eye color	Green
Number of dice to roll	2

Notice that some of the data in [Table 4.1](#) is numeric, some is binary (true or false), and some is text. Regardless of type, all the data in this table is used by rules in the game to do something. However, the data itself does not contain any rules.

Game Industry Tools

Game industry tools span a number of types of software and applications, including documentation tools, image editing tools, 3D modeling tools, flowcharting tools, databases, bug-tracking software, game engines, and spreadsheets. The following sections examine each of these tool types in detail.

Documentation Tools

Word processors are some of the oldest and most commonly used computer tools, and game developers use them constantly. Creating and maintaining documentation is the primary use for word processors. Generally speaking, each feature, level, concept, major game character, or system needs to be proposed. Once it has been proposed, or “pitched,” it is discussed, clarified, and organized. All of these initial design tasks can be done quickly and safely in a word processor without fear of introducing errors into the game. Developers use a game design document (GDD) to record most or all of the concepts in the game. A GDD can be a very short document meant to keep just the core ideas, or it can be absolutely mammoth and go on for hundreds of pages. In some large teams, there is even a developer whose sole task is to organize and maintain the GDD.

Microsoft Word, Google Docs (see [Figure 4.1](#)), and Apache OpenOffice Writer are examples of popular documentation tools.

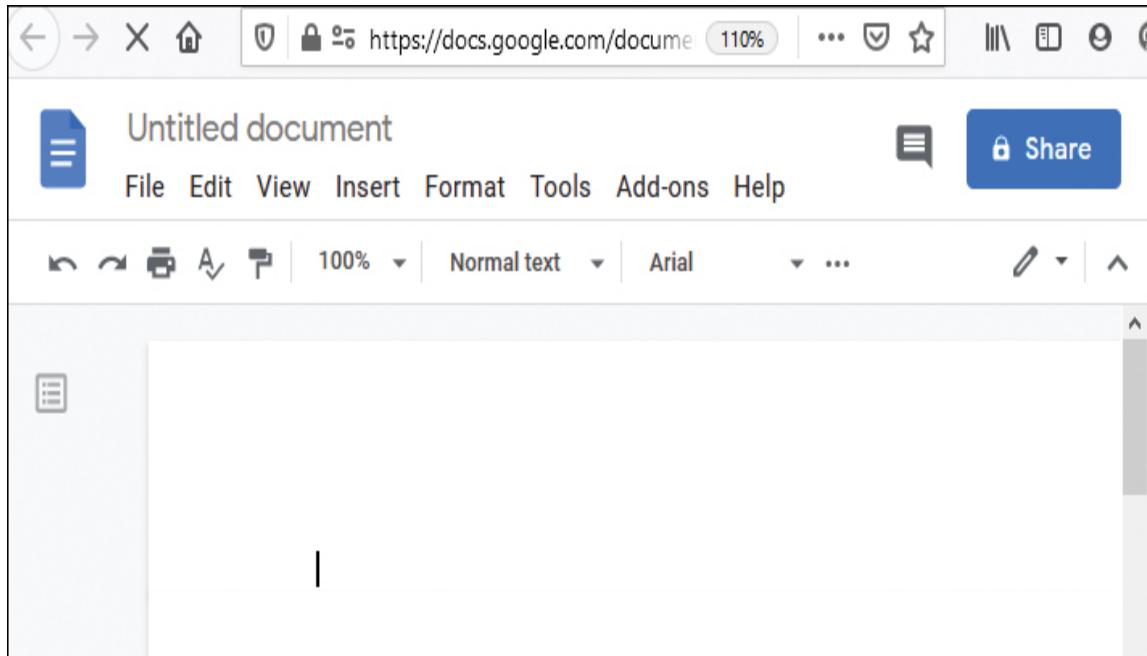


Figure 4.1 Google Docs

Screenshot of Google Docs ©2021 Google

Image Editing Tools

Everything that would have been painted or drawn in a traditional art setting is now created or modified using an image editing tool. These tools can import raw images from a camera or scanner, or they can be used to create entirely new images from scratch. Modern image editing tools are incredibly powerful and allow artists to quickly create and manipulate 2D images. Game developers use image editing tools for a myriad of tasks including creating UIs, creating textures on 3D objects, creating the skins on character models, creating concept art, and much more. Even though image editing tools are primarily the domain of artists, all developers, including designers, benefit from learning the basics of image editing.

Adobe Photoshop (see [Figure 4.2](#)) and GIMP are examples of popular image editing tools.

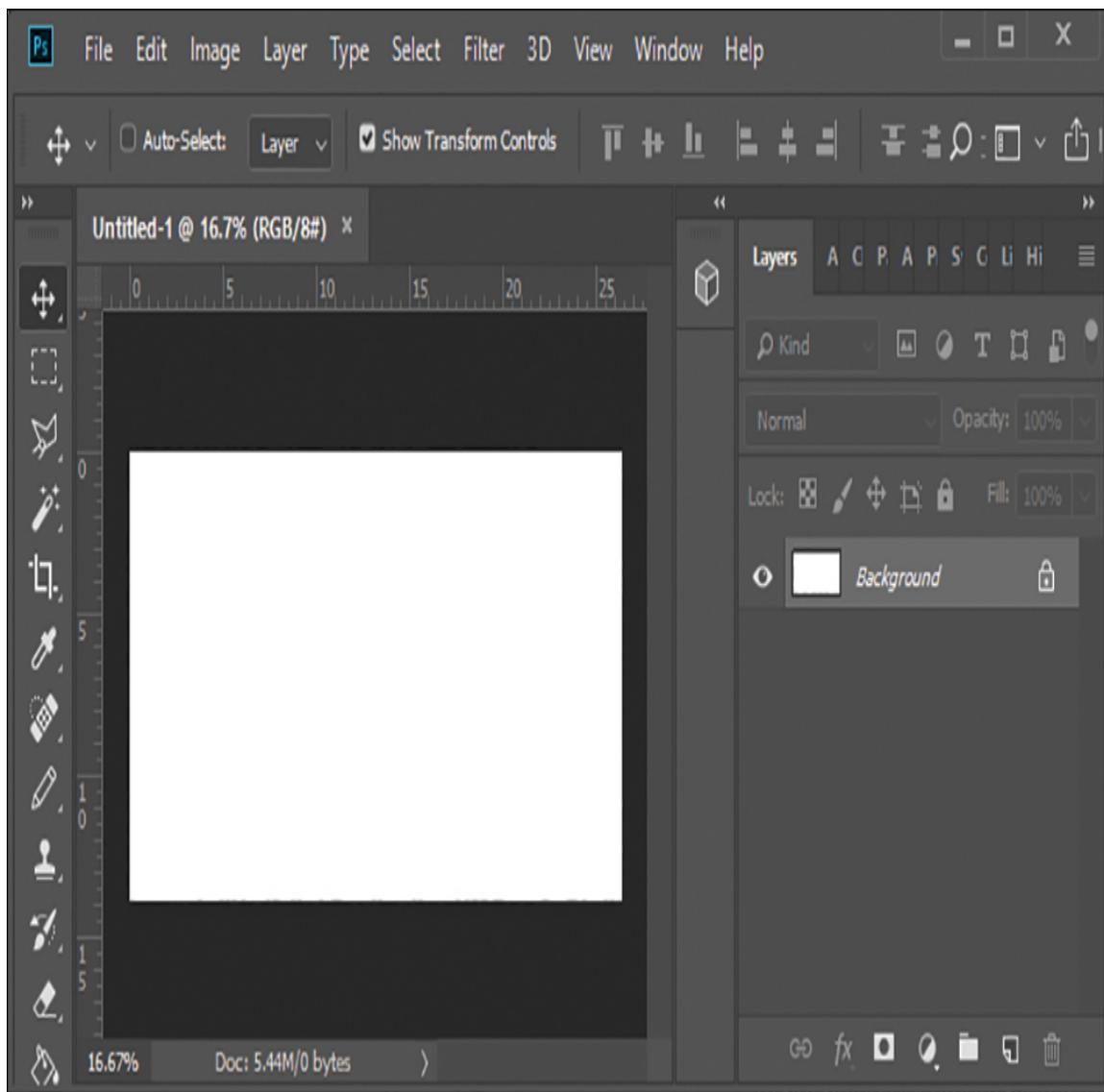


Figure 4.2 Adobe Photoshop
Screenshot of Adobe Photoshop © 2021 Adobe

3D Modeling Tools

In a 3D video game, a player can see and interact with 3D objects. Artists and designers use 3D modeling tools to make those objects. A 3D modeling tool has an interface that allows users to create 3D objects and manipulate them in simulated 3D space. Usually, a 3D modeling tool has multiple camera views that allow a user to see an object from multiple perspectives at the same time. These tools also have controls for manipulating 3D

objects to do things like stretch, carve, twist, resize, divide, and attach 3D shapes. Most of these functions have very specific names in 3D modeling tools, and having a working knowledge of the jargon for the tools is highly beneficial.

Once 3D objects are created, most tools can aid the developer in making the objects textured. In games, this does not mean exactly the same thing as it does in the real world. In the real world, a texture is something you can feel. In the game world, a texture is simulated with an image. A 3D modeling tool can import an image made in an image editing tool and apply that image to the surface of a 3D object, giving it the illusion of having a textured surface.

Many 3D modeling tools can also be used to animate 3D objects. A game developer can use animation for prerendered cinematic scenes or export animations as instructions for a game engine to use in real time.

The final task of a 3D modeling tool is to export an object in a file format that the game engine can understand.

3D Studio Max (see [Figure 4.3](#)), Maya, and Blender are examples of popular 3D modeling tools.

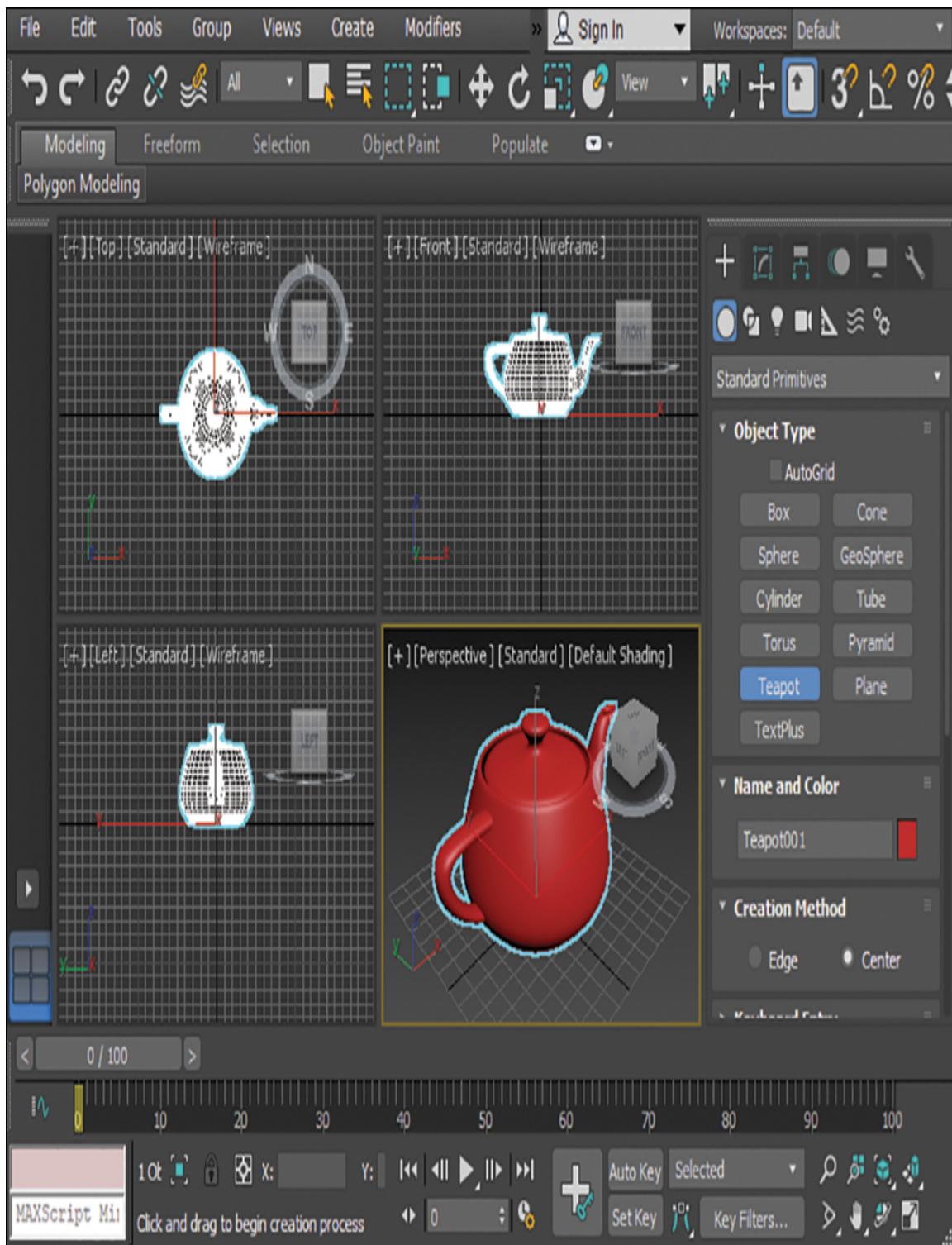


Figure 4.3 3D Studio Max

Screenshot of 3D Studio Max © 2021 Autodesk Inc.

Flowchart Tools

A flowchart, as illustrated in [Figure 4.4](#), is used to organize ideas and show progression through a system. Unlike written text or a linear list, a flowchart can graphically display branching, loops, decision points, and terminus conditions. Very often, level designers use flowcharts to rough out the concepts of a level before they commit to the time it takes to make the level into a map or 3D model. System designers and engineers often use flowcharts to show movement through a system and to rough out concepts quickly before committing the time to create the system in the game engine.

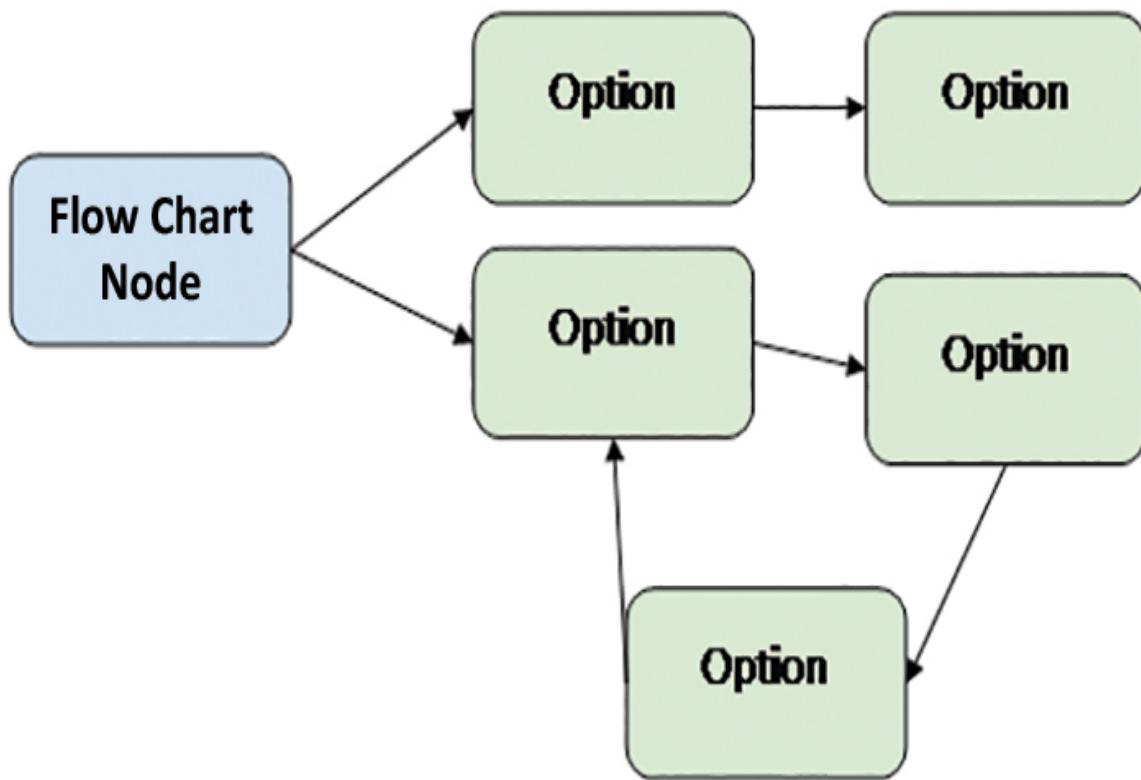


Figure 4.4 A flowchart

Google Drawings, Visio, and OmniGraffle are examples of popular flowcharting tools.

Databases

A database stores large amounts of data to be retrieved and used later on. A simple game may not need a database, but for a very large game, the amount of data that is stored can be monumental, and storing it in engine or in script can be unwieldy. System designers, in particular, need to be familiar with how to use a database as they are often the ones who manage game databases. Databases are commonly used to store weapons, armor, vehicles, power-ups, player-consumable items, AI characters, and other enemy character types. A database of objects acts as a central repository for game objects that can be used and reused, and it keeps those objects consistent across the entire game. Imagine how hard it would be to sort through all the scripts written for an MMO, looking for individual items and characters. A good analogy for a database is a file cabinet. You can go into the file cabinet, find the object you are looking for, and then either reference it for use in the game or make modifications in a central place that will affect the entire game.

A database management system provides the tools you use to actually access all the data in a database. To understand the difference between a database and a database management system, think about storing and using your music collection. On your hard drive, you might have hundreds or thousands of songs by different bands, in different genres. In this case, your hard drive is like a database, holding all your music information, including the songs themselves. In order to look at that information in a meaningful way, you could just browse through Windows file folders, but doing so would be very slow and inefficient, and you would likely miss a lot of important information. Instead, you are likely to use a music playing app, which is, essentially, a specialized database management system. You can use the app to filter for genre, or find every song by a band, or find the entire contents of a single album, or otherwise access the data. In the game development world, database management systems do a job similar to your music playing app. If you were working on a large RPG, you might want to look up every sword, or just shields, or just level 22 items that are used by knights. The database stores all the weapons, armor, and character class definitions, and the database management system allows you to access just the information you want.

Couldn't a spreadsheet handle the same tasks as a database management system? Yes, it could, and in many smaller games it does. But spreadsheets

load all the data stored in them every time they are used. If you are accessing a very large amount of data, the spreadsheet will be slow. In addition, if you are only loading the items you want to edit, there is less chance of introducing errors into objects you did not intend to touch.

Microsoft Access is an example of a popular database tool, and MySQL and Microsoft SQL Server are examples of database management systems.

Bug-Tracking Software

Bug-tracking software is a specific type of database management system that is used specifically to track bugs in software. The database holds information about the bugs in the game. The management system is set up to specifically deal with the kinds of information and tasks you need to do with bugs. Unlike a generic database management system, bug-tracking software does not need to be manually configured before it is ready to be used to track bugs. You can use bug-tracking software to record new bugs, search for similar or duplicate bugs, assign a bug to a different owner, track the progression or fixes to a bug, and filter out fixed bugs.

Jira, Bugzilla, Plutora, and Backlog are examples of popular bug-tracking software.

Game Engines

A game engine is where an entire game comes together. Many of the challenges faced when making a video game are the same regardless of the specific game. Examples include rendering the graphics on the screen, detecting when objects have collided with each other, and retrieving art assets from memory. Most games need such tasks accomplished, and it would not make sense to program the functionality from scratch for each game that is made. Therefore, game teams, studios, and dedicated companies package all the common functionality needed to make most parts of the game together into a program called a game engine.

In addition to taking care of the basic functions of a game, a game engine organizes the interface for the game so that it is easy to use. For example, many game engines have a 3D view so that developers can move through 3D space to see where in the game level objects are located and adjust them

or place more of them. Game engines also commonly have file management integrated so that artists, designers, and programmers can work with each other to make assets come to life in the game.

Several companies today make “middleware” engines, which are not made for any specific game or team but are instead designed to be used by anyone who needs a simple solution to game development. The upside of middleware is that it is designed to be used by as many teams as possible, so it is very flexible. The downside is that it will always be missing key components that make each game unique.

Unity 3D, Unreal Engine, CryEngine, and Cocos are examples of popular game engines.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further investigate game design tools:

- Do an Internet search for “Game engines for beginners.” You will find articles on a large array of modern game engines and introduction videos for most of the popular engines. Take some time to watch these videos and compare the features of the different game engines to get a better sense of where the game industry is going.
- Practice using documentation tools, which have many more features than you might realize. Game designers must be comfortable using these tools, not only to type out the words but also to properly format and organize documents. Microsoft Word and Google Docs are the two most popular documentation tools.
- Make a flowchart for a game—either a game you want to make or a simple game that already exists. Use the flowchart to show the flow of the game from the beginning and through various player decisions. To get some ideas about what the flowchart needs to include, do an Internet search for “How to make flowcharts for game designers” and spend time with tutorials on the using flowcharts to design games.

Chapter 5

Spreadsheet Basics

[Chapter 4](#) covers tools commonly used by game developers roles, but there is one tool that is the primary domain of the system/data designer: the spreadsheet. Because the spreadsheet is the primary tool you are likely to use as you design game systems, this chapter and the next are devoted to the use of spreadsheets. The spreadsheet examples in this book show Google Sheets (see [Figure 5.1](#)), which is a free and very powerful spreadsheet program. Several other spreadsheet programs are also well made and powerful. Almost everything covered in this chapter is also applicable to most other spreadsheet programs, though the interfaces may look somewhat different.

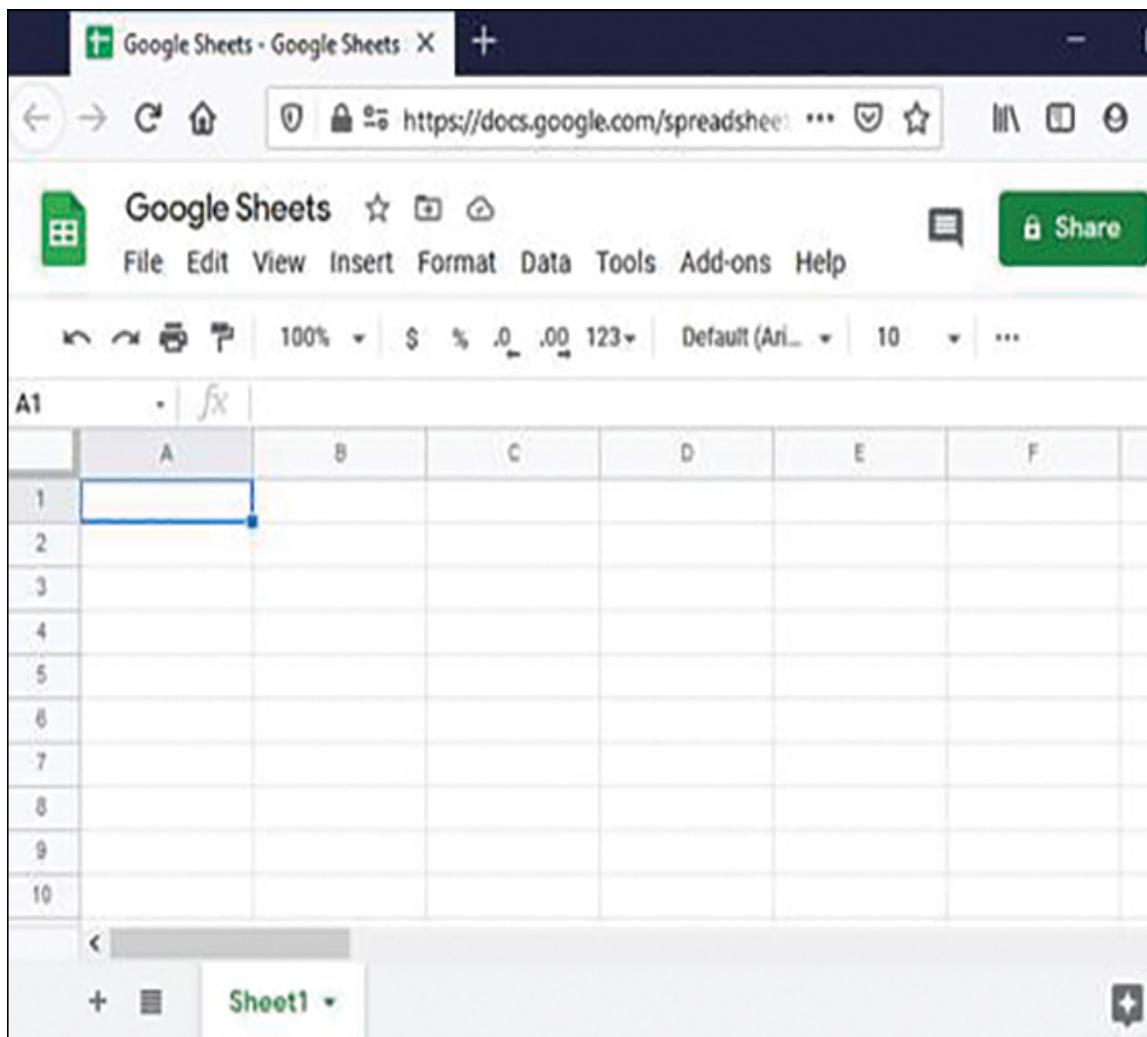


Figure 5.1 Google Sheets

Screenshot of a Google Sheet ©2021 Google.

Why Spreadsheets?

Imagine that your job involves moving ridiculously heavy things all day long every day. It is an awful, painful, and slow job. Then you learn of a giant who loves moving things and can move very heavy things all day long without complaint. It seems like your job is about to become easy quite quickly. However, the giant does not speak your language but uses his own Giantish language, and you don't know it. If you learn to speak his language, he will happily do whatever you want, and he will be able to easily do more than any human ever could. Once you learn to speak his

language, you need only tell the giant what to do, and then you can sit back in the shade and watch him happily work.

In this analogy, the giant is a spreadsheet, the language is spreadsheet functions, and the heavy things are math. If you learn to speak the language of spreadsheets, you can easily perform tasks that are either impossible or completely impractical for any human to do manually. The job of a system designer is often to calculate massive amounts of data through complex formulas, but it would simply not be possible to do this manually. A spreadsheet is a perfect tool for a system designer.

Additionally, spreadsheet skills are some of the most common and transferable skills in the industry. All the other tools discussed so far in this book are in a fairly constant state of change. New game engines, art products, and team tools are added to the market regularly. In addition, many very old products are still under constant development and change. Even the mainstay Photoshop has undergone such drastic reimagining that the product is almost unrecognizable from its original form. So even if you learned older tools years ago, if you haven't used them lately, you would need to ramp up again if you wanted to use their modern incarnations.

As all the tools in the industry mature and grow, the spreadsheet remains remarkably unchanged. A stable and useful spreadsheet design popped up very early and has stuck around since then. For example, [Figure 5.2](#) shows VisiCalc, which is the very first program that is considered a fully functioning consumer spreadsheet.

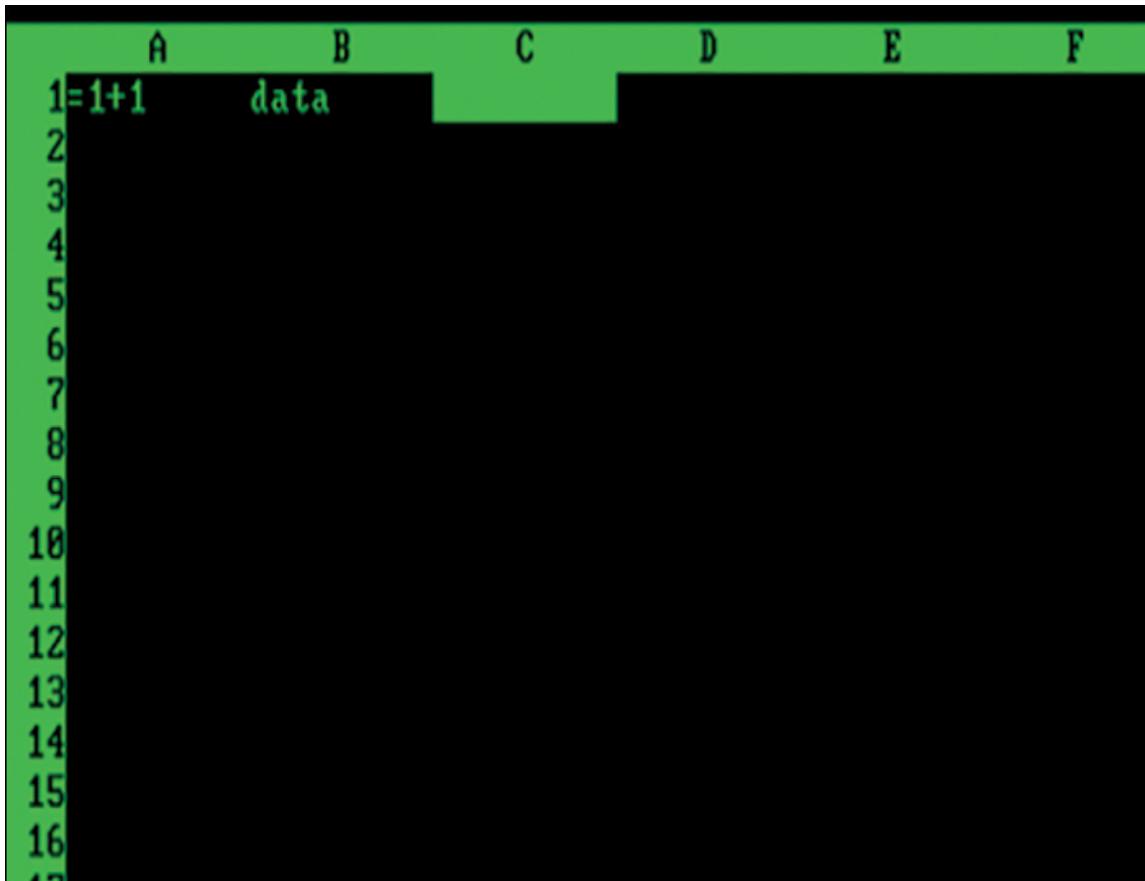


Figure 5.2 VisiCalc

Screenshot of VisiCalc © Copyright 1999-2018 by Daniel Bricklin.

Even though VisiCalc was released in 1979, you can pretty easily see all the same basic components that are present in a modern spreadsheet. Over the years, spreadsheet programs have added a lot of new functionality, but even a spreadsheet built in the late 1990s would have the vast majority of functionality available in a modern program. As spreadsheets have matured, the rate of change has actually slowed, whereas with many other tools the rate of change has increased. Learning to use spreadsheets is likely to be one of the most long-term game tool investments you can make. Popular examples of spreadsheet tools include Microsoft Excel, Google Sheets, and OpenOffice.org Calc.

Whereas game engines, 3D modeling tools, and many other game development tools tend to be pretty unique and require the user to learn

specific skills, modern spreadsheet programs are very similar in usage and functionality. If you learn one, it is trivially easy to pick up any other spreadsheet tool. It is also possible and even fairly easy to convert data files from each of the programs to the others. This is another unique benefit of a spreadsheet. Most other game design tools have specific file formats that are difficult or impossible to transfer from tool to tool.

What Is a Spreadsheet?

Basically, a *spreadsheet* is a group of data containers that can communicate with each other. We use spreadsheets to gather and contain data. We also use spreadsheets to organize, manipulate, and calculate this data so that we can find meaningful information. From a practical standpoint, this means we take a whole bunch of raw data that is hard or impossible to understand, and we let a spreadsheet do the heavy work of cleaning up the data and answering questions we have about the data. We can then further use the tool to manipulate that data individually or en masse to make it behave the way we want it to.

Spreadsheet Cells: The Building Blocks of Data

This section discusses the common elements of all the spreadsheet tools. Each tool may have additional features unique to that program, but once you know the common elements, it is fairly easy to learn the unique ones. As you read through the following sections, it would be very helpful to open a new spreadsheet and follow along in it, re-creating the examples provided. If you don't have a spreadsheet program already, you can get the free Google Sheets program, which is available on any modern web browser.

Note

This chapter explores spreadsheets as they pertain to game system design. There are many great resources in other books and online that cover spreadsheets in much greater depth for other disciplines.

Cells

The smallest unit of data container in a spreadsheet is called a *cell*. When you first open a spreadsheet and look at the grid, what you are seeing is a series of cells. The cell is the foundation on which all spreadsheets are built. Cells can hold data, retrieve data, calculate data, or perform more complex functions. Cells can also communicate with each other to perform even more complex tasks. Each individual cell can also contain many different types of information. The following sections examine the most common aspects of a cell.

Cell Address

A spreadsheet uses *cell addresses* to organize and refer to individual cells. Every cell, even an empty one, has an address that can't be removed or modified. A cell address is based on the intersection of the column and row where the cell resides; it consists of a letter (for the column) and a number (for the row). Both numbers and letters start at the top left of the sheet and increase by one for every cell they move from the top left. For example, the top-left cell in a spreadsheet has the address A1. The cell directly to the right of that one is cell B1, and the cell directly below A1 is A2. [Figure 5.3](#) shows cell B2 highlighted and selected.

	A	B	C
1			
2			
3			

Figure 5.3 Cell B2

By knowing the address of a cell, you can access the contents of that cell. This is one of the most valuable aspects of a spreadsheet and what really distinguishes it from a calculator. By using the cell address to reference information in another cell, you can perform not one but a series of calculations, building in complexity, while leaving the original cell data unchanged.

Cell Value

If you select any cell in a spreadsheet, you will notice a cursor flashing in it. When you see the flashing cursor, you can type pretty much anything you want into the cell. You can store basic text, numbers, and many other forms of data in a cell. For example, if you were to type HELLO in a cell and then press Enter, the value of that cell would become HELLO. You could, for example, store a list of items by typing a different item into each of a series of cells to create a list.

A cell value can be any kind of data, and it specifically refers to what you see in the cell after you press Enter. It is important to know that what you see in a cell and what you type in the cell may be different. What you see in the cell is always referred to as the *value*.

Cell Formula

Whereas a cell value can be seen as a noun, a cell formula is more like a verb. A cell formula is a calculation you enter in a spreadsheet when you want the spreadsheet to calculate an answer (which it displays as a value).

The Formula Bar

Directly above the grid of columns and rows in a spreadsheet is the formula bar. You can spot it by the distinctive *fx* symbol, which is called the *function symbol* (see [Figure 5.4](#)). The formula bar shows what is happening inside the cell, regardless of what the cell is displaying. You can enter information directly into a cell or inside the formula bar.

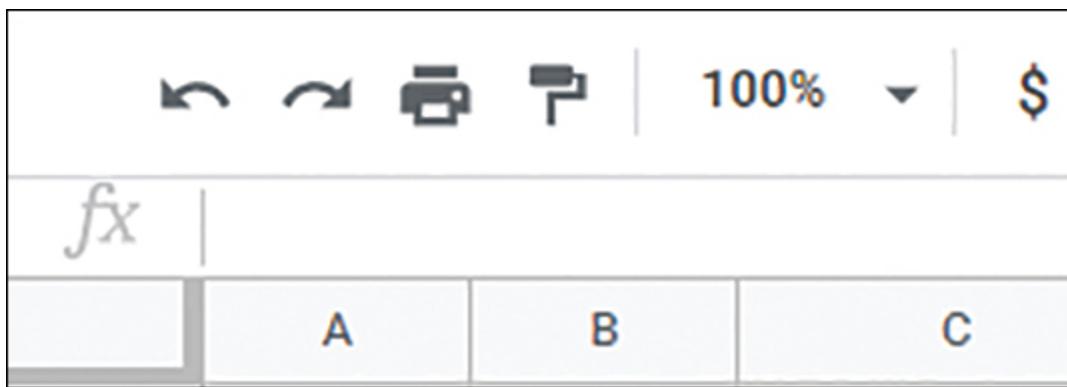


Figure 5.4 The formula bar

Spreadsheet Symbols

Spreadsheets use a wide range of symbols for a host of data manipulations and calculations. The following sections describe the basic symbols that are used most often.

Equal Sign

Spreadsheets can do both simple and sophisticated calculations in cells. To communicate with a spreadsheet that you are going to enter a formula—and not just enter a data value—you need a special code. In this case the code symbol is = (the equal sign). In a spreadsheet, the equal sign does not mean what it means in written mathematics when used to start a formula. In a spreadsheet, it's used as a special symbol that means “do something.” When you use an equal sign in a spreadsheet, it is as if you were saying, “I want this cell’s value to equal the result of the formula I am about to type.”

Unlike in basic math, where the equal sign is between the parts of an operation (for example, $1 + 1 = 2$), the equal sign appears first in a spreadsheet cell that contains a formula, function, or reference. An equal sign is the most important symbol used in a spreadsheet, as it unlocks the use of all the rest of the symbols. Without an equal sign at the start of a cell, the spreadsheet cannot do calculations.

For example, if you type = $1+1$ in a cell, the cell will display the value 2. If you type in just $1+1$, the cell will display the value $1+1$. In [Figure 5.5](#), note

that cell A1 is selected, and the contents of that cell are shown in the formula bar. Also note that the display of =1+1 in the formula bar is not the same as what is displayed in cell A1 in the spreadsheet, which is 2. Technically, both are correct; remember that the formula bar always displays what you have typed in, whereas the cell (by default) displays the value—the result of what you have typed in.

A screenshot of a spreadsheet application. The formula bar at the top shows the formula $=1+1$. The spreadsheet grid below has columns labeled A and B and rows labeled 1 and 2. Cell A1 contains the value 2, and cell B1 contains the formula $=1+1$. The cell A1 is highlighted with a blue border, indicating it is selected. The cell B1 is also highlighted with a blue border, indicating it is active or selected.

	A	B
1	2	$=1+1$

Figure 5.5 The formula bar showing a formula and cell A1 showing the result of the formula

Formulas can get very complex and powerful. You can, for example, use the equal sign in conjunction with a cell address to tell the spreadsheet to access the contents of the designated cell. In [Figure 5.6](#), for example, cell B1 is referencing cell A1.

A screenshot of a spreadsheet application. The formula bar at the top is not visible in this specific view. The spreadsheet grid below has columns labeled A and B and rows labeled 1 and 2. Cell A1 contains the value 1, and cell B1 contains the formula $=A1$. The cell A1 is highlighted with a blue border, indicating it is selected. The cell B1 is also highlighted with a blue border, indicating it is active or selected.

	A	B
1	1	$=A1$

Figure 5.6 Cell B1 referencing cell A1

The calculated value for cell B1 in this case will be the same as the calculated value for A1 (see [Figure 5.7](#)).

<i>fx</i>	=A1
	A B
1	10
2	10

Figure 5.7 A1 is the same as B1

A spreadsheet can use a cell reference to perform simple and complex calculations. In the example in [Figure 5.8](#), cell B1 is adding 5 to the reference of cell A1.

<i>fx</i>	=5+A1
	A B
1	10
2	15

Figure 5.8 A reference modified with a formula

Note

You may at times include multiple formulas or functions in the same cell, but the equal sign is needed only as the first character in the cell. You do not, and should not, type an equal sign in front of multiple formulas and functions in the same cell. Starting a cell with = is like

flipping a switch that lets the spreadsheet know it is going to perform an active task in the cell, as opposed to holding passive data.

Parentheses

Parentheses work in a spreadsheet in a very similar manner to the way they work in mathematical notation, but they can also do even more. At their most basic, parentheses denote order of operation. Any calculation inside a matched set of open and close parentheses will occur before outside operations occur. [Figure 5.9](#) shows formulas in cells A1 and B1 that are the same except that the version in B1 includes parentheses. These parentheses cause the cell to return a completely different value: cell A1 will return 10.75, and B1 will return 2.75.

	A	B
1	=2*5+6/8	=2*(5+6)/8

Figure 5.9 Formulas with and without parentheses

Note

By default, the spreadsheet will only display the formula result in the cell, but the formula itself will always be displayed in the formula bar.

You can also use parentheses in conjunction with cell references, as shown [Figure 5.10](#). In cell A1, the value of 10 is a static numerical value that is being referenced by cell B1 to use in a formula.

<i>fx</i>	= $(5+A1)*3$
	A ▾
1	10
2	45

Figure 5.10 Using parentheses in conjunction with a cell reference

Beyond denoting order of operations, parentheses act as containers for functions, as discussed later in this chapter.

Quotation Marks

Quotation marks in a spreadsheet, like in many programming languages, denote text. A spreadsheet reads anything inside quotation marks as nonfunctional and nonnumeric. For example, in [Figure 5.11](#), the cell starts with an equal sign, as it should, but the calculation is enclosed in quotation marks, so the spreadsheet returns the display value of the calculation (that is, 1+1) instead of returning the calculated value (that is, 2).

<i>fx</i>	= $"1+1"$
	A
1	1+1
2	

Figure 5.11 Using quotation marks to turn numbers into text

Other Mathematical Symbols

Standard mathematical symbols that work in a calculator also work in spreadsheet formulas:

- +: Addition
- *: Multiplication
- /: Division
- -: Subtraction
- ^: Exponentiation (to the power of)

Any of these symbols can be applied to a formula or reference. In [Figure 5.12](#), for example, cell B1 is adding together the contents of cells A1 and A2 and then multiplying that result by 5.

The screenshot shows a spreadsheet interface. The formula bar at the top contains the formula $=\text{A}1+\text{A}2)*5$. Below the formula bar is a 2x2 grid of cells labeled A and B. Cell A1 contains the value 1, and cell A2 contains the value 2. Cell B1 contains the value 25, which is the result of the formula. Cell B2 contains the value 3. The cell B1 is highlighted with a blue border, indicating it is selected. The entire grid is enclosed in a light gray border.

	A	B
1	2	25
2	3	

Figure 5.12 A formula with multiple references in cell B1

Ampersand

The ampersand (& symbol) represents concatenation—which basically means sticking text together. With concatenation and quotation marks, a spreadsheet can take several different pieces of text and numbers and create a complete phrase with them. In [Figure 5.13](#), for example, cell B2

references information stored on column A and concatenates all of it together to form a larger piece of text.

	A	B	C
1	pizza	I will pay double for pizza which is:8	
2	4		

Figure 5.13 Concatenation in cell B1

Because of the cell reference, the formula in cell B2 is considered *live*, or *dynamic*, which means the same formula produces different results when the data it references changes. In [Figure 5.14](#), for example, pizza has been switched to sushi, and the value has increased.

	A	B	C
1	sushi	I will pay double for sushi which is:18	
2	9		

Figure 5.14 A dynamically updated cell

Writing complex calculations that dynamically change based on data allows you to use a spreadsheet to do many sophisticated tasks quickly without the need to rewrite evaluations and create new formulas.

Data Containers in Spreadsheets

Spreadsheets are made up of collections of cells in columns and rows. On a larger scale, columns and rows make up a grid, which is the spreadsheet. In modern programs, users are able to make multiple spreadsheets connected to each other and all open in memory at the same time. This is called a *workbook* and contains all sheets as tabs, which can be seen below the current sheet.

Columns and Rows

Rows and columns are the next largest groupings in a spreadsheet after cells. A row contains a series of cells that runs horizontally across a sheet, and a column contains a series of cells that run vertically down a sheet. Both columns and rows are used to organize data. You can access individual cells in a column or row and edit them, or you can select an entire column or row and do group editing. In the example shown in [Figure 5.15](#), the user has clicked on the row header labeled 2, which selects all of row 2 at once. Rows are always labeled with numbers, starting with row 1, and the rows are always laid out horizontally. Columns are always labeled with letters, starting at A, and they are laid out vertically. To help remember this, think of columns in Greek architecture, which are vertical structures. Conversely, farmers plant rows of crops on the ground, and they go back and forth horizontally like rows in a spreadsheet.

	A	B	C	D
1				
2				
3				

Figure 5.15 Highlighting row 2 by selecting its row header

Sheets

The next largest grouping after columns and rows is the sheet. A sheet contains a series of rows and columns in a grid. When someone speaks of working in a spreadsheet, this is most likely what they are referring to. However, it is possible and useful to have more than one sheet. All sheets are identical when it is created; a sheet always contains the same series of lettered columns and numbered rows. As discussed in the preceding section, it is possible to select everything in a column or row by clicking on the header of that column or row. To select everything in an entire sheet, you can use the select all button, which is at the very top left of the sheet, between the column A header and the row 1 header (see [Figure 5.16](#)). By clicking the select all button, you select every cell in the sheet and can then do operations on all cells at once.

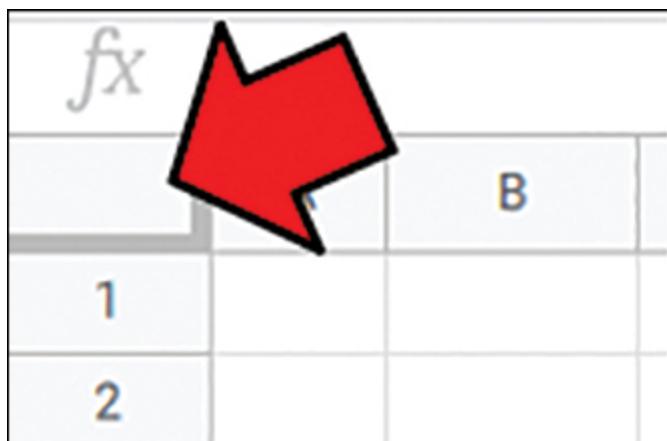


Figure 5.16 The select all button

Workbooks

When you open a spreadsheet program and load up what most people refer to as a *spreadsheet*, you are actually opening a *workbook*. The term *workbook* is also the name of a spreadsheet file that resides on your hard drive or in the cloud (in Google Sheets, for example). This term is a holdover from the days when people would manually write down figures on sheets of paper and bind them into a workbook. It also serves as a good analogy for how we work in spreadsheets. It is completely possible to create

many different, individual sheets to do all of your work in different files, but there are two distinct advantages to grouping sheets together in a workbook instead:

- **Spreadsheet organization:** When sheets are grouped together in a workbook, it helps you keep game data organized. If you know that a project is going to have characters, weapons, armor, loot, and monsters, for example, you can make a sheet for each of these different types of data but keep them all contained together in a single workbook for the game as a whole. That will help distinguish the characters in this game from characters in another game.
- **Spreadsheet connections:** When multiple sheets are contained in a workbook, they are all loaded into memory every time the workbook is opened. This means that any sheet has full and immediate access to the data on every other sheet. It is possible to access data from different workbooks, but this access involves more overhead and complications. In general, it's better to use more sheets in a single workbook if the data is related than to create separate workbooks.

To create a new sheet in a workbook, you look below the bottom of the grid of cells, where you see a gray bar there with a single tab (by default). This tab is the current sheet you are working on. In Google Sheets, it is called Sheet1 by default (see [Figure 5.17](#)).

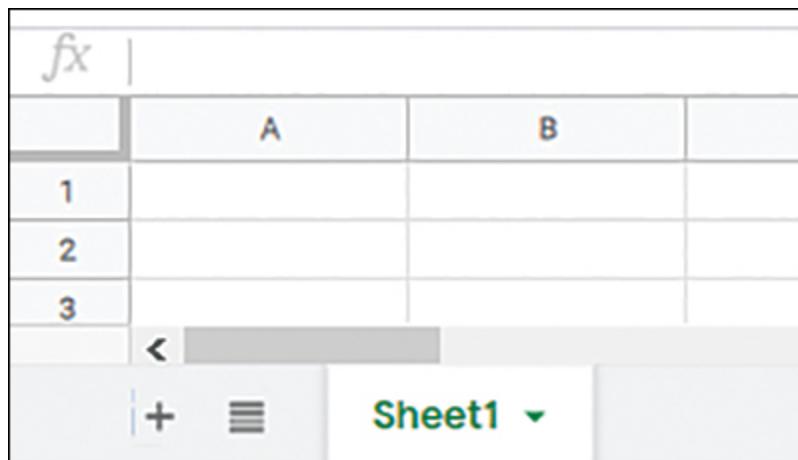


Figure 5.17 A sheet tab

To the left of all the sheets is a button that shows several horizontal lines. That is a list of all sheets. In very large workbooks, this symbol is handy because you might have more sheets than the program can display at once, and clicking this button gives you quick access to all sheets without having to scroll through them.

On the far left of the bottom tab is a + button, which you click to add a new sheet. If you click this button, the program adds a new sheet, with a new tab for the sheet, and gives it a default name; [Figure 5.18](#) shows that a new sheet called Sheet2 has been added. You can click on the new sheet tab to change your view from the original sheet to the new one.

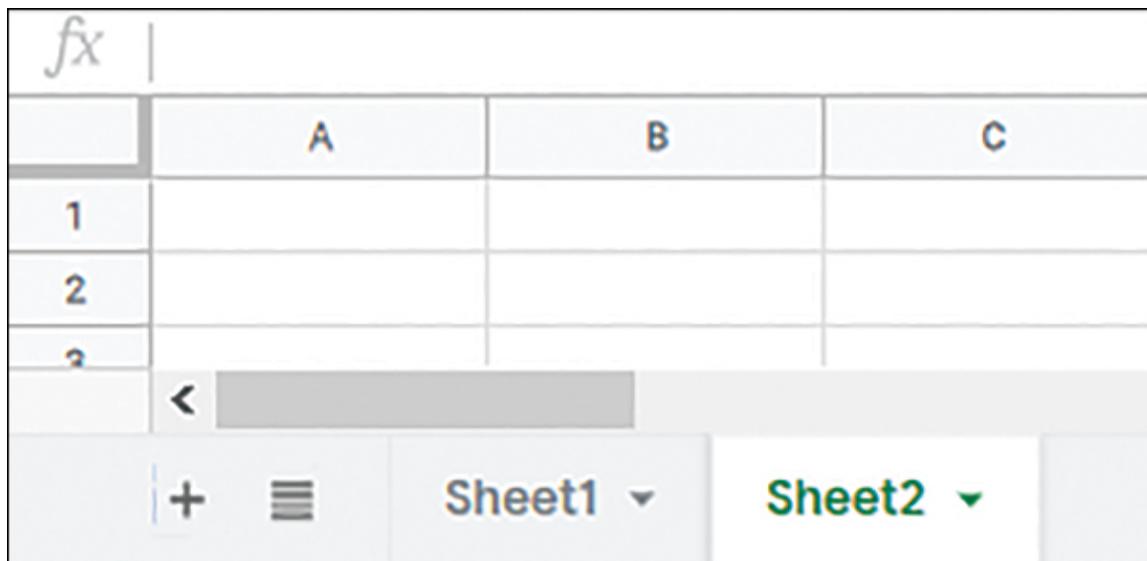


Figure 5.18 A new sheet tab

Note

It is important to note that when a workbook is open, all the data on every sheet of the workbook is open and in memory.

When a new sheet is created, it is advisable to rename the sheet to suit the contents of the sheet (see [Figure 5.19](#)). If you want, you can also color code

the sheet tabs to better organize the workbook, although the colors do not give the sheets any special functionality.

	A	B	C
1			
2			
3			
.			

The screenshot shows a spreadsheet interface with a grid of cells. The columns are labeled A, B, and C, and the rows are labeled 1, 2, 3, and a separator dot. Below the grid, there is a toolbar with icons for adding a new sheet (+), deleting a sheet (trash), and renaming a sheet (ellipsis). Two sheet tabs are visible: 'Characters' (highlighted in green) and 'Monsters' (highlighted in blue). The 'Characters' tab is currently active, showing the data in the grid.

Figure 5.19 Renamed sheets

You can move a sheet tab left or right, delete it, and duplicate it, and in some programs you can move it to another workbook. You can also hide it or protect it, as discussed in more detail later in this chapter.

Spreadsheet Operations

This section focuses on tasks you can perform with spreadsheets, such as referencing and hiding data, freezing portions of a sheet, using comments and notes, filling a form, and using filters.

Referencing a Separate Sheet

When a workbook contains multiple sheets, you need to be able to reference the data from one sheet in another sheet. [Figure 5.20](#) shows a sheet for characters and a sheet for monsters. On the Monsters sheet, you can see that there is an ogre whose strength is 10.

	A	B	C	D
1	Name	Strength		
2	Ogre	10		
3	Troll	9		
4				

Character Sheet:

- + New Sheet
- Characters
- Monsters

Figure 5.20 Organized sheet names

Say that on the Characters sheet, you would like to do a comparison between the “Soldier” character and a reference to the “Ogre Strength” in cell C2. We can see in the Monsters sheet, that the data for the ogre strength is in cell B2 of that sheet (see [Figure 5.21](#)).

	A	B	C	D	E
1	Name	Strength	Ogre Strength	Compared	
2	Soldier	5	=B2		
3					
4					

Character Sheet:

- + New Sheet
- Characters
- Monsters

Figure 5.21 A cell for doing a comparison

You can't simply write a formula that points to B2, however, because B2 on the Characters sheet contains the soldier's strength, not the ogre's. In this case, you need to use a special symbol to access a separate sheet: the ! operator.

To access data in a separate sheet, the spreadsheet needs to know where you are looking to retrieve the data. To make this clear, you need to use the name of the sheet. However, because a sheet can be given any name, the spreadsheet does not know whether what you are typing is the name of a sheet, a function, a variable, or just plain text. The ! in a spreadsheet allows you to contain a sheet name in a format that the spreadsheet can understand. When you place an ! at the end of the name, the spreadsheet recognizes the sheet name and enters that sheet. For example, to refer to the Monsters sheet, you would start by writing the following:

=Monsters!

After the ! is found at the end of the name, you can write the rest of the expression as if you were working on that sheet. In this example, if you were on the Monsters sheet and wanted to refer to ogre strength, you would write =B2. Because you want to access that same cell from a different sheet, however, you need to add the name of the sheet at the beginning of the expression with the ! to designate that it is a sheet, and then you can add the reference after that. In this example, the final expression would look as shown in [Figure 5.22](#).

The screenshot shows a spreadsheet interface with two tabs at the bottom: 'Characters' (selected) and 'Monsters'. In the 'Characters' sheet, cell A2 contains the formula '=Monsters!B2'. The formula bar also displays '=Monsters!B2'. The 'Monsters' sheet is visible in the background. The data in the 'Monsters' sheet includes columns for Name, Strength, Ogre Strength, and Compared. The 'Strength' column has a header 'Strength' and contains values 5, 10, and 15 for rows 2, 3, and 4 respectively. The 'Compared' column has a header 'Compared' and contains values 10, 15, and 20 for rows 2, 3, and 4 respectively. Row 1 is a header row with columns for Name, Strength, Ogre Strength, and Compared.

	A	B	C	D	E
1	Name	Strength	Ogre Strength	Compared	
2	Soldier	5	10	15	
3					
4					

Figure 5.22 Sheet-specific reference

Now that the cell on Monsters!B2 has been referenced, the data it contains can be used as if it resided on the current sheet.

Hiding Data

In a spreadsheet, you often want to have information on a sheet or in a workbook, but you don't need users to see that information. For example, it's a common practice to separate out building blocks of complex operations into separate cells. While these intermediate steps are important, they do not need to be in view at all times. In such cases, you can use the built-in functionality called *hiding*. You can hide rows, or columns, or entire sheets. To hide a row or column, you click on what you want to hide to select it and then right-click on the header and choose Hide Column. Once the column or row is hidden, it is no longer visible on the sheet, but it can still be accessed by formulas or functions. In the example in [Figure 5.23](#), the strength of monsters has been hidden, but the selected cell is referring to a hidden cell and showing the result.

The screenshot shows a portion of an Excel spreadsheet. The top left corner displays the formula bar with the text '=B2'. Below the formula bar is a grid of cells. Row 1 contains the header 'Name' in cell A1. Row 2 contains 'Ogre' in cell A2 and '10' in cell D2. Row 3 contains 'Troll' in cell A3. Row 4 is empty. To the right of the grid, columns A through F are labeled. A blue selection box highlights cell D2. At the bottom of the screen, there is a ribbon with tabs for 'Characters' and 'Monsters'.

	A	C	D	E	F
1	Name				
2	Ogre		10		
3	Troll				
4					

Figure 5.23 Referenced cell

Note that when a column is hidden, you can no longer see its header letter. Hidden columns are all collapsed and replaced with a single set of arrows. To make the hidden information visible again, you can click on the collapsed arrows and all hidden columns open up.

Similarly, you can hide entire sheets. To do so, you right-click a sheet tab and select Hide Sheet. Again, the information is still there and loaded into memory, but it is not visible. You might want to hide a sheet, for example, if it contains mountains of information that would clutter the workbook without providing important information for the user.

Freezing Part of a Sheet

Sheets often contain more data than can fit on a single visible page. While a spreadsheet can handle the extra information without issue, humans have a hard time reading such a spreadsheet because they have to scroll to make more information visible. In [Figure 5.24](#), for example, there is a large set of data, and you can see only some of the data that's partway down the sheet.

25	Swamp Man	m	5	5	25	2	2	4	3	4	1	Wood	y	1
28	Mummy	u	10	4	20	6	2	2	3	4	2	Cloth	y	1
27	Giant Scarab	a	8	5	15	9	2	4	3	4	1	Stone	y	1
28	Razorback	a	6	6	15	6	3	5	4	4	1	Skin	y	1
29	Serpent	a	6	6	15	6	6	4	4	4	1	Skin	y	1
30	Giant Scorpion	a	6	5	15	9	4	5	4	4	1	Stone	y	1
31	Black Bear	a	7	6	15	6	4	6	4	4	1	Skin	y	1
32	Harpy	m	6	8	15	6	5	7	4	4	2	Cloth	y	1
33	Gargoyle	m	7	5	15	6	5	6	5	4	1	Stone	y	1
34	Stone Golem	m	8	5	25	8	3	4	5	4	1	Stone	y	1
35	Giant Crab	a	8	5	15	12	5	6	5	5	2	Stone	y	1
36	Gorilla	a	9	7	15	3	3	4	6	5	1	Skin	y	1
37	Iron Golem	m	9	5	30	10	3	4	6	5	1	Iron	y	1
38	Hill Troll	m	12	4	20	5	1	4	6	5	2	Stone	y	1
39	Sasquatch	m	9	6	15	4	6	5	6	5	1	Wood	y	1
40	Cave Gobbler	a	10	8	3	1	1	4	7	5	2	Stone	y	1
41	Sabertooth	a	10	6	20	7	5	6	8	6	2	Skin	y	1
42	Giant Spider	a	8	7	15	6	5	6	8	6	1	Stone	y	1
43	Dragon	m	6	0	20	4	6	1	0	0	0	Stone	y	1

Figure 5.24 Viewing part of a table of data

What are the values listed in this example? It's impossible to tell because the labels for the attributes are at the top of the sheet, above the part of the sheet you can see here. On very simple sheets, this might not be an issue, but as the data becomes more complex, it's more important to be able to see data labels along with the data at all times. To alleviate this problem, you can freeze columns or rows. This simply means making a column or row visible, regardless of what other portion of the sheet is visible. To freeze the column, for example, you select the row, select the toolbar option View, select the Freeze menu option, and then select First Row. There are multiple options in the Freeze menu: You can freeze the top row, the first column,

and more. In our example, you can see that freezing the top row, which includes the column headings, makes the data clearer (see [Figure 5.25](#)).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Character Type	PLR	STR	DEX	HP	AV	D/MV	LVL	XP	Loot Cards	Loot	Use?	Deck	
25	Swamp Man	m		5	5	25	2 2	4	3	4	1 Wood	▼ y		
26	Mummy	u		10	4	20	6 2	2	3	4	2 Cloth	▼ y	1	
27	Giant Scarab	a		8	5	15	9 2	4	3	4	1 Stone	▼ y	1	
28	Razorback	a		6	6	15	6 3	5	4	4	1 Skin	▼ y	1	
29	Serpent	a		6	6	15	6 6	4	4	4	1 Skin	▼ y	1	
30	Giant Scorpion	a		6	5	15	9 4	5	4	4	1 Stone	▼ y	1	
31	Black Bear	a		7	6	15	6 4	6	4	4	1 Skin	▼ y	1	
32	Harpy	m		6	8	15	6 5	7	4	4	2 Cloth	▼ y	1	
33	Gargoyle	m		7	5	15	6 5	6	5	4	1 Stone	▼ y	1	
34	Stone Golem	m		8	5	25	8 3	4	5	4	1 Stone	▼ y	1	
35	Giant Crab	a		8	5	15	12 5	6	5	5	2 Stone	▼ y	1	
36	Gorilla	a		9	7	15	3 3	4	6	5	1 Skin	▼ y	1	
37	Iron Golem	m		9	5	30	10 3	4	6	5	1 Iron	▼ y	1	
38	Hill Troll	m		12	4	20	5 1	4	6	5	2 Stone	▼ y	1	
39	Sasquatch	m		9	6	15	4 6	5	6	5	1 Wood	▼ y	1	
40	Cave Gobbler	a		10	8	3	1 1	4	7	5	2 Stone	▼ y	1	
41	Sabertooth	a		10	6	20	7 5	6	8	6	2 Skin	▼ y	1	
42	Giant Spider	a		8	7	15	6 5	6	8	6	1 Stone	▼ y	1	

Figure 5.25 Frozen row

You can now see each column label, even when you scroll down the sheet.

Using Comments and Notes

Cells that contain data, formulas, or functions don't have much meaning on their own. This is why you often take extra time to add a label row or column. Frequently, however, individual cells need even more information than a column or row label can provide. It is sometimes possible to add information in an adjacent cell, but often it isn't, or there is too much information to be visible in the cell. Further, if you are not entirely sure how the flow of reference works, adding text to cells in a shared spreadsheet can potentially break functions and formulas. To solve all of these problems, modern spreadsheets provide two options that can help: comments and notes. These two options work similarly to each other but have a few differences that make each the better choice in some circumstances.

Note

With both comments and notes, the information you input can't be accessed by other cells in formulas or functions. It is added "on top of" the spreadsheet and not as active content within the sheet.

Comments

A comment is meant to be a temporary "call to action" in a spreadsheet. To add one, select the cell you would like to comment on, right-click, and select Add Comment (which is near the bottom of the context menu). A dialog pops up next to the designated cell, showing the name of the user adding the comment, a small picture of the user taken from their user profile, a text box to add the comments, and Comment and Cancel buttons (see [Figure 5.26](#)).

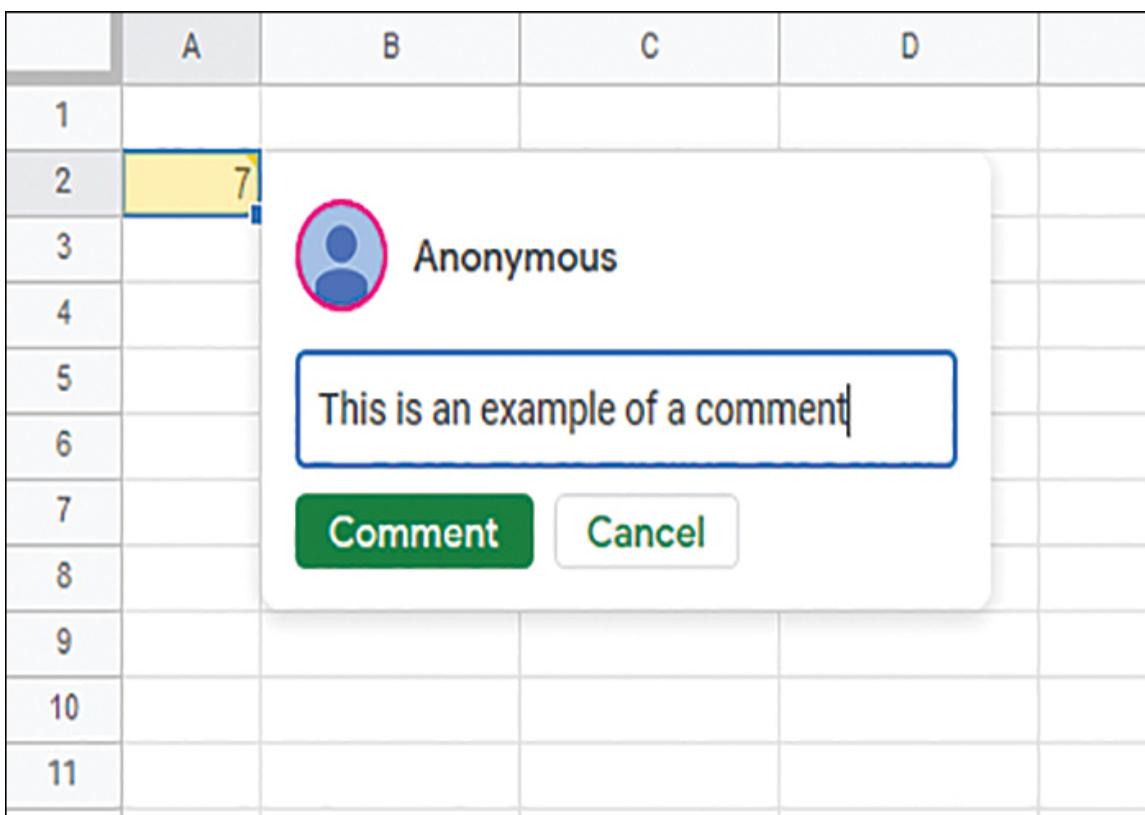


Figure 5.26 Comment dialog

In general, you should be logged on to a current account when adding comments. When you are, other users can reply to your comment, and Google Notes notifies you by email when the comment has been updated. However, adding comments anonymously, as shown in [Figure 5.26](#), still allows you to add useful comments. When you have the comment dialog box open, you can type or paste in the information you want to add to the cell. A comment is attached to a particular cell, to help others understand something about that cell's contents. Comments are meant to be temporary, so you should not add background or explanatory information. The following are some examples of the types of information you might include in comments:

This formula is referencing the wrong data.

We may need to update this value when we hear back from the programming team.

What is this formula supposed to be doing?

Can we delete this sheet?

When your comment text is complete, you can click the Comment button to commit the comment to the sheet. After that, there are a few indicators on the sheet that have changed. For example, notice in [Figure 5.27](#) how the cell with the comment, A2, now has a little orange triangle in the top-right corner. This is a *flag*, which provides an indication that there is a comment for the cell that a user can see by hovering over the cell with the mouse pointer. Also notice that on the sheet tab there is now a small word balloon with the number 1 in it. This balloon shows the number of comments on the sheet.

The screenshot shows a portion of a Google Sheets spreadsheet. The visible range is A1 to B6. Cell A2 contains the value '1'. Cell B2 contains the value '7'. The cell A2 has a small orange triangle in its top-right corner, indicating a comment. At the bottom of the screen, the 'Data' tab is selected, and a green speech bubble icon with the number '1' is visible, indicating one comment on the sheet.

	A	B
1		
2		7
3		
4		
5		
6		

Figure 5.27 Comment flag

The purpose of comments is to grab the attention of a user and direct them to the comment. When a user navigates to a sheet containing the comment

and hovers the mouse cursor over the flagged cell, the comment pops up, as shown in [Figure 5.28](#).

The screenshot shows a portion of a Google Sheets spreadsheet. The top row has columns labeled A, B, C, D, and E. Row 1 contains the number 1. Row 2 contains the number 2 and a blue checkmark icon in the top-right corner of its cell. Row 3 contains the text "Anonymous" and "11:47 AM Today". Row 4 contains the text "This is an example of a comment". The cell containing "This is an example of a comment" has a small black flag icon in its top-right corner. To the right of this cell is a vertical toolbar with a green checkmark button, a three-dot menu button, and up/down arrow buttons.

	A	B	C	D	E
1					
2	7				
3		Anonymous 11:47 AM Today			
4					
5					
6					

Figure 5.28 Comment

The user can read the comment and close it by clicking the checkmark at the top right. Google Sheets sends an email to the creator of the comment to indicate that the comment has been read.

Additionally, users can reply to a comment and start a conversation. In Google Sheets, this sends an email to the person you are replying to, allowing them to reply back.

Notes

Whereas comments are made to be temporary calls to action, notes are made to provide extra information over the long term. Unlike comments, notes do not indicate who added them, prompt emails to users, or allow for conversations. They also aren't marked by notifications on sheet tabs. Instead, notes operate very much like Notepad text files that are embedded in cells. A note is distinguished by a black flag in the top-right corner (see [Figure 5.29](#)). Notes are generally meant to be used to further explain the contents of a cell without cluttering the sheet.

	A	B	C	D
1				
2	7	Average Damage per turn for doing total character value calculations. 2.4 was original value		
3				
4				
5				
6				

Figure 5.29 Note

In [Figure 5.29](#), you can see a solitary 7 typed in a cell with no surrounding context. This would be confusing for anyone who happened on the sheet, including the future self of the person who typed it. This is a great example of a place to add a note: The note can explain why that 7 is there and what it does.

In the example in [Figure 5.29](#), the note explains the use case for the 7 in the cell and also explains a little history on why that number is there. Notes are excellent for both of these uses. In addition, notes are great for adding the following kinds of information:

- Maximum and minimum expected values for a cell
- Game rules that explain the formula in a cell
- The owner of the data in the cell
- The source of the data or formula
- Usage notes on specific functions or formulas
- Expiration dates for data that may not be valuable anymore
- Any other small snippet of information that might be helpful

The best way to use both notes and comments is to use them frequently. Adding comments on cells allows you and others to better understand the intention and usage of what might otherwise be very confusing data.

Using Formfill

One of the major advantages of using a spreadsheet is that data can be spread out over many cells. In addition, formulas can be spread out and repeated over many cells. In the example in [Figure 5.30](#), several players have played a game three times. Each time they finished the game, they entered their final scores.

	A	B	C	D	E
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total
2	Adam	37	58	49	
3	Bob	55	81	10	
4	Carl	67	32	54	
5	Dave	11	79	25	
6	Ed	56	12	89	
7	Fred	63	87	32	
8	Gary	22	86	15	
9	Herb	73	37	46	

Figure 5.30 Recorded scores for players

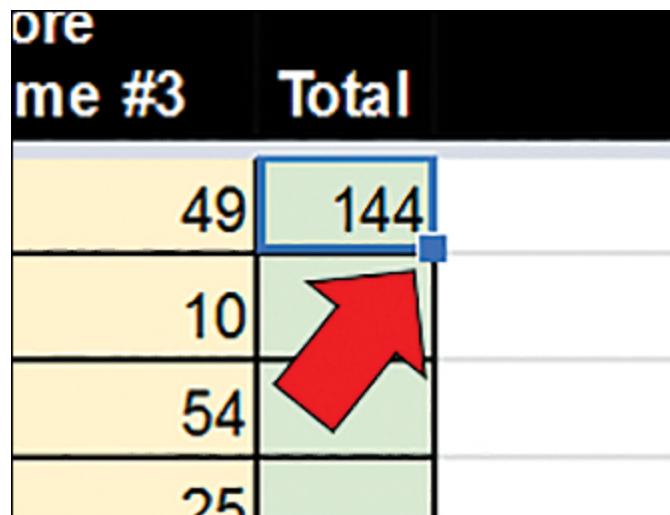
Say that you want to get the total of each player's scores. This requires a simple formula (though it could be done with a function as well, as discussed later in this chapter). In cell E2, you would write the formula =B2+C2+D2 to get Adam's total score, which turns out to be 144. Next, you would do exactly the same thing for each additional player. Rewriting the same formula seven more times would be slow and prone to error. Instead, you can use the formfill feature of the spreadsheet to let it do all the work for you.

To start using formfill, write the desired formula in a single cell. Writing the formula in the topmost cell makes the process a bit easier, although it's not required to start in this cell. Once the formula is written and behaving as intended, you can select the cell you want to copy (see [Figure 5.31](#)).

	A	B	C	D	E
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total
2	Adam	37	58	49	144
3	Bob	55	81	0	
4	Carl	67	32	54	
5	Dave	11	79	25	
6	Ed	56	12	89	
7	Fred	63	87	32	
8	Gary	22	86	15	
9	Herb	73	37	46	
10					

Figure 5.31 A template cell is selected to formfill

If you zoom in, you can see that the border around the cell has changed: It has become thicker, and the color has changed to blue (see [Figure 5.32](#)). In addition, there is a little blue box in the bottom-right corner of the selection. That is the formfill widget. When you move the mouse cursor over the widget, the cursor changes from the default arrow to a crosshair, which indicates that you can use the widget. To use it, left-click on the widget and drag your mouse down the page in the column until you reach the end of cells into which you want to copy the formula, and then let go. This tells the spreadsheet to copy and paste the formula from the first cell into all the other selected cells and update the formula to match each new selection. Alternatively, you can double-click the formfill widget to instruct the spreadsheet to copy the current cell down until it finds a blank cell one column to the left. This may seem confusing, but trying it in a spreadsheet a couple times makes this useful option completely clear.



A screenshot of a spreadsheet application showing a 4x2 grid of cells. The columns are labeled 'Score' (row 1), 'Game #3' (row 2), and 'Total' (row 3). The data is as follows:

Score	Game #3	Total
49	144	
10		
54		
25		

A red arrow points from the bottom-right corner of the cell containing '144' towards the bottom-right corner of the entire selection box, indicating the location of the formfill widget.

Figure 5.32 Formfill widget

[Figure 5.33](#) shows the result of using formfill. Notice that the spreadsheet knew to update the row references for the entries in column E as the formula was copied down. This is called *relative referencing*, and it allows you to write a formula a single time and then copy it down or over to do the same calculations in a series. It is a powerful tool in a spreadsheet, but it also has some limitations and drawbacks.

	A	B	C	D	E
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total
2	Adam	37	58	49	=B2+C2+D2
3	Bob	55	81	10	=B3+C3+D3
4	Carl	67	32	54	=B4+C4+D4
5	Dave	11	79	25	=B5+C5+D5
6	Ed	56	12	89	=B6+C6+D6
7	Fred	63	87	32	=B7+C7+D7
8	Gary	22	86	15	=B8+C8+D8
9	Herb	73	37	46	=B9+C9+D9

Figure 5.33 Result of formfill

The biggest drawback of relative referencing is that you don't always want to update every cell reference. To better show this, let's consider another example (see [Figure 5.34](#)).

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Minus Total	Minus Expected
2	Adam	37	58	49	144	
3	Bob	55	81	10	146	
4	Carl	67	32	54	153	
5	Dave	11	79	25	115	
6	Ed	56	12	89	157	
7	Fred	63	87	32	182	
8	Gary	22	86	15	123	
9	Herb	73	37	46	156	
10						
11	Expected Score	130				

Figure 5.34 Static number that needs to be referenced

Say that you now want to compare each player's score with the score you expected the player to get on the three attempts. You might also want to change the expected score at some time in the future, so you can't assume that a given number, 130 for example, will always be the desired number.

Note

It is not advisable to put variables like “expected score” on the same sheet as the data, but it is done in this example for ease of display in the book.

To calculate column F, the “score minus expected” value, you need to subtract the expected score from the total listed for each player in column E. It is straightforward to write the first formula in cell F2 as a single calculation, but in this case, you need to write it in such a way that it can be formfilled down. If it is written the same way as the formula in cell E2 was written, it looks as shown in [Figure 5.35](#).

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total	Minus Expected
2	Adam	37	58	49	144	14
3	Bob	55	81	10	146	
4	Carl	67	32	54	153	
5	Dave	11	79	25	115	
6	Ed	56	12	89	157	
7	Fred	63	87	32	182	
8	Gary	22	86	15	123	
9	Herb	73	37	46	156	
10						
11	Expected Score	130				
12						

Figure 5.35 A selected cell

This would work well for cell F2, but what happens when it formfilled down? Each of the references in the formula is then updated to a new row (see [Figure 5.36](#)).

fx | =E3-B12

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total	Minus Expected
2	Adam	37	58	49	144	14
3	Bob	55	81	10	146	146
4	Carl	67	32	54	153	153
5	Dave	11	79	25	115	115
6	Ed	56	12	89	157	157
7	Fred	63	87	32	182	182
8	Gary	22	86	15	123	123
9	Herb	73	37	46	156	156
10						
11	Expected Score	130				

Figure 5.36 A row that has been formfilled

In [Figure 5.36](#), notice how, starting in cell F3, the expected score of 130 is no longer being subtracted. In the formula bar, you can see that the new formula does correctly reference Bob's total of 146 in cell E2, but now the formula is pointing to cell B12 instead of B11 for the expected score. While it makes intuitive sense that you would want to update each row for each new player but not update each row past the expected score, this does not happen automatically with the spreadsheet. Instead, you need to instruct the spreadsheet that some cell references need to be updated and changed as they are formfilled and others do not. To instruct the spreadsheet to do this, you have two options:

- **Use a relative reference:** You can use a relative reference for any cell reference that you want to change as the formula is formfilled. This is the default state for any reference and requires no changes to the reference in the original cell to make it work.
- **Use an absolute reference:** You can use an absolute reference for any cell reference that you want to remain the same regardless of whether the formula is formfilled. To designate this, you need another special character. In this case, you use the dollar sign (\$) to signify that a reference is absolute and should not be changed. It is important to note that when using \$, column and row references are independent. So, if you want an absolute reference to a single cell, you need to place a \$ in front of both the row and the column label. In this example, the expected score in cell B2 is not going to be updated, so you would enter \$B\$2 to indicate that the spreadsheet should be referencing that cell exactly, regardless of where the original cell is formfilled to. With this in mind, [Figure 5.37](#) shows the example updated to show the new formula using the proper forms of absolute and relative referencing.

fx | =\$E2-\$B\$11

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total	Minus Expected
2	Adam	37	58	49	144	14
3	Bob	55	81	10	146	16
4	Carl	67	32	54	153	23
5	Dave	11	79	25	115	-15
6	Ed	56	12	89	157	27
7	Fred	63	87	32	182	52
8	Gary	22	86	15	123	-7
9	Herb	73	37	46	156	26
10						
11	Expected Score	130				

Figure 5.37 An updated formula using absolute and relative referencing

The formula shown in the formula bar is now ready to be copied. Because the total for each player is always in column E, the cell reference has been changed to be absolute toward column E. But you want the sheet to update the reference to each new row after row 2, so this reference is left as relative. You also know you want the reference for cell B11 to never change, so both the column and row are converted to use absolute references in respect to column and row. When the formula is formfilled down the rest of the rows, it now produces the desired calculation. To provide a bit more clarity, [Figure 5.38](#) shows the same sheet with all the formulas visible.

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total	Minus Expected
2	Adam	37	58	49	=B2+C2+D2	=\$E2-\$B\$11
3	Bob	55	81	10	=B3+C3+D3	=\$E3-\$B\$11
4	Carl	67	32	54	=B4+C4+D4	=\$E4-\$B\$11
5	Dave	11	79	25	=B5+C5+D5	=\$E5-\$B\$11
6	Ed	56	12	89	=B6+C6+D6	=\$E6-\$B\$11
7	Fred	63	87	32	=B7+C7+D7	=\$E7-\$B\$11
8	Gary	22	86	15	=B8+C8+D8	=\$E8-\$B\$11
9	Herb	73	37	46	=B9+C9+D9	=\$E9-\$B\$11
10						
11	Expected Score	130				
12						

Figure 5.38 Showing formulas for all cells

Relative and absolute references are some of the less intuitive features of spreadsheets. It takes time and considerable practice to understand when to use one and when to use the other.

Using Filters

In addition to giving you the built-in columns, rows, sheets, and workbooks, spreadsheets enable you to build your own defined selections of cells with the filters feature. To use a filter, you first enter your data as discussed earlier in this chapter. Let's look at an example that continues the previous example of players playing games and tracking scores (see [Figure 5.39](#)).

	A	B	C	D	E	F	
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Total	Minus Expected	
2	Adam	37	58	49	144	14	
3	Bob	55	81	10	146	16	
4	Carl	67	32	54	153	23	
5	Dave	11	79	25	115	-15	
6	Ed	56	12	89	157	27	
7	Fred	63	87	32	182	52	
8	Gary	22	86	15	123	-7	
9	Herb	73	37	46	156	26	
10	Adam	73	37	46	156	26	
11	Bob	67	32	54	153	23	
12	Carl	37	58	49	144	14	
13	Dave	11	79	25	115	-15	
14	Ed	56	12	89	157	27	
15	Fred	63	87	32	182	52	
16	Gary	55	81	10	146	16	
17	Herb	22	86	15	123	-7	
18							

Figure 5.39 Data to filter

To use a filter, you select all of your data, including the headers, and then click the Create Filter button (which looks like a funnel) from the far-right end of the toolbar (see [Figure 5.40](#)).

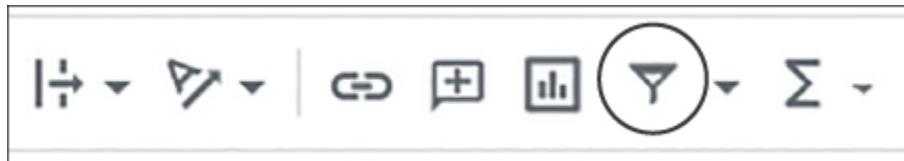


Figure 5.40 The Create Filter button

When you click this button, the spreadsheet applies filters to your entire selected data range, which opens up an entire suite of new options. The data also visibly changes. The column headers change to a shade of green, and new triangular buttons appear in the first row (see [Figure 5.41](#)).

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Minus Total	Expected
2	Adam	37	58	49	144	14
3	Bob	55	81	10	146	16
4	Carl	67	32	54	153	23
5	Dave	11	79	25	115	-15
6	Ed	56	12	89	157	27
7	Fred	63	87	32	182	52
8	Gary	22	86	15	123	-7
9	Herb	73	37	46	156	26
10	Adam	73	37	46	156	26
11	Bob	67	32	54	153	23
12	Carl	37	58	49	144	14
13	Dave	11	79	25	115	-15
14	Ed	56	12	89	157	27
15	Fred	63	87	32	182	52
16	Gary	55	81	10	146	16
17	Herb	22	86	15	123	-7
18						

Figure 5.41 A table with filters applied

Clicking on the triangular filter button in cell A1 brings up a new filter dialog that looks as shown in [Figure 5.42](#).

A	B	C	D	E
Player Name	Score Game #1	Score Game #2	Score Game #3	Total
Adam	Sort A → Z			144
Bob				146
Carl	Sort Z → A			153
Dave				115
Ed	Sort by color			157
Fred				182
Gary	Filter by color			123
Herb	▶ Filter by condition			156
Adam				156
Bob	▼ Filter by values			153
Carl				144
Dave				115
Ed				157
Fred				182
Gary				146
Herb				123

[Select all - Clear](#)

🔍

(Blanks)

Adam

Bob

Carl

Cancel OK

Figure 5.42 Filter dialog

Most of the options in this dialog are fairly self-explanatory. You can sort the entire set of data alphabetically or by color for the selected column, or you apply a filter based on many different conditions. The filtering options in this dialog are as follows:

- **Filter by color:** Shows only cells with the specified background color.
- **Filter by condition:** Allows you to filter numbers below or above a number you want, or based on many other variations of ranges.
- **Filter by values:** Shows a list of all the values listed in that column. In the current example, this would be the names of players. This list is dynamically generated by the spreadsheet for each column to have all the current values. From here, you can check or uncheck each value individually, or you can select all or clear all.

By selecting the desired filters, you can keep a lot of information in a single sheet but display only the rows you want to see at any given time. For this example, say that you want to filter down to seeing only Adam and Bob, as shown in [Figure 5.43](#).

	A	B	C	D	E	F
1	Player Name	Score Game #1	Score Game #2	Score Game #3	Minus Total	Expected
2	Adam	37	58	49	144	14
3	Bob	55	81	10	146	16
10	Adam	73	37	46	156	26
11	Bob	67	32	54	153	23
12						

Figure 5.43 Filtered data

When filters are applied, only the rows you want to see are visible, but all the other data remains in the spreadsheet. You can tell the rows are still there by looking at the row headers. Notice in [Figure 5.43](#) how the row

numbers skip from 3 to 10. Rows 4 through 9 are currently being hidden by the filter.

Another important thing to note is that the columns that are causing the filter to be applied have now replaced the triangle with a funnel icon in the header cell (in this case, cell A1). If you ever open a spreadsheet and find that much of the expected data is missing, you should check to see if any filters are applied. Look across the header row for any funnel icons, which indicate that the columns are being filtered. To remove a filtered view, right-click the funnel icon and undo any filters by selecting them or choosing Select All from the context menu. You can also remove all filters by selecting all cells and clicking the filter button in the toolbar.

Data Validation

By default, users can input any kind of data they want into any given cell of a spreadsheet. This is great for creating new workbooks with a vast array of variation, but once a workbook is created and you have a better idea what kinds of data you want in a cell, being able to input anything becomes a liability. If there are formulas or functions in the sheet that need to do mathematical equations on referenced cells, then it is vital that the cells contain the correct kind of information. For example, say that you want to find the total number of siblings a group of people has, and you ask everyone to enter that information in a cell on a spreadsheet. The following is a sample of the kinds of responses you might get:

2

Three

A brother

Only child

Sally and Dave

My parents keep having kids

2, but also a stepbrother

If we were to put this list into a spreadsheet and add the cells, you would get the result shown in [Figure 5.44](#).

	A	B
1		2
2	Three	
3	A brother	
4	Only child	
5	Sally and Dave	
6	My parents keep having kids	
7	2, but also a step brother	
8		2 Total

Figure 5.44 Badly entered data

This spreadsheet calculates the answer to be 2, although you know this answer is incorrect. Unfortunately, you don't know what the real answer is and can't know from the information listed. Row 6 does not even give an answer that can be turned into a number. The list is filled with *improperly formed*, or *invalid*, data. Getting this invalid data is an almost universal problem with collecting and analyzing data. To solve this problem, you need to take away some power from users and limit what they are allowed to enter in cells. To do this, you use data validation. Data validation allows you to specify exactly what a user is allowed to enter in any given cell. To start using data validation, you select a cell that you want to limit. You can remake the list from the siblings example and label it better for easier use, as shown in [Figure 5.45](#).

	A	B	
1	User #	Number of siblings	
2	1		
3	2		
4	3		
5	4		
6	5		
7	6		
8	7		
9		0 Total	
10			

Figure 5.45 An organized entry list

Now the sheet is formatted in a way that will provide some clues about what to enter, with color coding and proper labeling of the input column, including an instruction. To take it further and guarantee that only numeric values are entered, you need to select the range of cells that should be limited—in this case, from cell B2 through cell B8. To select a range in a spreadsheet, you first click on one end of the range (in this case, B2), hold down the Shift key, and select the other end of the range (in this case, B8). When you do this, the entire range is selected and surrounded with the same box that has shown cell selection on all other operations (see [Figure 5.46](#)).

	A	B	
1	User #	Number of siblings	
2	1		
3	2		
4	3		
5	4		
6	5		
7	6		
8	7		
9		0 Total	

Figure 5.46 A selection of cells

The Data Validation Dialog

Once the range you want to limit is selected, you can right-click inside the range. scroll down the menu until you find Data Validation (near the bottom of the list), and click it. The Data Validation dialog, shown in [Figure 5.47](#), pops up.

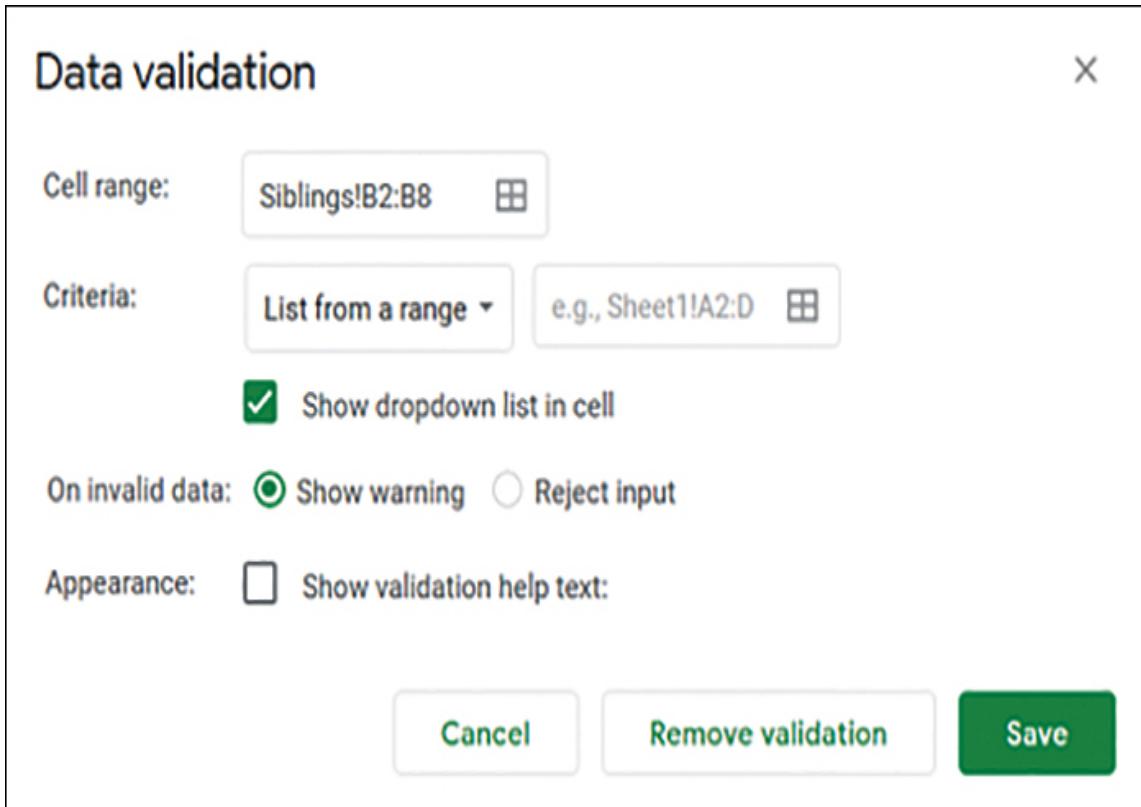


Figure 5.47 Data Validation dialog

The Data Validation dialog has several options for limiting selections:

- **Cell Range:** This is the list of cells you want to limit. Notice that in [Figure 5.47](#), Cell Range is set to Siblings!B2:B8. This is a reference to a range of cells. It does not list out each cell individually but instead uses a colon (:) to show that there is an entire range of cells between the first listed cell, B2, and the last, B8. This method of defining a range is available for all operations in spreadsheets.
- **Criteria:** By default, Criteria is set to List from a Range, as shown in [Figure 5.47](#). In this case, you know you want numbers, so you need to change this setting. To change how the cell is limited, click Criteria and choose which method you want to use, such as Number. As shown in [Figure 5.48](#), when you select Number, more options appear.

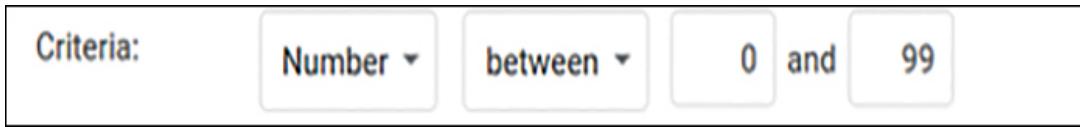


Figure 5.48 Validation Criteria

For this example, you want any number between the minimum and maximum possible siblings. Because it's hard to say what would be the largest number you could receive, you should choose a number that is larger than you would expect. If you want very specific minimum and maximum values, then use those numbers.

- **On Invalid Data:** This option lets you either warn a user they are putting in questionable data but allows them to do so anyway or rejects the data. For our example, an incorrect response should be rejected.
- **Appearance:** This final option allows you to write a helpful note to users.

After all the options have been adjusted, the Data Validation dialog should look as shown in [Figure 5.49](#). When it looks as you want it, click Save.

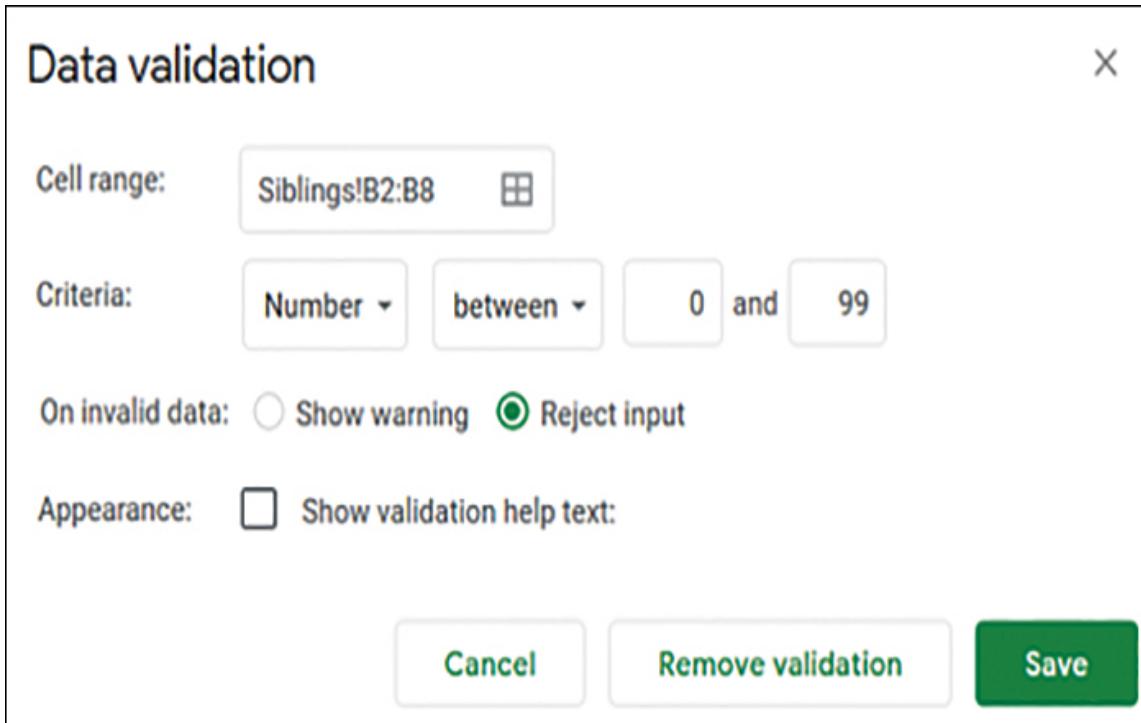


Figure 5.49 Completed Data Validation dialog

With these rules now in place, users are not allowed to enter invalid data.

Time Validation

Data validation works not only for numbers but for many other forms of data as well. In particular, dates are troublesome. The recording format of dates and times has been problematic since the first computers tried to use dates. The scales we use for time don't match up with how computers calculate time. Further, different regions and organizations use a vast array of formats for dates and time. For example, midnight on January 1, 2000, could be written in any of the following formats—and more:

2000/1/1 12am
00/01/01 midnight
1,1,2000 12
1-2000-1 0:00

All of these formats cause havoc when trying to do calculations. To prevent this issue, spreadsheets use a single date and time format for calculations:

Month/Day/Year Hour:Minute:Second

For midnight January 1, 2000, this format looks as follows:

1/1/2000 0:00:00

Strangely, if you type this date into a cell and then reformat the cell to show as a number, you see this:

36,526.000

While this number might seem strange, it is not broken. This is the value that the computer uses to store this date. Because computers can't do calculations in actual calendar time, they convert every time into a single number. That number is the number of days that have passed since 12/30/1899 0:00:00. Hours and minutes are converted into partial days as decimal values. So midnight on January 1, 2000, happened exactly 36,526 days after the beginning of "computer time" began. We rarely if ever look at dates in this format, but it is valuable to understand what is going on behind the scenes.

List Validation

List validation is a great tool for limiting user input and also for speeding up the job of a data designer. With list validation, you create a drop-down list in a cell that is populated with any items you want. You can manually input a list of items, or you can refer to a range of data.

Named Ranges

A named range is a collection of cells that have a user-defined reference name. Named ranges are not required for setting up a sheet, but they are a powerful tool that gives you easier access to stored data or values. A named range can be as simple as a single cell, or it can include multiple cells or rows or both. A selection of any size that can be made in a sheet can become a named range.

While named ranges are not technically needed, they help in simplifying references to existing data. We can see this with the monster list from earlier in this chapter. If you wanted to use data validation to make a drop-down list populated with all the possible monsters, you could select Data, Named Ranges to open the Data Validation dialog and select the range, as shown in [Figure 5.50](#).

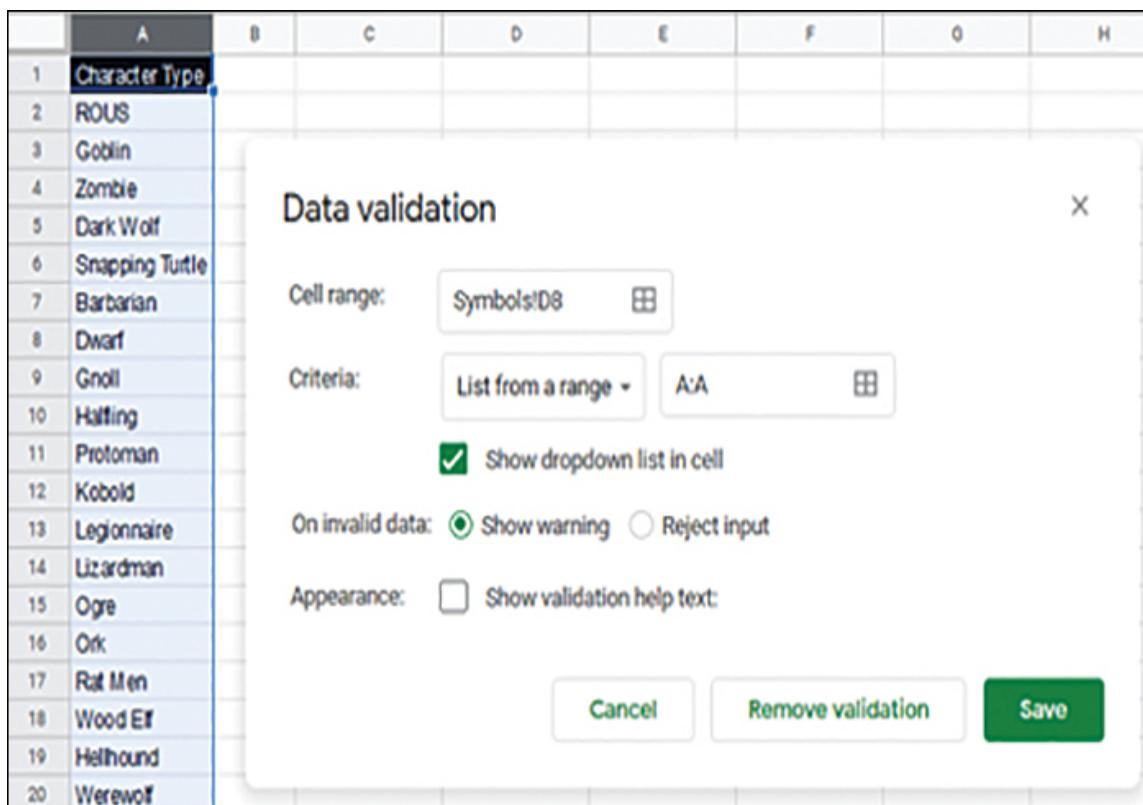


Figure 5.50 List validation

When you make the selections shown in [Figure 5.50](#), you get a drop-down list in cell D8. However, when you review the Data Validation dialog, you must remember what some of the information refers to (such as column A:A in [Figure 5.50](#)). Sometimes, it will be obvious, such as when the information is on the same page and fully visible. However, if the data is on another sheet or if it is hidden, it can take some digging to discover what A:A actually means. Because every sheet has this same range, there is no built-in clue about what is stored in that range. To prevent this kind of

confusion, you can give a range a name that is more recognizable and informative to humans (such as MonsterNames in this case). When you then see that name in the future, you will understand what data is being referenced.

To define a named range, you select the cells you want to include in the named range. Then you go to Data, Named Ranges, and you see the dialog shown in [Figure 5.51](#).

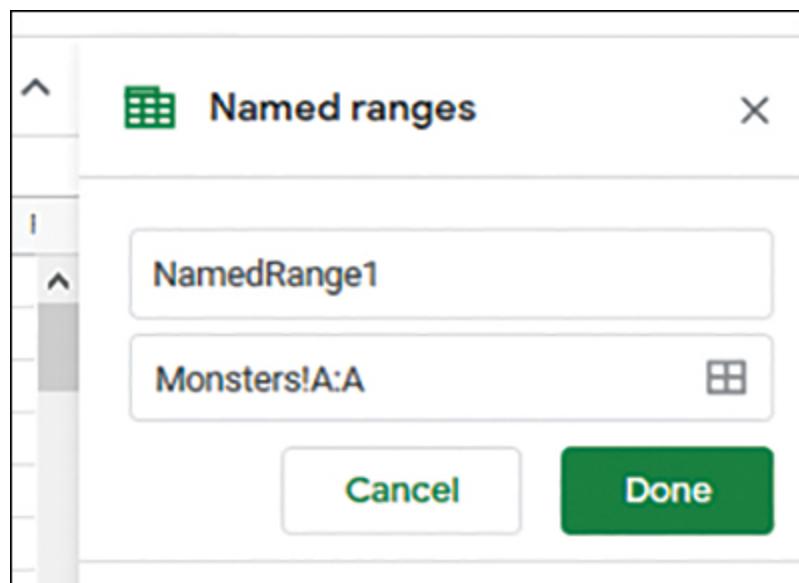


Figure 5.51 Named Ranges dialog

Because you selected the data you wanted included, the spreadsheet automatically populates the selected range for you. Note that you can modify the range in this interface, or you can even select the Mouse Select option to choose an entirely different range. These same rules for accessing other sheets also apply to this interface. Above the selection is the actual name of the range. By default, Google Sheets applies a placeholder name. It's a good idea to replace this placeholder with a more meaningful name, as the whole point of named ranges is to make the selection of areas more intuitive. In the current example, a name like MonsterNames works better (see [Figure 5.52](#)).

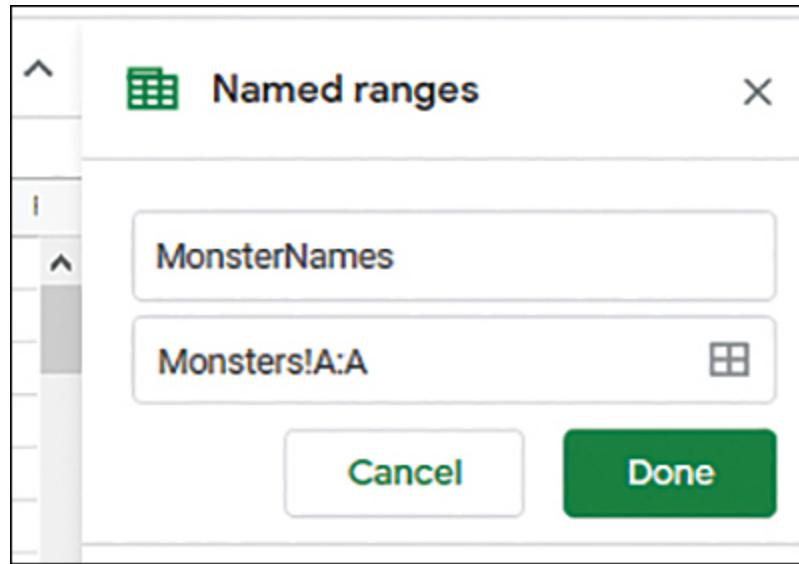


Figure 5.52 Renamed range

Note

After the name of a range and the range itself have been defined, you can click Done to create the range. It is possible to go back and rename or redefine a range at any time after this by accessing the same named range interface from the Data menu.

After the range has been named, you can use the name of the range anywhere you want to access the data in the range. For example with the example of using data validation to create a list from monster names instead of referring to Monsters!A:A, you can simply refer to the named range MonsterNames to get this functionality (see [Figure 5.53](#)).

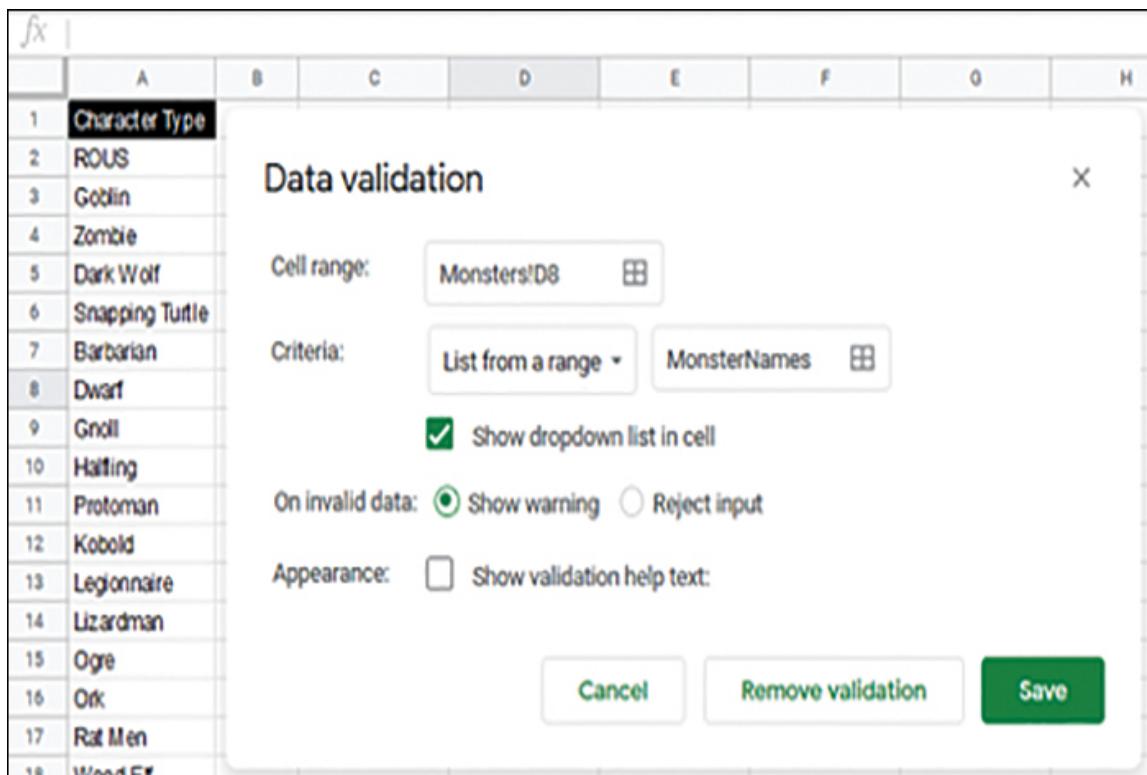


Figure 5.53 Validation with a named range

Named ranges can also be used in formulas or as variables in functions. Named ranges are particularly powerful when it comes to storing complex or long numbers. For example, if you had a cargo van in a game that could hold 1,480 pounds, it would not be particularly easy to remember that large number or to do math with it. Further, what if through the process of testing you found that the capacity needed to change? If you memorized 1,480 and typed it in manually everywhere that the number was needed, you would be left with the challenge of hunting down that number in numerous formulas and functions. Instead of going through this, it would be much easier to make a cell on a reference sheet, type in 1,480 and give that cell a name, as shown in [Figure 5.54](#).

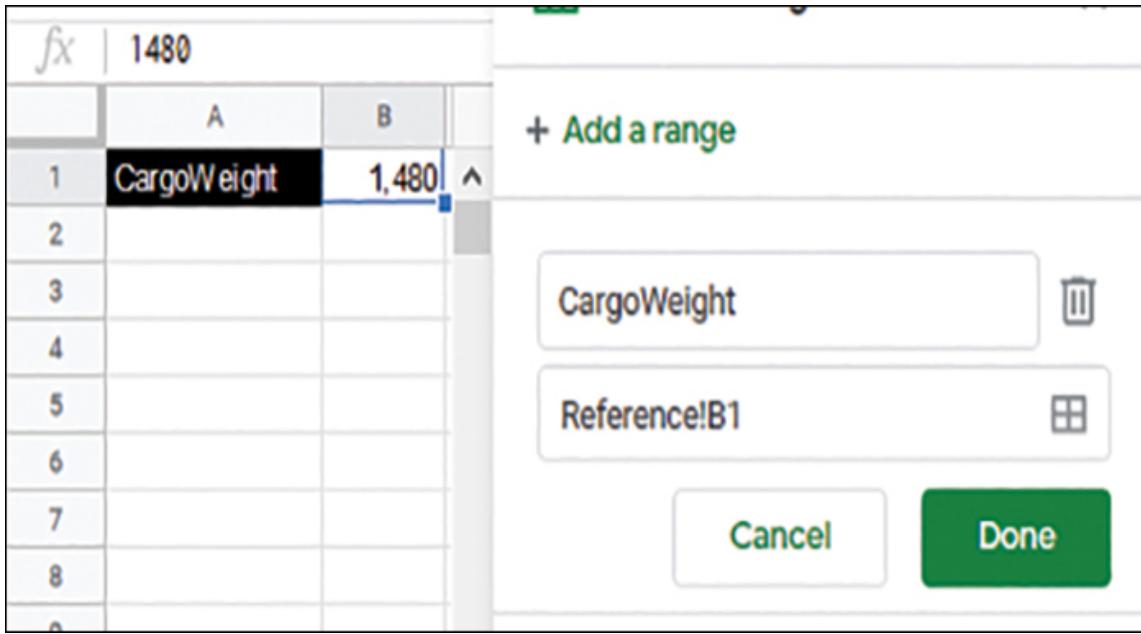


Figure 5.54 Named range cell

From this point on, the name **CargoWeight** can be used instead of the number. It is easier to remember than the number and dramatically easier to update. In a formula, it would look as shown in [Figure 5.55](#).

	A	B
1	CargoWeight	1,480
2		4440
3		

Figure 5.55 A named range in a formula

Keep in mind the following important notes about named ranges:

- Named ranges can be used from anywhere in a workbook (and so they are *global*), and they can be accessed from any sheet without requiring a reference to the sheet itself. So, for example, you can access MonsterNames and CargoWeight anywhere in the workbook by simply using those names.
- Named ranges are dynamic. If the contents of the cells they contain change, then anything referencing the range will also change. The same exception for drop-down validation also applies to named ranges: The list changes in the drop-down, but changing data in the list does not update any data elsewhere.
- A named range must have a unique name. Because they can be accessed globally, the spreadsheet needs to have distinct names for the named ranges.
- A named range's name can't contain a space. In a spreadsheet, a space denotes a new term, such as a new variable or new sheet name. You can use an underscore instead of a space in the name of a named range, or you can use camel case (for example, Monster_Names, or MonsterNames).
- Named ranges are not case sensitive, so CargoWeight, Cargoweight, and cargoweight are all treated as exactly the same name.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further investigate spreadsheet basics:

- The best way to learn to use a spreadsheet is to use spreadsheets a lot! Start thinking about ways that a spreadsheet might be useful. Making a budget is a common use, but there are also many more. You could, for example, use a spreadsheet to make an inventory of a home or an office. Or you could use a spreadsheet to track exercise numbers, weight change over time, or other health-related data. Spreadsheets are great for organizing catalogs of music, movies, and games.

- Spreadsheets can and should be used for game development. To get an idea of how helpful they are, find some game data for a game you like. Most popular games have fan sites or even official sites that show some form of data that you can copy and paste into a spreadsheet and then manipulate. By practicing with data that others have created, you can see some best practices in action and possibly some pitfalls to avoid when you start making spreadsheets with your own data.

Chapter 6

Spreadsheet Functions

Spreadsheets can calculate more than simple formulas; they can use *functions*, which are prewritten bits of code, each of which does a specific task. End users, including game system designers, use functions to quickly perform complex tasks without needing to know exactly what the underlying code does. Every function has a function name and parameters. The function is what should happen, and the parameters accept bits of data called *arguments* and tell the function what data to use as it performs the function.

Grouping Arguments

Computer programs do not know English intuitively as humans do, so you must write functions and input data in a format that the computer can understand without any ambiguity.

For example, consider the English language phrase “Pesto butter toast.” Does this phrase mean pesto and butter and toast? Or does it mean “pesto butter” and toast? Or does it mean pesto and “butter toast”? Or does it mean toast that has been coated with a pesto-infused butter mixture? As English speakers, we can use context and custom to figure out the meaning of words, but computers cannot. For a human, they could take vocal cues of emphasis, or visual clues related to what they see in front of them, to figure out the intended meaning. For a computer, you need to spell out what you mean in exacting detail. When inputting data into a function, each piece of data is called an *argument*, and you use commas to separate multiple arguments. The way you group the arguments has meaning for the computer. For example, you could write “pesto butter toast” in several different ways to communicate the exact items you want the spreadsheet to

understand. Here are several ways you could break it down for the computer, and how the computer would interpret each one:

- (pesto butter toast) is one item: toast with pesto butter on it.
- (pesto, butter toast) is two items: One is pesto, and the other is buttered toast.
- (pesto butter, toast) is two items: One is mixed pesto butter, and the other is plain toast.
- (pesto, butter, toast) is three items: Each ingredient in the list is separate.

Function Structure

Functions are, in behavior, much like machines in the real world. Things go into a machine, the machine does a preset operation on the things, and the machine returns the things changed in some way. To better understand how functions work, an analogy to using real-world machines can be helpful.

For example, consider a blender, which, as its name suggests, is a machine that blends up food. You, as the user, can place a wide variety of food items in the blender, and then the blender does what it says it does and returns a new product to you. Imagine that the blender is a spreadsheet function. Much like a blender, the function needs a machine, a container, and ingredients in order to do its job. For a spreadsheet, the “machine” is the name of the function, the “container” is a set of parentheses, and the “ingredients” are data, in the form of comma-separated arguments.

As you can see in [Figure 6.1](#), the blender is standing in for the name of the function. It is followed by the container full of ingredients. In this case, multiple ingredients (“any item”) can be placed in the container.

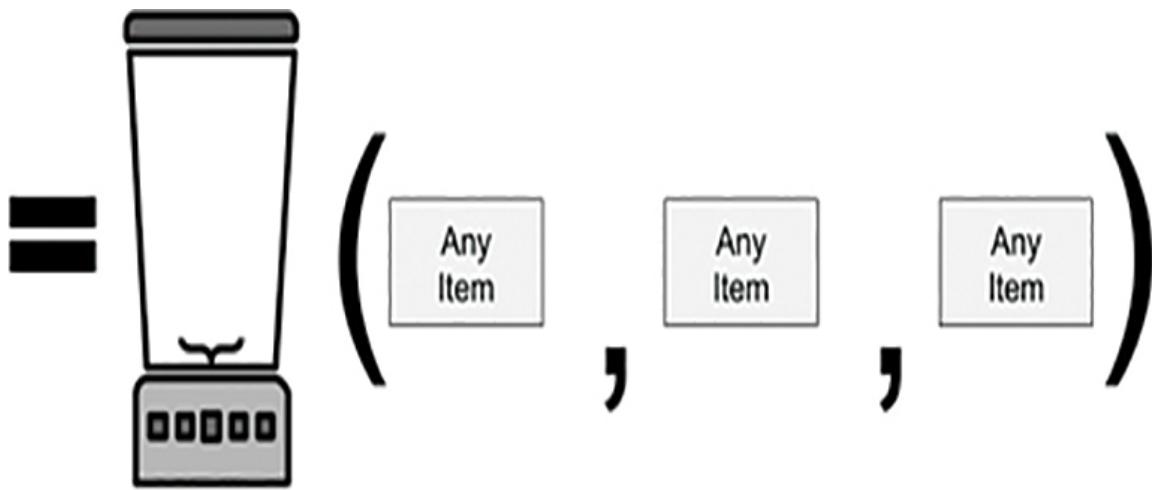


Figure 6.1 A blender as a function

[Figure 6.2](#) shows tomato, basil, and garlic placed inside the blender as the data. Once the blender is turned on, it processes these three ingredients to form an output—in this case, tomato sauce. [Figure 6.3](#) shows the same function but with different data.

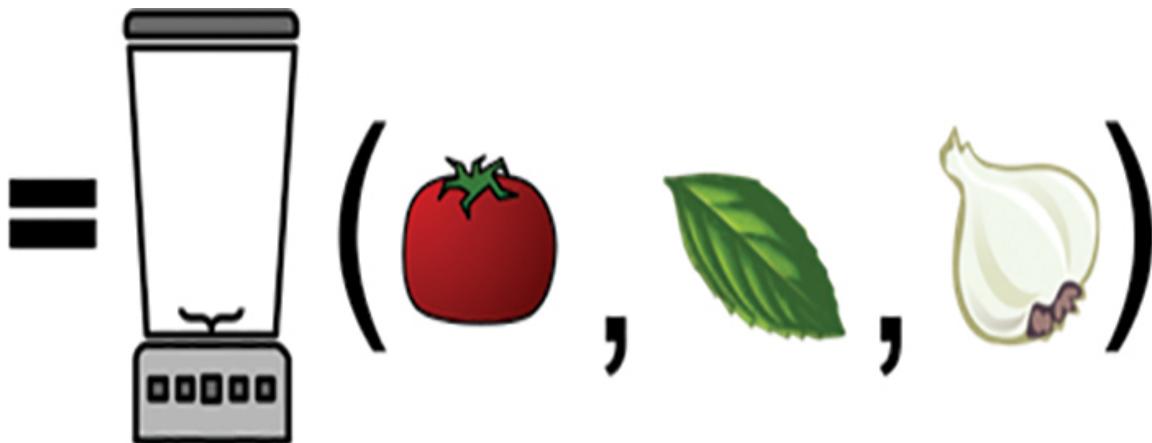


Figure 6.2 A blender with ingredients

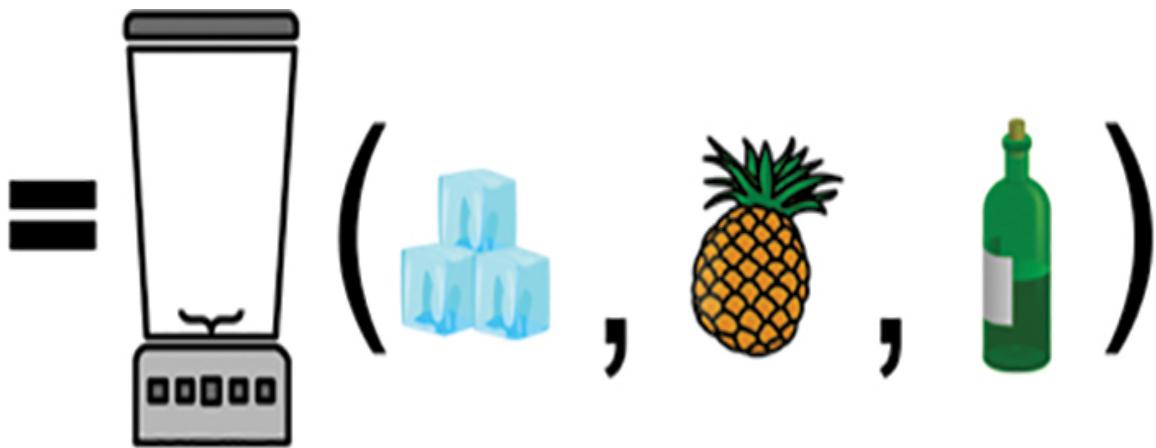


Figure 6.3 A blender with different ingredients

Again, the blender performs its job and blends together the ice, pineapple, and spirits to form a new output—in this case, a fruity drink. This is a completely different output from the same function because the data input is different.

We can break down a spreadsheet function in much the same way we've just broken down the blender ingredients. Let's consider the function SUM. Just as its name suggests, SUM is a function that adds together any data that is placed inside the function container. Here are a few examples:

=SUM(1,1) returns the value 2.

=SUM(1, 1+1, 5, 2) returns the value 10.

Functions allow for formulas or even other functions in their arguments. In the second example above, for instance, SUM would add 1+1 in the argument first, and then it would add the other arguments to get a total of 10.

In the example in [Figure 6.4](#), SUM is adding up the contents of two cells. Each cell is referred to in the arguments by the cell address.

fx	=sum(A1 ,A2)	
	A	B
1	2	
2	2	
3	SUM	4
.		

Figure 6.4 A sum or a reference

In the example in [Figure 6.5](#), SUM is adding the contents of a range that goes from A1 through A6. The colon (:) indicates to the spreadsheet that every cell in the range should be included in this argument. Notice also that different numbers of arguments have been used in this example. Some functions allow the user to input as many or few arguments as they want; SUM is one of them.

	<i>fx</i>	=sum(A1:A6)	
	A	B	
1		2	
2		2	
3		2	
4		2	
5		2	
6		2	
7	SUM	12	

Figure 6.5 The sum of a range

When working with any function for the first time, it is a good idea to look at the help file for the function to see what arguments are allowed and expected. This is called the *syntax*. For SUM, the help file shows the syntax illustrated in [Figure 6.6](#).

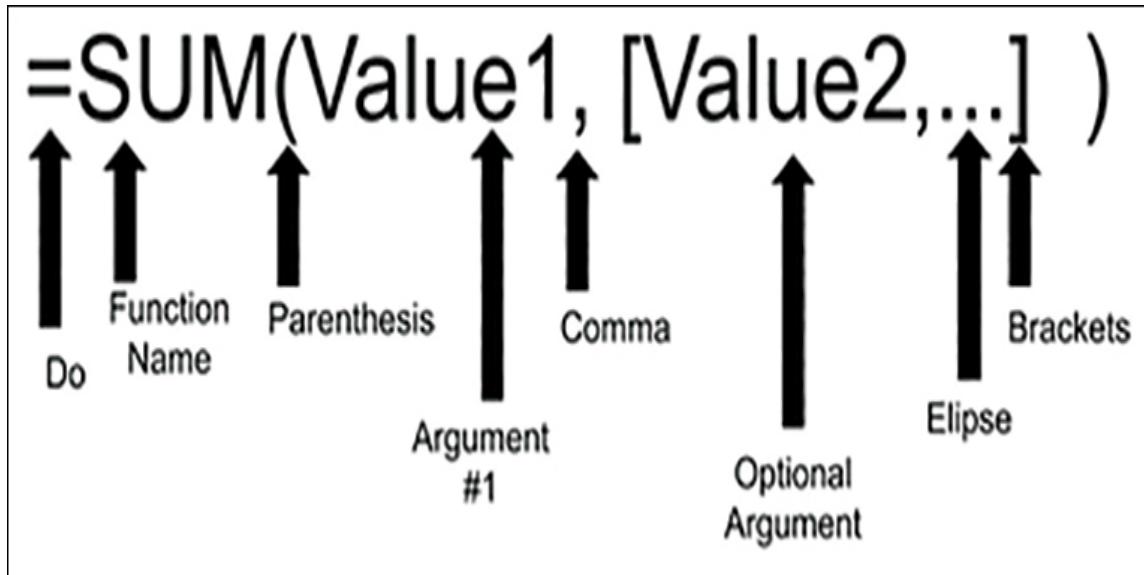


Figure 6.6 Syntax of a function

Each of these elements means something important in the context of the function:

- **Do:** Every function must start with = so the spreadsheet knows the cell is active.
- **Name:** To use a function, you need to reference it by name. In this example, the function name is SUM.
- **Parentheses:** These are the data “containers” that hold all the data arguments.
- **Argument 1:** In this function, there is only one required argument.
- **Comma:** A comma separates two individual arguments.
- **Brackets:** In the syntax of a function, brackets indicate that more arguments can be added, but they are not required. For example, SUM can be used to add together any quantity of numbers.
- **Ellipsis:** The ellipsis indicates that there could be more than one optional argument. For SUM, you are allowed to add as many arguments as you want, and the function will simply add them all together.

More Complex Functions

Like machines in the real world, some functions are more complex than others. They require that only specific types of data go in specific arguments. If you use the wrong number of arguments or the wrong type of data, the function breaks. You can see this in a real-world analogy of a car. In a simplified example of a car, it has three containers: one for gasoline, one for coolant, and one for the trunk. It would look as shown in [Figure 6.7](#) if it were a function.

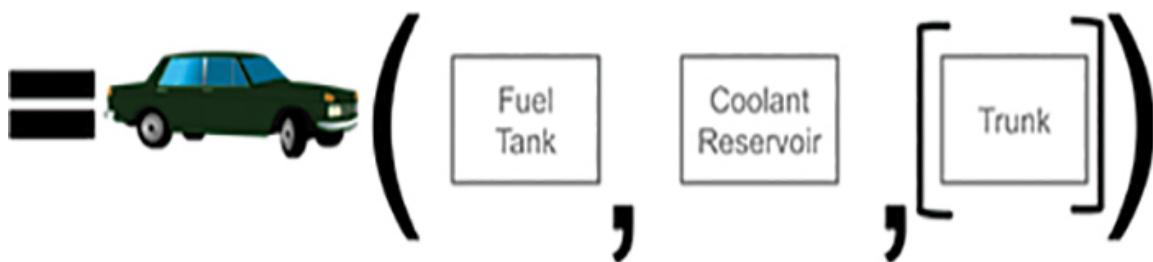


Figure 6.7 A car as a function

Notice that the arguments are no longer the vague “any item”; they are now specific items. Also, the trunk is optional (which you know because it is in brackets), as there does not need to be anything in the trunk. In [Figure 6.8](#), you can see the car with the proper arguments, running as it should.

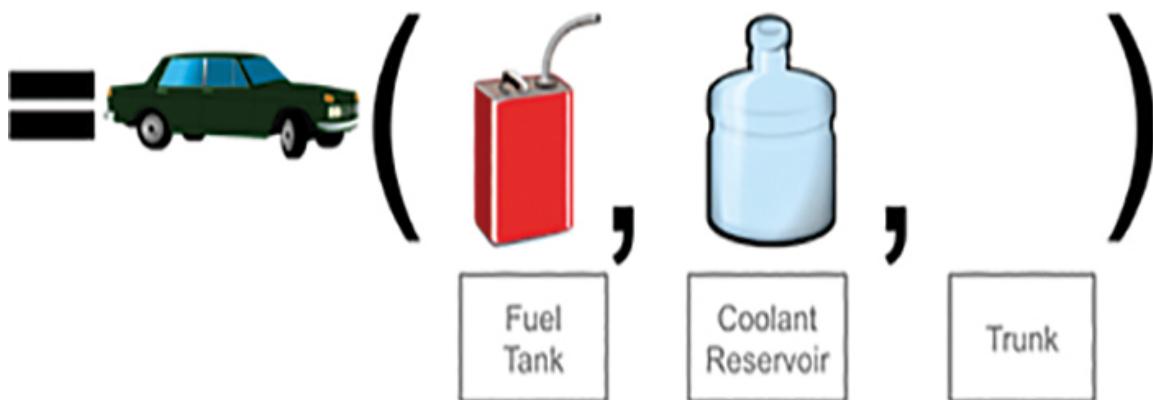


Figure 6.8 Materials in the proper places

This example shows gas in the fuel tank, water in the coolant reservoir, and nothing in the trunk. These are the correct data types, and they are placed in the correct argument spaces, so everything is working as it should.

Anything could be placed in the trunk, but in this case, it is empty. (It would work with something in it as well.)

In [Figure 6.9](#), the data types have been switched. What happens to the car now?

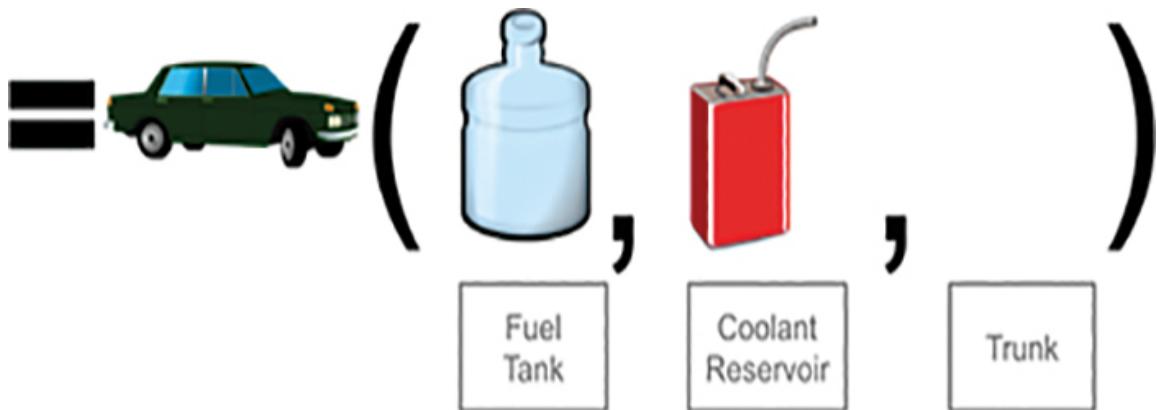


Figure 6.9 Correct materials in the wrong places

With the coolant in the fuel tank and gas in the coolant reservoir, the car does not run. Many spreadsheet functions work the same way: Not only do they require data to go in the arguments, they require specific types of data to go into the arguments. For example, the COUNTIF function has the following syntax:

`COUNTIF(range, criterion)`

This function “returns a conditional count across a range.” This means that you must give it a range and then what you are looking for. For example, if you have a list of people who are signing up to volunteer, and you want them to type “Here” when they arrive, you get a list that looks like the one in [Figure 6.10](#).

	A	B
1	Name	Check In
2	Charlotte	no
3	Ethan	no
4	Evelyn	Here
5	Mia	no
6	Elijah	Here
7	Amelia	Here
8	Isabella	no
9	Emma	Here
10	Benjamin	no

Figure 6.10 A list of people

Now that you have this list, you need to write a function to look through the list and count up the spaces where individuals have marked themselves as “Here.” For the function, this means the range is all of the cells in column B, starting at B2 and ending at B10. Within this range, the function needs to look for the word “Here” and count up how many times it appears. After the function runs, the spreadsheet looks as shown in [Figure 6.11](#).

<i>fx</i>	=COUNTIF(B2:B10, "here")	
	A	B
1	Name	Check In
2	Charlotte	no
3	Ethan	no
4	Evelyn	Here
5	Mia	no
6	Elijah	Here
7	Amelia	Here
8	Isabella	no
9	Emma	Here
10	Benjamin	no
11	# here	4

Figure 6.11 Using COUNTIF

Importantly, with the function COUNTIF, the order of arguments must be exact, and the arguments must contain the correct kind of information. Argument 1 must contain a range, and argument 2 must contain something to check for. If the order is reversed, the function will not work. If anything but a range is in argument 1, again, the function will not work.

When learning new functions, the help files are absolutely invaluable. Modern spreadsheets have both a small quick-reference help file and a larger more in-depth help file for every function available. By referencing

the help file, you can see what arguments are needed and in which order, and you can see what types of data are allowed in each.

Functions for System Designers

There are hundreds of functions available in spreadsheets, but most of them were created to solve specific problems for specific professions. This means that many of the available functions are not typically useful for a game system designer. This section focuses on the functions that are truly essential to a system designer. If you learn and understand the functions described in the following sections, you will know how to do the vast majority of the tasks that a system designer does every day.

Note

For general information on how these functions work, see the help files for your spreadsheet program. The following sections focus specifically on how to use these functions in game system design.

SUM

SUM is one of the most simple and useful functions for a system designer. As mentioned earlier in this chapter, it adds up whatever you give it. SUM can add up numbers you type into the function as arguments, or it can add up a range of cells that contain numbers—or both. [Figure 6.12](#) shows SUM being used to add up the total STR (short for Strength) attribute in a list of monsters.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	SUM	33

Figure 6.12 Using SUM

AVERAGE

The AVERAGE function finds the mean or average value of a list of values. Game system designers use this function constantly when analyzing data. Figure 6.13 shows AVERAGE being used on the same range as in Figure 6.12 to find the average STR.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	AVERAGE	4.7

Figure 6.13 Using AVERAGE

MEDIAN

The MEDIAN function serves as a good counterpoint and check against the output of the AVERAGE function. The median is the central score in a data distribution. It can provide similar information to the average, depending on the distribution of the data. However, if a set of data has extreme outliers, the average will be considerably different from the median, and knowing that there is a difference should prompt you to explore the data further. In the example in [Figure 6.14](#), for instance, the median is lower than the average. As you can see here, you can use MEDIAN in the same places as AVERAGE, and you can use both of these functions on the same data to get deeper insights into the data.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	MEDIAN	4.0

Figure 6.14 Using MEDIAN

MODE

Depending on the data set, the MODE function may or may not be useful. The mode is simply the most frequently appearing value in a data set. With data that is very granular and that has a large amount of variance, the mode doesn't provide much insight. However, if a data set has a lot of repetition, the mode can expose trends in the data. The example in [Figure 6.15](#) is multimodal (that is, more than one value is equally the most common), and the spreadsheet returns the first candidate.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	MODE	3.0

Figure 6.15 Using MODE

MAX and MIN

MAX returns the largest entry in a range of numbers. MIN returns the smallest number in a range. In [Figure 6.16](#), you can see that the highest STR value is 7, and the lowest STR value is 3.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	MAX	7
10	MIN	3

Figure 6.16 Using MAX and MIN

RANK

The RANK function can show the order of items in a distribution. In [Figure 6.17](#), for example, you can see that it also lists ties as having the same rank and then skips the next rank; for example, in this figure, two characters are tied for second, so they are both ranked 2, and no characters are ranked 3. RANK by default ranks items in order from largest to smallest, but it can also be reversed to find smallest to largest.

		A	B	C
1	Character Type	STR	Rank	
2	Goblin	3	6	
3	Zombie	4	4	
4	Dark Wolf	4	4	
5	Snapping Turtle	6	2	
6	Barbarian	7	1	
7	Dwarf	6	2	
8	Gnoll	3	6	

Figure 6.17 Using RANK

COUNT, COUNTA, and COUNTUNIQUE

COUNT, COUNTA, and COUNTUNIQUE are slightly different versions of a function that counts entries in a range:

- COUNT counts only entries in a range that are numbers.
- COUNTA counts the total number of entries.
- COUNTUNIQUE counts how many entries in a list are unique.

In the example in [Figure 6.18](#), the range A:A has a mix of words, numbers, and some duplicate words and numbers. With this example, COUNT finds any number, COUNTA counts any cell that is not blank, and COUNTUNIQUE counts only cells in the range that have unique entries.

	A	B	C	D
1	1	Function	Result	
2	2	COUNT	7	
3	3	COUNTA	10	
4	4	COUNTUNIQUE	8	
5	5			
6	6			
7	hello			
8	hello			
9	6			
10	words			

Figure 6.18 Using COUNT functions

LEN

The LEN function counts the number of characters in a cell. This might seem like something system designers would rarely use, but it is used quite frequently. In game design, many times the user interface for a player, dialog for characters, or text for instructions must fit in a designated space. To ensure that the text that is written will fit, without having to test each and every line in the game individually, teams often put all the text for a game into a spreadsheet. Then, using LEN, they can do a quick initial check and make sure no dialog is too long. In the example shown in [Figure 6.19](#), a historical quote is being checked for length to ensure that it can be quoted by an in-game character.

	A	B
1	Monolog Line	Len
2	If you hear a kind word spoken Of some worthy soul you know, It may fill his heart with sunshine If you'd only tell him so. If a deed, however humble, helps you On your way to go, Seek the one whose hand has helped you. Seek him out and tell him so. If your heart is touched and tender Towards sinner low and low, It might help him to do better If you could only tell him so.	375

Figure 6.19 Using LEN

IF

IF is a very powerful and highly adaptable function. It evaluates a *logical expression*, which is basically a statement, and calculates whether the expression is true or false. Based on this calculation, the function returns one of two different outcomes. System designers use IF all the time to do a wide variety of conditional checks. In the example in [Figure 6.20](#), an IF statement checks whether the quote from [Figure 6.19](#) is too long for the game (where 350 is the maximum character length for text).

	A	B	C
1	Monolog Line	Len	Len Check
2	If you hear a kind word spoken Of some worthy soul you know, It may fill his heart with sunshine If you'd only tell him so. If a deed, however humble, helps you On your way to go, Seek the one whose hand has helped you. Seek him out and tell him so. If your heart is touched and tender Towards sinner low and low, It might help him to do better If you could only tell him so.	375	TOO LONG!

Figure 6.20 Using IF

COUNTIF

The COUNTIF function, as its name suggests, combines features of the IF and COUNT functions you have already seen. With COUNTIF, you can apply a conditional to a counter and see how many items in that list match the conditional. In the example in [Figure 6.21](#), COUNTIF is being used to

find out how many characters in the list have a STR value that is larger than 3.

The screenshot shows a Microsoft Excel spreadsheet with a formula bar at the top containing the formula =COUNTIF(B2:B8, ">3"). Below the formula bar is a table with two columns: 'Character Type' and 'STR'. The table has 10 rows, numbered 1 to 10. Row 1 is a header row with the column names. Rows 2 through 8 list character types: Goblin, Zombie, Dark Wolf, Snapping Turtle, Barbarian, Dwarf, Gnoll, and More than 3. Rows 9 and 10 are empty. The 'STR' column contains numerical values: 3, 4, 4, 6, 7, 6, 3, and 5 respectively. The cell containing the formula is highlighted with a blue border. The cell containing the value '5' is also highlighted with a blue border.

	A	B
1	Character Type	STR
2	Goblin	3
3	Zombie	4
4	Dark Wolf	4
5	Snapping Turtle	6
6	Barbarian	7
7	Dwarf	6
8	Gnoll	3
9	More than 3	5
10		

Figure 6.21 Using COUNTIF

VLOOKUP

VLOOKUP is a slightly more sophisticated function than the ones presented so far. VLOOKUP dynamically searches a table of data and retrieves elements from it for use. You typically use it when you have lots of data in a very large spreadsheet but you want to focus on only a small part of it at one time. In the example in Figure 4.80, the spreadsheet is looking for the Barbarian character in the list and returning the value it finds in the STR column.

With VLOOKUP, the first argument is what you are looking for, which can be text (in quotation marks) or a reference. The second argument is the table that contains everything you want to search and everything you want to return. The third argument is the offset of the return. In [Figure 6.22](#), for example, the goal is to return not the first column, which is names, but the second column, which is the STR value. The final argument is optional and designates whether you want the spreadsheet to estimate (find the value nearest to the search term) or to use only an exact match. In this case, you want it to find only an exact match, so the argument is set to FALSE, which indicates “don’t estimate.”

	A	B	C	D
1	Character Type	STR		
2	Goblin	3		
3	Zombie	4		
4	Dark Wolf	4		
5	Snapping Turtle	6		
6	Barbarian	7		
7	Dwarf	6		
8	Gnoll	3		
9	Barbarian STR	7		

Figure 6.22 Using VLOOKUP

FIND

You use the FIND function to locate specific text within a large block of text in a cell. You use it to pull out key information from larger blocks of

text. In its default use, FIND returns the character position where the text starts. In [Figure 6.23](#), for example, the spreadsheet is finding the position of the word *spoken* in the large block of text.

	A	B
1	Monolog Line	FIND
2	If you hear a kind word spoken Of some worthy soul you know, It may fill his heart with sunshine If you'd only tell him so. If a deed, however humble, helps you On your way to go, Seek the one whose hand has helped you. Seek him out and tell him so. If your heart is touched and tender Towards sinner low and low, It might help him to do better If you could only tell him so.	25

Figure 6.23 Using FIND

MID

The MID function serves nearly the opposite purpose of concatenation. When you use the & character to concatenate text (as described in [Chapter 5](#), “[Spreadsheet Basics](#)”), you combined small pieces of data to make a larger text. On the other hand, MID extracts smaller pieces of data from a larger block. In the example in [Figure 6.24](#), MID is extracting all the characters up to the word *spoken*, and then it stops one character short of the first letter in *spoken*. ([Figure 6.24](#) is also a good example of a compound function that uses another function, FIND, as an argument.)

	A	B
1	Monolog Line	FIND
2	If you hear a kind word spoken Of some worthy soul you know, It may fill his heart with sunshine If you'd only tell him so. If a deed, however humble, helps you On your way to go, Seek the one whose hand has helped you. Seek him out and tell him so. If your heart is touched and tender Towards sinner low and low, It might help him to do better If you could only tell him so.	If you hear a kind word

Figure 6.24 Using MID

NOW

NOW is a bit of a strange function, in that it is not very useful on its own. If you type =now() into a spreadsheet cell, the spreadsheet returns the current time. Each time a cell is updated, the function again retrieves the current time and displays the new time. Essentially, it works as a clock that updates only when the sheet updates. The computer already has a clock, so what good is the NOW function? On its own, the NOW function does not have much use, but when combined with a static time, it becomes much more useful. By using validation or copying and pasting values only from a cell containing the NOW function, you can get a static time. When the spreadsheet has a static (unchanging) time, the NOW function can be compared against it to find out how much time has passed since previous times or how long until future times will happen. In the example in [Figure 6.25](#), cell A1 is using the function NOW, cell A2 contains a static time value, and cell A3 is measuring the distance between them.

The screenshot shows a spreadsheet interface. The formula bar at the top contains the formula $=A1 - A2$. Below the formula bar is a table with three rows. The first row has column headers 'A' and 'B'. Row 1 (cell A1) contains the date and time $9/18/2020 16:28:15$ and the label 'Now function'. Row 2 (cell A2) contains the date and time $9/17/2020 14:02:22$ and the label 'Static time'. Row 3 (cell A3) contains the duration $26:25:53$ and the label 'Duration'. The table has a light gray background, and the cells are colored in a light green for the first two rows and a light blue for the third row.

	A	B
1	9/18/2020 16:28:15	Now function
2	9/17/2020 14:02:22	Static time
3	26:25:53	Duration

Figure 6.25 Using NOW

Note

To make times in a spreadsheet easier to read for humans, it's important to change the formatting of the times (as discussed in [Chapter 5](#)) so that times are formatted as times and the distance between times is formatted as duration.

RAND

The RAND function produces a random number between 0 and 1 to a very large number of decimal places. This is another odd function, in that it updates every time the spreadsheet is updated, including when any cell is changed. Much like NOW, RAND is more useful when the result of the function is copied and then pasted as value only, to eliminate the underlying function and keep the randomly generated number. Because RAND produces a long decimal number, it is also not entirely useful for random numbers in games, which tend to avoid long random decimal numbers. However, RAND, like all other functions, can be used in other functions or as part of a formula to manipulate the default output number into something more useful, such as a number between 1 and 100. [Figure 6.26](#) shows raw output from RAND. Notice that this output, on its own, does not provide much information.

	=RAND()
1	0.54187400504263900000
2	

Figure 6.26 Using RAND

ROUND

The ROUND function does exactly what it sounds like it does: It rounds off any given number to a specified number of decimal places. You often need to use this function to make long decimal numbers more digestible by players or even your own team. [Figure 6.27](#) shows the ROUND function being used in combination with the RAND function in a simple formula to generate a more useful random range of 1 through 100.

<i>fx</i>	=round(rand()*100,0)
A	B
1	76.00

Figure 6.27 Using ROUND

Note

Functions are not case sensitive in spreadsheets. Whether you use capital letters, lowercase, or a mix of both, the spreadsheet will read any function you type in the same way. It will also update your writing to be all capitals.

RANDBETWEEN

For many applications, you don't want a random number that is between 0 and 1 or some multiple of that because you have a specific range in mind already. In such cases, you can use the RANDBETWEEN function to specify minimum and maximum values, and it randomly generates numbers within that designated range. In the example in [Figure 6.28](#), the sheet is showing the generation of random numbers between 10 and 20. Note that RANDBETWEEN, like RAND and NOW, is often best used when the result of the function is copied and then pasted as value only.

<i>fx</i>	=RANDBETWEEN(10,20)
A	B
1	14.00

Figure 6.28 Using RANDBETWEEN

Learning About More Functions

While there are many hundreds more functions in every spreadsheet, the ones listed here are the ones that system designers most commonly use. If you take time to learn these functions thoroughly, you will be able to use them to do the majority of work needed to make a game.

To find even more functions, you can click on the function symbol (the Greek letter sigma) in the toolbar (see [Figure 6.29](#)). Then, if you select Learn More, you will see an list of every function available in the spreadsheet program. Don't be overwhelmed by how many of them there are. There may be hundreds of functions, but most of them are designed to be used for very specialized professions and can be ignored by everyone else.



Figure 6.29 The function symbol

How to Choose the Right Function

You should practice with the functions described in this chapter to become more familiar with how they work. As you continue to work on spreadsheets as you make games, you are likely to run into situations where you know what you want to do, but you don't know what function can make it happen. This is a very normal problem that's even common for experienced professionals. The following steps can help you determine what function to use:

- 1. Make a plan.** Think of what you want to do and then write it out in great detail. After you have written out exactly what you want the spreadsheet to do for you, parse each word of the plan and think about what it would mean exactly.
- 2. Do it manually once.** When you know what you want the spreadsheet to do for you, go ahead and do it manually once. This might mean adding up numbers in your head or searching through large sheets of data with your eyes.
- 3. Write down your exact steps.** As you did your task manually, did you need to click on another sheet? Did you use the mouse wheel to scroll down the page to find something? Did you add things up? Take all the steps, one at a time, and write down all the steps in exact, excruciating detail.
- 4. Extract keywords.** Look at the steps you just wrote. What keywords stick out? Look for words like *searched*, *added*, *counted*, and *averaged*. Extract these keywords into a short list.
- 5. Do some research.** Go to an Internet browser window and type in “google sheets” and one or two of your extracted keywords. It is quite possible that someone else has tried to do exactly what you are doing and asked about it. It is also possible that someone else has already answered that question and given clear instructions on how to do it. In spreadsheets, functions are often named intuitively based on what they do, so if you search your spreadsheet program’s help file for your keywords, you might find that a function that does exactly what you already exists. If it does not, then you might find functions that get you part of the way there and that can be combined with other functions for exactly the effect you want.
- 6. Ask questions.** If doing research and trying the functions you find is not giving you the results you want, you can use the steps you wrote down in step 3 to ask either someone you know or people on the Internet for help. There are multiple forums that discuss spreadsheet usage, and the people on those forums tend to be very helpful to those

who ask questions clearly. See [Chapter 3](#) for information on how to ask a question.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further practice asking and refining questions:

- Create several spreadsheets and practice using the formulas and functions shown in this chapter. If you completed the exercises in the “[Further Steps](#)” section of [Chapter 5](#), you already have game data in a spreadsheet. Think about how each function can be used on that data to get an interesting result. (It is a good idea to create this spreadsheet now if you haven’t already as it will come up again in later chapters.)
- Read the help files in your spreadsheet program. This chapter discusses some important functions in enough detail to get you started, but you can find a lot more information in the help files. Functions are also updated occasionally, so it is a good idea to look at the help files in the program of your choice to make sure you understand the current working version of each function.

Chapter 7

Distilling Life into Systems

Now that you understand the fundamentals of creating and using spreadsheets, it's time to start populating a spreadsheet with new game data. When you're creating a new game, where does game data come from? Obviously, the data is generated from game systems and, more generally, from games. This means, to generate data, you need systems, and to create systems, you need to start actually making a game. This chapter covers the basics of coming up with game ideas completely from scratch and then building them out with systems and data.

The first concept to understand is that no game is created in a vacuum. When you look closely at the mechanics that compose any game, you find that they are analogs for aspects of real life, even if they are abstracted. This similarity to real life helps explain why humans (and other animals) play games.

Life is complicated, messy, random, and unfair. It is more complex than we as a society can understand. Life is constantly changing, and for the most part, it's out of our control. "Solving" life is impossible, but humans like to solve things. We like to figure out systems and understand them in total. We also like to have a full grasp of rules and completely understand them. We like to have agency in our surroundings and make decisions that have meaningful consequences. Although it is impossible to have all these things in life, it is completely possible in a game. If you remove the vast majority of systems from life and distill a game down to just a small number of mechanics, you can get a grasp of the systems and how they interact, as well as what the game means. You can also do something else in games that is not possible in life: You can make games fair. To illustrate the complexities and how we distill a system of the real world, let's take a look at two examples.

First, [Figure 7.1](#) shows a picture of a city street. Before reading on, start thinking about all the systems you can see in this single image.



Figure 7.1 A picture of a city street

Here is a small selection of systems represented in this single image:

- **City layout:** People have decided where all the buildings go.
- **Architecture, engineering, and construction:** Beyond deciding where all the buildings are located, many different groups have come up with complex systems for how each individual building is designed, constructed, and maintained.
- **Plumbing:** Each building has pipes that bring water from a centralized location to places throughout the building. Each building also has pipes that remove dirty water from the building and the streets and move it to a centralized location for treatment.

- **Electrical system:** Each building has an electrical system, as does the city street.
- **Traffic:** People are walking and driving in designated locations and according to rules. Without these systems, transportation would be nearly impossible.
- **Plant life:** You can see that a few trees are growing on this street. The decision about where to plant trees is a system, as are the trees themselves. Beyond the trees you can see that there are smaller plants growing here and there.
- **Weather and beyond:** You can see the sky in the background, which contains a multitude of weather and air systems. In addition, the rest of the entire universe is actually out there, though it's out of sight.
- **People:** Each individual person in the image is made up of a multitude of complex systems.
- **Clothing:** Everyone in the image is wearing clothes, and complex systems govern how those clothes are designed, created, purchased, and worn.
- **Society:** Taken as a whole, we humans have complex societal systems that influence everything we do.
- **Commerce:** There are multiple sophisticated economic systems in play anywhere there are large numbers of people.
- **Law:** Beyond basic societal rules, there are more rigid rules of law in place everywhere.
- **Advertising:** We can see advertisements throughout the street. Each of these obeys complex rules of law and advertising tactics to entice consumers.
- **Parasites:** It is strange to think of it, but almost certainly in an image of this many people, at least one of them has some form of parasite, whether it's lice, worms, or a case of athlete's foot. This illustrates how quickly we can go deep with systems. We could spiral into these kinds of details endlessly.

This is just a brief overview of the uncountable array of systems, rules, and interactions taking place in this single image. No game has ever gotten close to this kind of detail in its rules, and this is absolutely on purpose. Would you really want to simulate parasites in an RPG? Probably not. Would you want to address building zoning permits in a car racing game? Absolutely not. Instead, you want to make a game a very simplified impression of the world, so players can recognize it for being an abstraction of the world but without all the tiny details they don't care about within the game.

Now let's look at an abstracted version of a city street (see [Figure 7.2](#)).



Figure 7.2 An abstract image of a building

In this abstracted image of a city street, we can also see various systems: people, trees, utility poles, roads, sky, and buildings.

This list is smaller than the earlier one, and it's missing all the detail of that list. In fact, if we get very literal, we can see that this image is actually a few strokes of paint. The people in it are just a few paint blobs with paint circles slightly above them. The road is just a smudge of gray, and the trees are a few longer strokes of brown and green paint. If we were to look at any single part of the painting in total isolation, it would be difficult to even identify what the object was supposed to represent. However, when we take a casual look at the first and second images, we can easily describe each of them as showing a city view with some people on a road. In the painting, our mind fills in all the tiny details that are missing, or we simply ignore them because they aren't important to what the image is trying to communicate.

The difference between the photograph and the abstract painting is akin to the difference between real life and a game. To create a game, we look at life and then pull out a few key elements and present them to players in such a way that the players understand the impression of what we are presenting without being bogged down by the myriad details.

A game mechanic can be as simple as pattern recognition. For example, in tic-tac-toe, the goal is to make either three X's or three O's in a row. Tic-tac-toe has a very simple mechanic and a very simple set of rules. While tic-tac-toe is so simple that it is not particularly compelling for adults, it is appealing in that we can fully know all the rules. Even small children can grasp all the rules of the game. And, unlike in real life, in tic-tac-toe, players are never surprised or disappointed by unexpected rule changes midgame. Even simple interactions in real life can have unpredictable results that we don't fully understand. In tic-tac-toe, we are never denied an X because it is lost or never lose the game because our opponents bought another turn. These kinds of chaotic events happen in the real world whether we like it or think it's fair, but in the reality of the game, we do have control over these kinds of elements. We can decide what is fair and enforce it for the game. This gives games and their rules a kind of purity we never see in the real world—and that is inherently appealing to many humans.

If we take this discussion to a simple extreme, we can pull out just a few simple elements from a real-life scenario to make a well-known game. If

you have ever tightly loaded a vehicle or packed things into a storage space, there is a good chance that it reminded you of a game. When you look closely at the details, you see that it's actually much more complicated than that. [Figure 7.3](#) shows a packed car, and [Figure 7.4](#) shows a Tetris-like game. They bear an obvious resemblance to one another, and some of the basic mechanics are the same.



Figure 7.3 A packed car that resembles a game

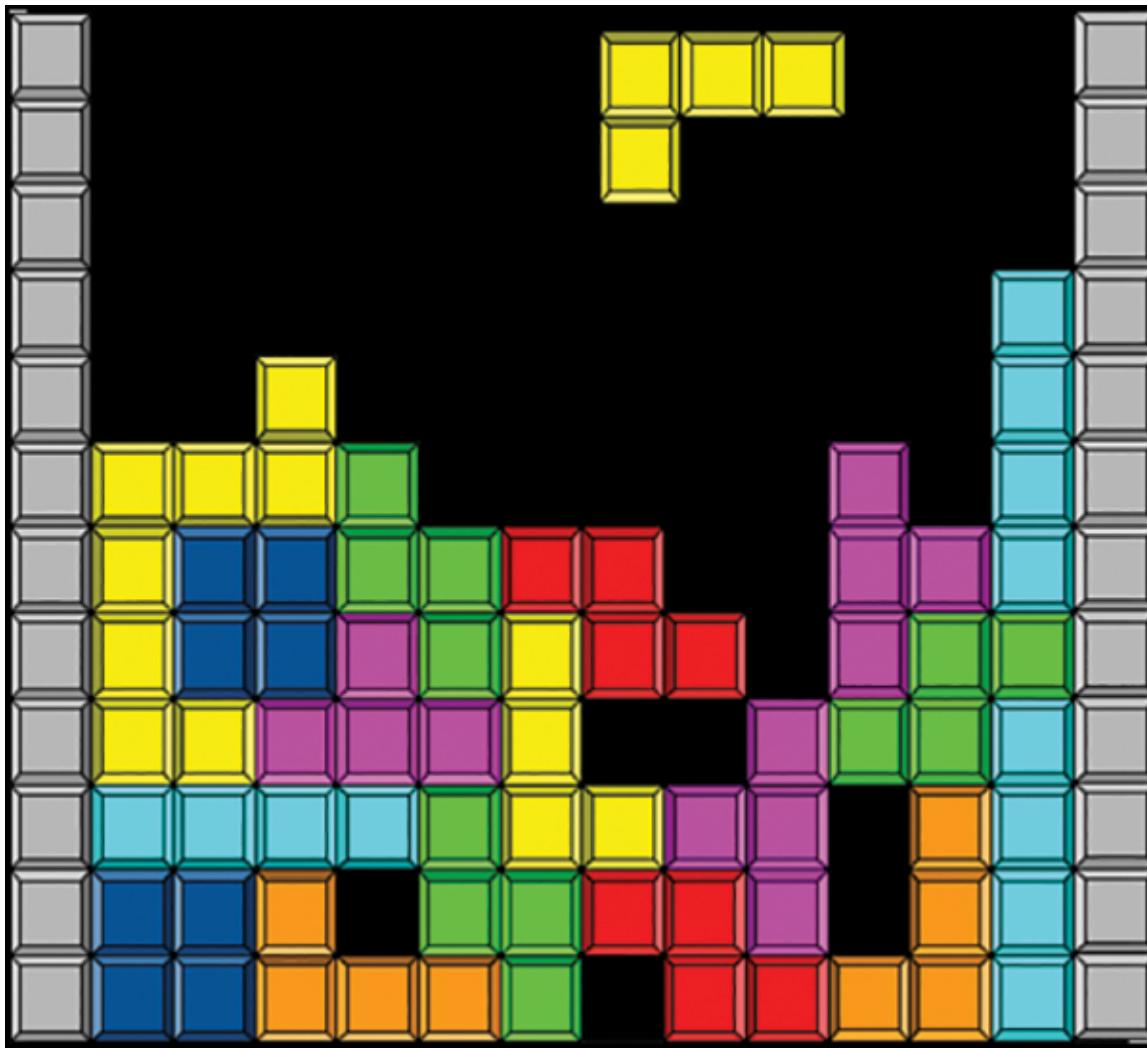


Figure 7.4 A Tetris-like game

In both instances, the “player” is trying to pack as many objects into a given space as possible, leaving as few empty spaces as possible. This underlying mechanic drives the game of Tetris, and even though the mechanic is abstracted, the inspiration is clear: The main mechanic of Tetris arises from the same built-in feeling of satisfaction we, as humans, get when we efficiently pack a space.

Now, let’s look at some major differences between the packed car and the Tetris-like game. In the car, you can see small gaps between some of the items, and it might be possible to slip very small items in there. Some of the boxes might be fragile and best placed near the top of the car. Some might

be very heavy and best placed at the bottom. In addition, it's important to consider whether all the items are going to be secure when the car moves. And is the car insured properly? Hopefully none of those containers have anything perishable or living in them that will not survive the trip. Such concerns are real and directly applicable to the real-life activity of packing up a bunch of items into a fixed space.

The game Tetris eliminates a lot of the car-packing concerns. Why did the designer decide to exclude those aspects of the real-life analog? Mostly because they aren't fun, and they distract from the core mechanic the designer wants the player to focus on: fitting shapes together in a confined space. Notice also that Tetris does things that are not possible in real life, such as eliminating a row when it is complete. The game designer has looked at the real world, recognized a mechanic, and distilled that mechanic and added to it in an abstract way to make it more enjoyable for players. This is the essence of good game design and one of the reasons Tetris has been so successful for such a long time.

An Abstract Example

Now let's take a look at a more sophisticated set of abstracted mechanics that we can later rework to make them into a popular game. For this example, we are going to look at paleolithic hunting techniques. Humans lived in that phase of existence for a very long time, and many of our most deeply held instincts were developed then. The following sections look at a few of paleolithic hunters' greatest strengths.

Throwing

Paleolithic hunters often need to throw things, such as rocks, accurately. Let's examine the mechanic of throwing things. Humans, unlike other animals, can accurately and quickly throw objects heavy enough to kill even large animals. This gave humans an enormous advantage in paleolithic hunting. By attacking with a disposable object from a safe distance, they could do damage to a target without being in immediate peril. Humans developed their throwing ability further and further, to the point that throwing things is a deeply ingrained human instinct. Just watch small children playing: They don't need to be taught to throw things, and, in fact,

parents often have a hard time stopping them from throwing things. This happens across all countries and cultures, so we can clearly say that it is a built-in instinct and not something learned through observation.

Sticks

Paleolithic humans also learned how to use sticks and other tools to gain an advantage while hunting. Unlike any other animal, humans are able to accurately and strongly swing a stick to hit something. If the stick is damaged during this activity, that's fine by us because it is easily replaced, and using a stick allows us to maintain a safer distance while attacking a target. Again, waving around sticks is something all children all over the world pick up naturally, showing just how deeply connected we are to the activity as a species.

Running

Most animals run, but humans are particularly good at it. We have evolutionary adaptations that allow us to run long distances, particularly in hot weather. Running fast is also deeply ingrained in people as an inherently good thing. Again, watch children anywhere in the world and in any society, and you will find that running is valued in some form of sport. This is a feature of humans that has been used in hunting since before we were even fully human.

Teamwork

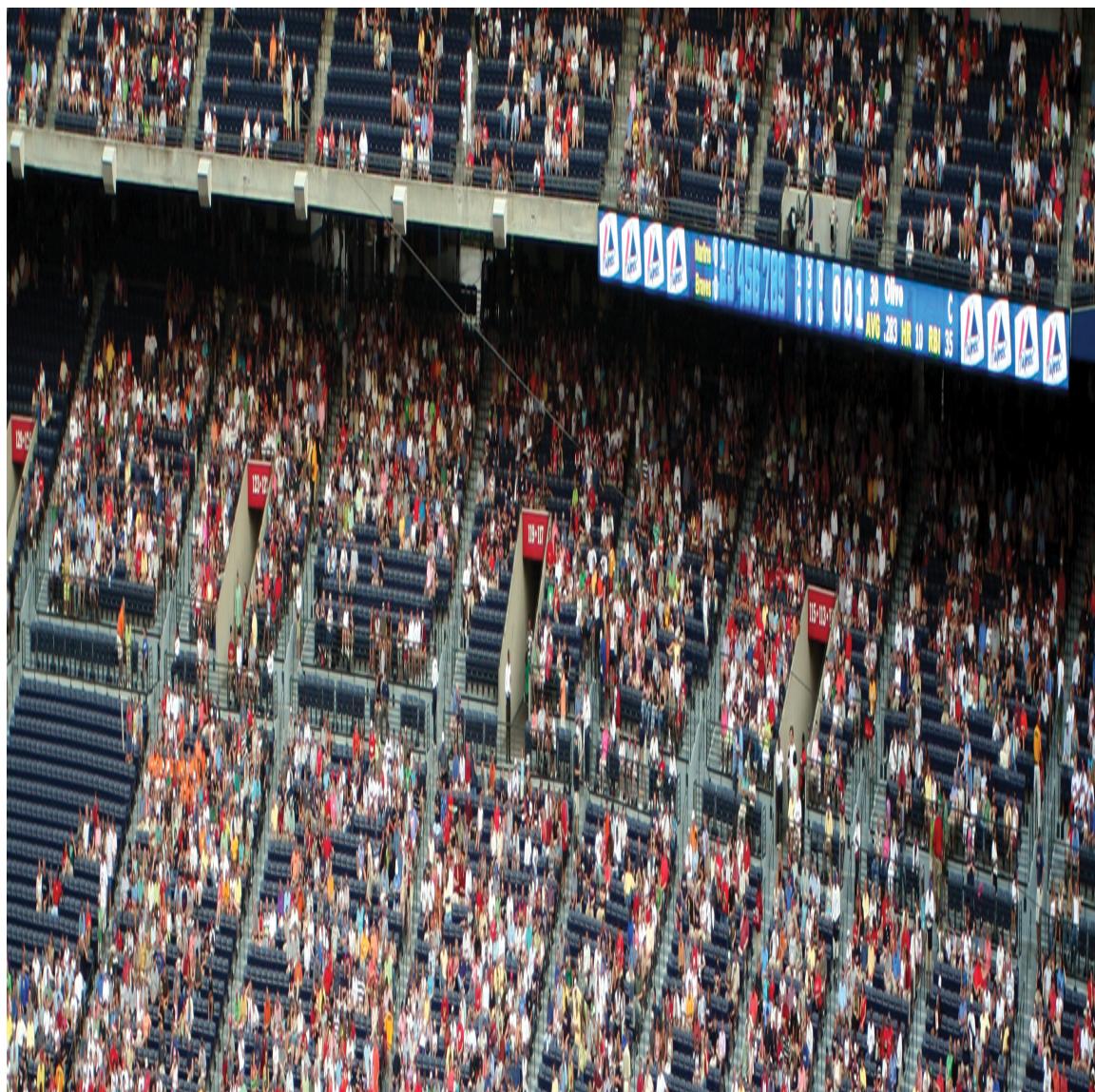
Finally, humans in the paleolithic began to work in groups. Other animals also work in groups and do so effectively, but humans are very much specialized in doing all the things we do together.

Putting Together the Mechanics

If we distill all the abstracted mechanics just discussed, rework them a bit, and combine them, we can use this information to create a game for modern humans that's based on hunting activities of paleolithic humans.

One game that is an abstraction of paleolithic hunting techniques that springs to mind is baseball (see [Figure 7.5](#)). It is highly unlikely that anyone

making the rules for baseball, rounders, or any related games thought about this, but they did not need to. Instead, what they most likely did was think about activities that were fun and compelling for them and their friends. They observed what kids were doing while out playing at the park. The activities that are characteristic of baseball and paleolithic hunting are fun and compelling because they are deeply embedded in the human mind as being fun and compelling. These feelings reach back a very long way into human development. So it's actually not surprising that we would make games that include the same mechanics that are involved in paleolithic activities. It's also unsurprising that many of the same mechanical themes pop up in games across cultures and across times.



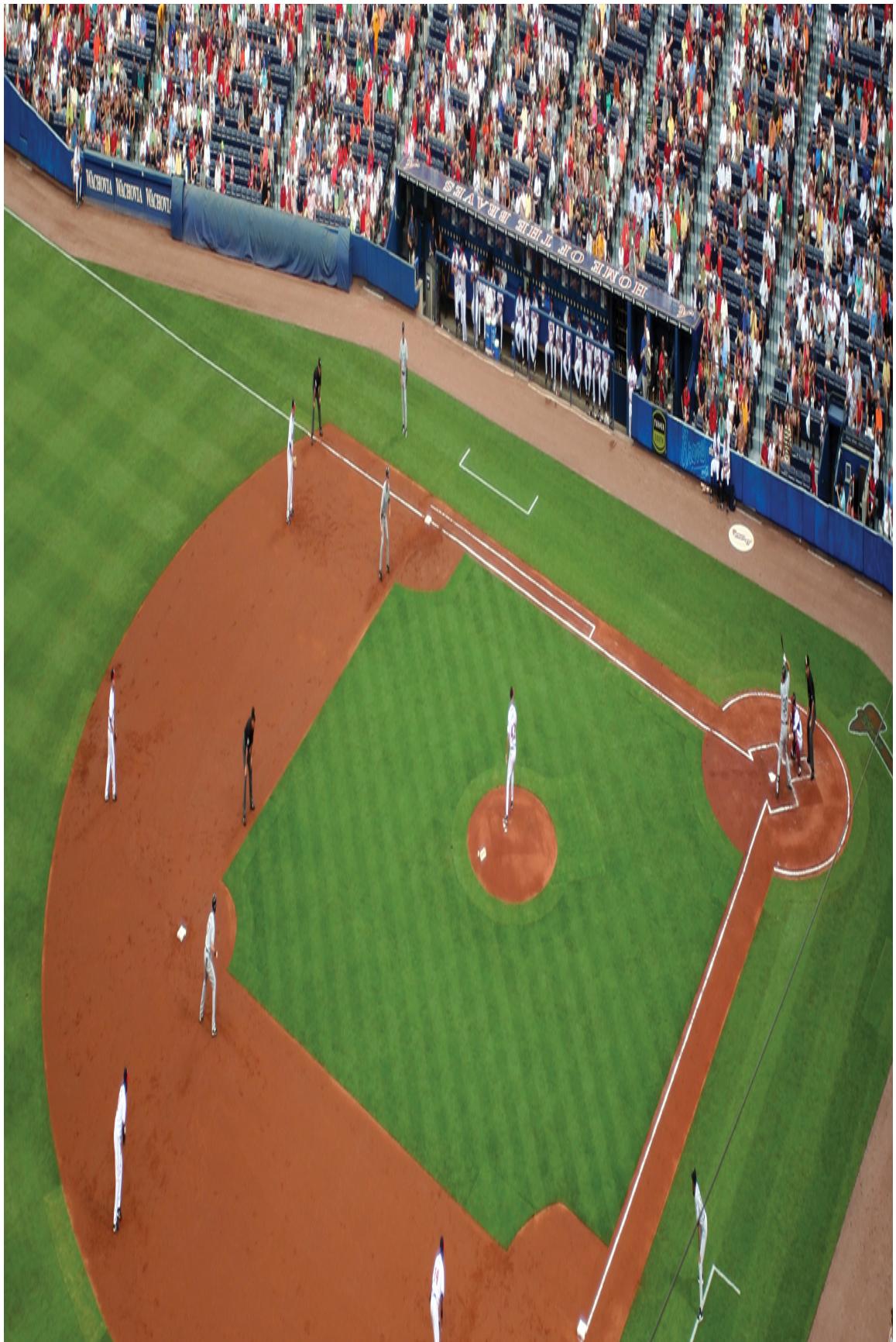




Figure 7.5 Baseball

Credit: Rob Marmion/Shutterstock

Distilling mechanics from life for use in a game does not need to be a conscious or purposeful activity. If you look closely at any game and really start studying the mechanics of the game, you can find analogs in the real world.

Story in Games

This is not a section about how to write stories for games. Writing stories for games is not in the domain of a system designer. It is certainly possible that a system designer might also be a narrative designer, but in that case, the single person would be fulfilling two different roles. Instead, this section is about how system designers create stories based on what someone else has written. Think about the following statement:

It's not the story we put in the game that matters; it's the story the players get out of the game that matters.

Consider what this statement actually means. It does not try to invalidate the work done by narrative designers but instead shows how we as system designers use a different method to get stories into our players' heads. Also, of important note, the two methods of creating stories are not mutually exclusive. Developers can spend a great deal of time creating fantastic backstories and character arcs. System designers want to use the game itself to enhance—or even completely create—the narrative for the players. Let's look at some examples.

Poker has absolutely no story built into the rules of the game. The “character backgrounds” are those of the players; none have been written. There is no narrative arc in the rules, and there is no background narrative, but many thousands of stories have come out of poker. Poker stories have been created for movies, books, and comics, and many stories have been told to friends about amazing poker games. Poker stories contain tales of heroes and villains, of triumphs so large as to change a person's life, and of defeats devastating enough to affect the economy of a town. How do all of these stories happen when no narrative designer crafted them? The systems of the game created fertile ground for those stories to grow. This is the domain of the modern system designers. As a system designer, it is your job to create possibilities for players to craft their own dramas within the bounds of the rules.

Think about any game you have played that has really stuck with you. There might have been some well-written narrative embedded in the game, but it is highly likely (especially if you are reading this book) that there was something more. Something happened to you that has not happened before, and it was amazing. It was hard to describe the event to friends, especially those who are not familiar with the game. This is a unique domain of

games. Movies play out for everyone the same way. Books have the same beginning, middle, and end, no matter who reads them. But games are interactive and provide unique experiences for their players. Further, they give players agency in what happens over the course of the game. No other medium does this.

As a system designer, you should be constantly thinking of mechanics and systems that allow your players to create their own stories. Keep this thought in mind as you read the following chapters. Consider how you can make your game in such a way that players can tell their own unique stories with it.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further games as abstractions of the real world:

- Analyze several old games and look at the mechanics involved. What are the analogs in those games to human activities?
- Go a step further and break down individual game mechanics in classic games and in your favorite modern games. What abstractions are being used for just those mechanics and not the game as a whole?
- Observe the world around you and pick out several mechanics from the real world that you have not commonly seen in games. Could you turn those mechanics into a game? Could you build those mechanics on to an existing game to improve it?
- Study several modern games and, instead of looking at the mechanics they included and how they were abstracted, think about what they excluded. What mechanics were left out of the games that are present in the real world? Why do you think the designers left out those features?

Chapter 8

Coming Up with Ideas

Coming up with ideas can be hard. Even more difficult is coming up with new ideas or new modifications for ideas. Harder still is coming up with new ideas that work well, can be made into reality, and actually please people. There is a common notion that some people are creative and others are not. Like so many other aspects of what people do, however, creativity is a skill; it is not just an innate talent. Much like learning to play an instrument, drive a car, play a sport, draw, dance, or sculpt, being creative is partly talent, but mostly it is a learned skill. People who appear to be very creative were likely born with some degree of natural talent, but it is guaranteed that they continued to develop their natural talent with hard work and a plan.

This chapter describes methods and practice exercises you can use to develop your skill at being creative, specifically in regard to coming up with new ideas. If you feel that you are not naturally creative, you should read this chapter carefully and work through the exercises in earnest. All game system designers are expected to be fountains of ideas. Your challenge should be to choose which of your overflowing ideas to use rather than to come up with a single idea. Don't worry if it feels difficult and unnatural at first; this is to be expected. Like any other learned skill, using your mind in new ways will, at first, seem unnatural. Just as with learning any other skill, the only way to get better is to practice. Keep working on these methods and doing the exercises until they start to feel comfortable or even natural. You will likely find that the ideas start coming easier and faster, eventually to the point that it is hard to stop them from popping out constantly.

Idea Buffet

Every game designer, and creative person in general, should develop and keep an idea buffet, which is basically a list of ideas and short descriptions. It can be game ideas, or characters, mechanics, weapons, vehicles, level setups, or anything else you want. Eventually, you will probably want to organize these ideas into loose categories to make finding the ideas easier when the buffet becomes very large.

I use the word *buffet* to describe this list because it should act much like a real-world buffet. The idea is to have more ideas laid out in front of you than you could possibly use. Many of the ideas will not go well together. But you can pop in to the idea buffet when you need inspiration and pick and choose the options that suit your needs at that moment.

Sample Idea Buffet

The following is a snippet of an idea buffet. As you can see in the following example, your idea buffet can include general ideas, bits of ideas, questions, and anything else you want. There is no strict format for how to record the ideas in your idea buffet. Note that most of the ideas are just brief descriptions. They are not meant to be full explanations or even be useful on their own. The only intention with lists like this is to help with memory and trigger new ideas when looking through the list. Typically these lists should be many pages long.

Game Ideas

- Bouncy physics-ish game. Jelly Stacker.
- Jenga with odd shapes or physics. Mobile or VR.
- “Military survival manual”–based game. Serious game.
- Knot-tying game based on real knots. VR? Race through tubes made of knots?
- Parimutuel betting: Monster fights, dog races, and so on.
- Decision-making game from the perspective of animals. Start with very low forms of life and very simple decisions. As the forms of life get more complicated, so do the decisions and controls.
- Choose your own adventure in a puzzle game.

- Slide puzzle combined with a platformer.
- Real-time rock paper scissors.

Mechanics

- Mixing colors to match an enemy color. Lasers?
- Traveling salesperson puzzle in an RPG.
- Bad camera controls is the challenge.
- Glass breaking in unpredictable ways to be used by a player.
- Giving a dog commands through a simple set of symbols.
- An enemy that takes damage only if you don't hit him.
- Play as an NPC in someone else's game.

Running a Brainstorming Session

Brainstorming can take many forms, and it can be used in a very particular way in making games. This section discusses how to run a brainstorming session and methods you can use to dislodge ideas that are not flowing freely.

Having Goals

It is important to have goals for every brainstorming session. These goals should be written down and shown to everyone in the group prior to the session. They should be simple, one-phrase goals that are easy to understand but don't restrict the creative thought process. Here are a few examples of goals that system designers might have for a brainstorming session:

- New character classes that complement our old ones
- New mechanics for vehicle interactions with the world
- Better weapon/armor combinations
- Environment mechanics that set our game apart from others

- How to use the new controller for our target console

Once you have a short list of goals (for example, one to three goals), you can get started with your brainstorming session.

Note

Goals should not contain topics like “solve this problem” or “make a final decision.” Those topics are not in the domain of a brainstorming session. A well-run brainstorming session should be for the singular purpose of generating ideas, not implementing, critiquing, or judging ideas.

Gathering the Troops

One person should be the moderator for a brainstorming session. While you can brainstorm all by yourself—and it is often done that way—brainstorming as a group is even better. People in the session should be familiar with the problem but need not be experts. New perspectives are a great thing. Two people is enough for a group brainstorming session, and five is about the maximum number of participants for a useful session. Having more than five people in a session makes it hard for everyone to participate and creates issues with coordination. When more than five people need to be involved, it would be better to run separate sessions with smaller groups than to run one big one.

It is important for everyone involved to have time completely cleared for the session. It is also important to eliminate all distractions and interruptions. Ringing cell phones are a complete mood killer in a brainstorming session. No one should be on any screen or other media for the duration of the session. The only exception could be for a note taker to have a document open for typing session notes. Distractions destroy creativity.

It's best to have a small set of ground rules for the session written on a white board or another viewable area. Here is an example of such a list:

- Session is 1 hour long, from 2 p.m. to 3 p.m.
- No screens, headphones, or calls.
- Door is shut; no one leaves or enters during the session.
- No ideas are bad ideas.
- No person is wrong for coming up with new ideas.
- Have fun and be silly.
- Participate.
- Goals: (list session goals).

Giving Yourself a Block of Time

A brainstorming session should have defined start and end times. One hour is a good amount of time for a new group. As you continue sessions, the group will gain more endurance—but not indefinitely. More than 2 hours is probably too long. Under 30 minutes is too short to get into the proper mindset for most groups. The start time needs to be firm. Everyone in the group needs to be committed. It badly hurts the process if one person shows up partway into a brainstorming session. The latecomer is prone to repeat ideas, fail to understand the mood of the room, or want to impose their own current mood on the room.

The end time should be firm but not inflexible. If ideas are really flowing and progress is spilling out, then you can keep going for a little bit. If the mood of the room is bad, and there is more negativity than creativity, then the session might need to end early. If this happens, the issue needs to be addressed with the group before the next session. Leave some designated time at the end of the session for note gathering and for planning future steps. If people try to start planning during the session (which is very common), the moderator should gently remind them that there will be time for planning at the end.

Don't Accept the First Answer

One of the biggest traps in brainstorming sessions and the creative process in general is the temptation to accept the first answer. Let's look at a quick

example. Say that you have been given some red buttons, blue buttons, a piece of grid paper, and two dice (see [Figure 8.1](#)). The goal for the session is to come up with game ideas that use these objects. Think about it for a few seconds before you continue reading.

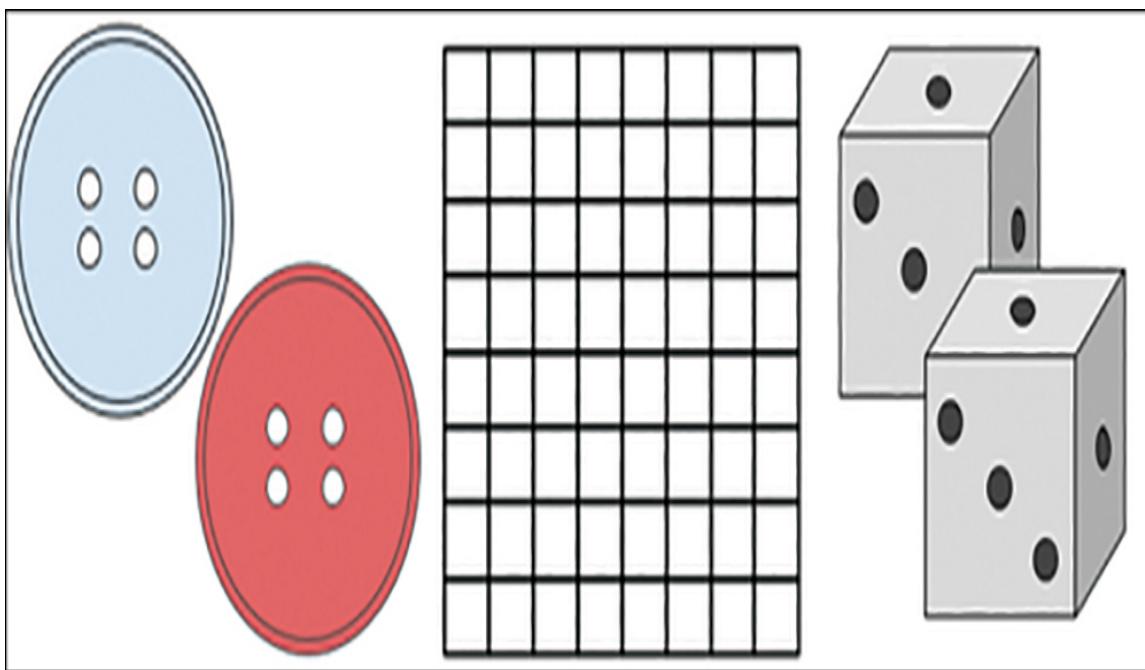


Figure 8.1 Game Pieces

Your first answer might be to create a game like checkers but with random dice movement. This simple challenge has been issued to countless hundreds of prospective designers, and that is by far the most common answer given. However, in a brainstorming session where the goal is to come up with new ideas, you don't want to go with the obvious answer. There is a big temptation in brainstorming sessions to latch on to the first idea that someone blurts out and fixate on it. You need to fight that urge. The moderator should say “great idea, keep going” to every answer.

You can try this now with the buttons and dice example: Write down your first idea and then come up with 10 more. Make sure those 10 answers are completely different from each other. For example, make sure you should have several ideas where the buttons and grid are not used together, several where combinations of items are used together in a novel way, and several

that make no logical sense at all. Most of these ideas will be bad, but that's fine. You don't need tons of ideas in the end but are mining for a precious few among all the rest.

In some cases, the first idea suggested may end up being a diamond. That's great! The additional time coming up with more answers is still not wasted. Getting a great idea out of a session is a win no matter what, and you can feel more confident that it's the best when you can compare it to all the others. In brainstorming, it doesn't matter whether the best idea is first, fifth, or one hundredth. What matters is finding the best idea you can.

Avoiding Criticism

The goal during a brainstorming session is always to add ideas and never to eliminate them. As technical developers who want to produce quality products, it is in the game developers' nature to be very critical. When doing production work, especially near the end of a project, this is a good thing. During a brainstorming session, however, it is a disaster. Criticism forces people into a defensive mindset, where they are thinking about defending a single idea more than coming up with new ideas. Hearing criticism also discourages participation, which is crucial to success.

It is important for a brainstorming session moderator to keep a close eye out for criticism and address it. Since combativeness is also destructive to the creative process, one good way to address criticism when it's spotted is to say something like "I see you have found a potential flaw, but let's focus on a solution." Additional good phrases are "That idea may not work, but what's an idea that would?" and "Okay, how could that idea be even better?" By being obsessively positive and guiding others toward positivity, the moderator can help the session produce more ideas of better quality.

Keeping on Topic (Kind Of)

If the goal of a brainstorming session is to come up with character ideas, but the session produces only new game ideas, the session is of limited usefulness. However, staying too focused can stifle creativity. A moderator needs to let the group flow and bounce ideas without wandering too far away. Think of it like herding puppies through a field back to home. The

moderator wants to get them all home but also wants to let them run and play in the field to get all that energy out. The moderator should let participants wander off a bit and let them be silly and riff off each other but should then gently bring them back. The moderator should be sure not to be critical or tell anyone they are doing it wrong but gently steer participants back toward the goal. A statement like “Let’s get back on track” can be stifling. Instead, the moderator could say something like, “I hear some very creative ideas here. How can we relate that all back to our goal?”

Capturing the Creativity

A brainstorming group should be capturing ideas as they come out, using a white board, paper, or a computer. It’s best if all participants can see the ideas as existing ideas often spawn more ideas—and that is just what you want. Don’t worry about typos or organization; instead, focus on getting ideas recorded. In a good session, ideas will be coming out fast and often, and it is important not to slow people down or stop them from producing new ideas. If needed, have more than one person jot down ideas. Plan for time near the end of the session to do cleanup. Building in this cleanup time helps very organized types of people relax a bit. When you see someone erasing and rewriting ideas during the session, gently remind them that time at the end of the session is allocated to cleanup, so they need not do it quite yet.

Don’t erase anything as you go. If, at the end of a session, you find some ideas that are gibberish, or if everyone in the meeting has now forgotten what those ideas mean, go ahead and remove those. But any ideas, bad or good, that you can still understand should be recorded.

After the session, you can organize all of the generated ideas into an idea buffet.

Keeping Expectations Reasonable

Brainstorming sessions are tough. Generating new ideas is very hard. Don’t expect that one session will produce every answer you need—or even any answer. Try not to get frustrated. With patience and practice, your results will improve.

Percolating

When a brainstorming session is over, there is still more to do. At this point, you should have rivers of good, creative, silly, or plain bad ideas flowing through your head. In the hours and days to come, your mind will continue to churn on all these ideas and new, possibly great ideas will pop out when you least expect them. Be ready for this by having a recording device handy. It is not uncommon for designers to remain in this phase once they enter it. You can generate ideas and solutions at any time. Your creative mind doesn't care about your schedule or whether you're in a brainstorming session; it does what it wants when it wants. After a brainstorming session, your job is to be vigilant in waiting for those ideas to surface and then to capture them.

Methods to Force Creativity

Creativity does not always come easily, and sometimes (maybe even often) it stalls completely. If, as a moderator or a brainstorming session participant, you find a session prematurely slowing down, or if you are completely stuck, you can reach for a few methods to break through the barrier and get the ideas flowing again. Note that the methods discussed in the following sections are best used when the natural ebb and flow of a session has stalled. In a good session, none of them will be needed at all, and the moderator will just direct traffic and keep the group moving forward.

Bad Storming

This method is a great session opener. People tend to be reluctant to come up with bad ideas, which makes them hesitant to come up with any ideas at all—for fear that the ideas will be considered bad. To combat that, force the issue by bad storming. The goal of this exercise is for everyone in the session to purposefully come up with a bad idea.

For example, here's a very bad idea for a game with the buttons, grid, and dice example: The first of two player sticks two red buttons up in her nose and then sneezes as hard as possible, shooting both objects onto the grid paper. Both players then secretly set the dice to a wager. The second player

then stuffs the blue buttons up his nose and sneezes, attempting to shoot the buttons out, hit the red buttons, and knock the red buttons off the grid. If the second player succeeds, he wins the amount of his wager; if not, he loses the amount the first player wagered. Is this a good idea for a game? Absolutely not. Did it make you chuckle? Maybe. Did it make you think about your items into a completely different light? You bet!

Requiring everyone to come up with bad ideas tends to get more creativity going because the participants don't have to obey any inner preconceptions about "right" answers. This also gets people loosened up. After they have had the courage to say one really bad idea, their fear of saying more decreases or even vanishes completely.

Jokes

Encourage jokes and joke ideas. This technique is an extension of bad storming. When an idea is phrased as a joke, there is less pressure for it to be a good idea. As a tension breaker, have participants start saying the idea with a phrase like "I've got a crazy idea. Here goes!" While the joke idea may not be a winner, it might inspire other ideas that are more serious. In professional settings, many times the best answer springs out of a joke. Jokes also keep the mood light and help the team flow together. While excellent craftsmanship can come from somber discipline, fun and creative ideas cannot.

Building Blocks

One of the best methods to start generating ideas, especially for system designers, is to break down objects or concepts into their smallest pieces. We will discuss this technique further in [Chapter 9, "Attributes: Creating and Quantifying Life,"](#) but it is noted here because it is also an excellent exercise to perform in a brainstorming session. The first phase involves breaking down an idea into the smallest-sized components possible. This will likely generate a large list. The next phase involves changing one of those components to something else and then guessing what would happen to the system as a whole with that one change.

For example, think about a toy wagon that has a metal bed, four wheels, two axles, a handle, and a handle hinge. What would happen if you were to change one of those aspects into something else? For example, say that you give the wagon skis instead of wheels. Now you have a sled. What if you exchange the metal bed for a boiling vat of soup? Now you don't have a wagon but a mobile food source for a frozen tundra. By changing the smallest building blocks of a concept, you can dramatically change the concept as a whole. Again, none of the ideas generated in this way may be winners, but the thought process of breaking down an object into components may produce a winner.

Future Past

The future past method is best used when you have a specific problem that needs a solution. With this technique, you state the problem, and then someone blurts out a vague, random, or even purposefully wrong solution. The exercise then is not to determine whether the solution works but instead to imagine that you are in the future, and it turns out that the solution did work. Now you need to figure out how it worked. People tend to think too much about *whether* ideas will work or not, which keeps them from focusing on *how* ideas can work.

The future past method is particularly powerful and can be applied more broadly to problem solving in many aspects. Let's say you have a goal of growing your company from a garage-sized startup to a prosperous indie studio. One way to get started with solving that problem would be to imagine it is the future, and you now work for that thriving studio. How did you get there? Think about the steps you might have taken to go from where you were to where you wanted to be. Reframing the question from "Is this possible?" to "How is this going to be possible?" can dramatically change your insights on a problem.

Iterative Stepping

In an iterative stepping exercise, you start with a known idea and then consider the question "What's close to that?" In a group setting, you would hand off the question from one person to the next. It's best not to dwell on

the iterations but to blurt out the first thing that comes to mind. This technique can meander into odd places, but it produces interesting results.

For example, say that you start with the idea horse. Your iterative stepping could go something like this:

- 1. Horse power**
- 2. Muscle car**
- 3. Mussels (the clam-like critters)**
- 4. Seafood pasta**
- 5. Dinner with mom**
- 6. Family drama**
- 7. Family comedy**
- 8. TV**
- 9. Ratings**

You could carry on indefinitely.

Iterative stepping is also a great method to use when you want to break free from tropes. If the group is stuck on a tired idea, then iterating away can knock that loose, starting with small changes from the idea and eventually moving to fresh new ideas. Let's look at another example that is more closely related to games. We can start with the tired trope of breaking pots to find treasure. The iterative stepping might go like this:

- 1. Smoking pot to find treasure**
- 2. Smoking beef jerky to find treasure**
- 3. Smoking beef jerky to find yourself**
- 4. Finding treasure to find yourself**

- 5.** Questioning why you need treasure at all
- 6.** Questioning why you break pots
- 7.** Is it morally okay for a hero to break a peasant's pots?
- 8.** Punishment and repayment for broken pots
- 9.** Balancing broken pot punishment with treasure pot reward

This is a very silly list, but that's okay! There are many interesting interactions listed here. Maybe none of them will be the idea you are looking for, but the clue to what you want may very well be in there somewhere. At the very least, you are not defaulting to a boring old trope.

Halfway Between

You can use the halfway between method when you have two very different ideas. It's sort of the opposite of the iterative steps. In this exercise, the goal is not to diverge from a point to somewhere unknown but to find an imaginary, nonexistent point between two known ideas. When you have a pair of wildly differing concepts among the ideas generated in your brainstorming session, you can consider the mismatched pair and try to figure out what would be halfway between the two ideas.

Here are a few example of pairs of ideas that you can practice with:

- A first-person shooter and a farming game
- A race car and a racehorse
- A sword and a shield
- A crafting system and a damage system

Try it! If needed, solve each pair more than once with different answers. You could even have each participant write down a different answer and then have everyone reveal their answers at the same time to see how much variation there is. It is likely that new or interesting ideas will come up as you consider what might be halfway between each pair.

As with all the other methods listed in this chapter, the deeper you dig using the halfway between technique, the more likely you are to find a diamond of an idea.

Opposite Of

With the opposite of method, you start with a topic and try to come up with the opposite of that idea. You then take the new idea and come up with the opposite of that idea, with the caveat that you can't choose the first item again. This is challenging to do, but it yields some very interesting results. In function, it is similar to translating some text though several languages (using Google Translate, for example) before translating it back to English. You rarely end up with what you started with; the results are often funny and also often very different and interesting.

For example, say that you start using opposite of with the idea of a gun. Here's what you might get in four passes:

- 1.** Flower (peace instead of war)
- 2.** Sheep (plant eater instead of plant)
- 3.** Nails (spikey instead of soft)
- 4.** Hammer (drives nails instead of being driven)

You could keep going as long as you like.

Again, this technique generates a lot more ore than diamonds, but that is not what matters. Getting that one diamond by any means necessary is what matters.

Random Connections

Classic random word association is another technique that can help you get unstuck during a brainstorming session. Start with one of your generated ideas or topics and then pick a random word by using a random word generator or an old-fashioned dictionary. Now list the ways those ideas are

similar and how they are different. Get specific. This technique doesn't always yield diamonds, but it often produces surprising results.

Stream of Consciousness Writing

If you're still having trouble generating ideas after using the methods described, or if you just want more ideas (there are never too many), you can try stream of consciousness writing. To do so, get pencil and paper and set a timer for 2 minutes. This might seem like a very short amount of time, but when you are in the middle of it, it will feel much longer. When the timer begins to run, start writing. You can write anything. Ideas are best, but nothing is off limits. The only rule is that you must be actively writing for the entire 120 seconds, with no pauses, no breaks, no corrections, no reflection, and no talking. Just write for a solid 2 minutes. If no thoughts are coming to your head, write that down. Try to change tracks in your mind as often as possible and don't get stuck writing lists of items that are all related. By the time the timer is finished, hopefully a few nuggets have popped out. You can expect that most of your writing will not produce anything useful, but that's fine. When you are mining for diamonds, you may need to dig through a lot of ore.

Note

All the methods listed in this section can be useful by helping break up a stalled session and getting people thinking of ideas again. Keep a list of these methods handy. If you notice that a session has stalled and needs a nudge, you can try one of these methods to get it going again. However, it is also important to use these methods with some constraint. You do not want your brainstorming session to feel like a bunch of party games meant for killing time rather than generating ideas. If a session is going well and ideas are flowing right from the start, sit back, gently guide the flow when needed, and enjoy. In an experienced team, the moderator may not need to do much at all, and this is a great thing!

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore brainstorming methods:

- Perform a solo brainstorming session. Pick a narrow topic, such as a game mechanic you are working on, and go through all the steps of brainstorming to see what you get out of it.
- Perform a group brainstorming session. Find a few friends and go through the method outlined in this chapter. You can pick any topic, but a game-related one will be most beneficial. In many brainstorming sessions, it's not necessary to go through all the steps and try all of the forced creativity activities, but for your practice session, do so anyway. This will allow you to get familiar with each step and method, without the pressure of needing to produce any results.

Chapter 9

Attributes: Creating and Quantifying Life

One of the most common early tasks system designers perform is to create attributes for game objects. We can define *game object* as any individual object in a game that does something. It could be something as simple as a block of ground that has a texture and collision, all the way up to a character that has a vast array of attributes, animations, 3D meshes, and more.

Mechanics Versus Attributes

Game objects can be defined using attributes and mechanics. Simply put, *mechanics* are rules that govern usage, and *attributes* are values that influence those rules. In a practical sense for video games, mechanics require some amount of encoding, whereas attributes are represented as values (data). In a board game, mechanics require rules, or combinations of rules, and attributes are represented as values.

Let's look at a simple analogy that illustrates the relationship between mechanics and attributes. In a game, data objects and attributes function as nouns function in writing, and mechanics function as verbs. We can take this analogy further and say that microsystems function much as written sentences, macrosystems function as chapters, and games function as stories. The basic building blocks always come back to the nouns and verbs or, in the case of games, data and mechanics.

Think of the game chess. You could say that each square on the board is an object. There are rules about what pieces can be placed on each square to begin the game. In addition, each piece is also an object. All of these objects have their own mechanics and attributes.

A king and a queen in chess both have the mechanic that they can move in any of eight directions on a turn. However, the king may only move one space if the path is clear, whereas the queen may move up to eight spaces if the path is clear. So we could say that the king has a movement attribute of 1, with the mechanic being that the space must be clear, and the queen has a movement attribute of 8, with the same movement mechanic. The king also has additional mechanics regarding check and checkmate, but neither of these has an associated attribute; each is just a simple mechanic.

Let's now consider a slightly more complex example: the classic arcade game *Galaga*. A player's spaceship has several mechanics: It can move side to side, it can fire projectiles at enemy ships, and it takes damage when it collides with an enemy ship or projectile. A player's spaceship also has a few attributes that drive these mechanics, including moving speed, firing rate, and hit points. Without changing the mechanics, you can dramatically affect game play by manipulating the attributes. Imagine what would happen if you slowed down the player character (the PC) and allowed it to move only half as fast as before. How much harder would the game be? Now imagine if the PC could shoot 10 times as fast as usual. Each of these attribute changes would not require a change in the rules of the game but would have a dramatic impact on game play.

Listing Attributes

For a very simple game, a game object may have only one or even no attributes. Checkers, for example, has no functional attributes since all of the pieces behave in exactly the same manner. For more complicated games, game objects may have dozens of attributes—some seen by the player, some hidden, and some derived with calculations.

The first step in creating attributes for an object is to take a very close look at the object you want to define. This can be done even before you have mechanics and might inform what mechanics you want. You need to think about all the facets of that object that might be different from instance to instance.

Let's look at an example of creating attributes for an object—in this case, the very generic “person” shown in [Figure 9.1](#).



Figure 9.1 A generic human character

Initial Brainstorming

The first step in coming up with a list of attributes is to just write down what comes to mind quickly. What attributes can people have that change among instances of people? The following are some very obvious attributes you might come up with off the top of your head:

- Height
- Weight
- Waist size

- Wingspan
- Shoe size
- Hat size
- Blood pressure
- Heart rate
- Age
- Body fat percentage
- Strength
- Dexterity
- Speed
- Intelligence
- Wealth
- Dancing ability

Blue-Sky Brainstorming

After you've listed the very obvious attributes—the ones that spring quickly to mind—you enter the “blue sky” phase of attribute brainstorming. During this phase, you should be open to any and all contributions, even silly ones. In this phase, you might add the following attributes to your list:

- Magic aptitude
- Magnetic personality
- Humor
- Stench
- Clumsiness
- Luck

In this phase of creating attributes, you want to come up with as many attributes as possible. Quantity is more important than quality. A little later

on, you can weed out what you don't want or what does not fit your game. While going through the process of defining objects, people tend to get more closed-minded and focused. It is therefore especially important to take advantage of this very early blue-sky phase to be as creative and open to new ideas as possible. You can use the techniques covered in [Chapter 8](#), “[Coming Up with Ideas](#),” during this phase of attribute development.

Researching Attributes

Once you have a large list of both basic attributes and more creative attributes, you can move on to the next phase of creation, which is research. It is highly recommended to do this phase after the creative phase because research will form a construct in your mind about what these objects “should be,” based on what others have observed. While it is good to be grounded in reality and to have a working knowledge of objects similar to your own object, expectations tend to stifle creativity. So you should do research only after you have exhausted your own internal creative well.

For a generic person object, you would want to research two things: attributes that define people in the real world and attributes that others have used to define people in previous games.

Researching Real-World Attributes

The Internet is a valuable resource for this type of research. For example, when researching a person object, you might look at sites that provide statistics for individual people, such as exercise and diet sites, medical reference sites, education sites, and demographic studies. At this point, you are not concerned with the results obtained by any of those sources; instead, you are concerned with what attributes they are studying. In addition, you probably don't need attributes as detailed as the ones you might find in medical textbooks, so you can simplify them.

When you research attributes from real life, you might come up with the following:

- Blood oxygen level
- Bench press weight

- Biceps curl weight
- Squat weight
- Pancreatic function
- Kidney function
- Stomach pH
- Neurons in the brain
- Hearing ability
- 100-meter dash time
- Marathon time
- Baseball pitching speed
- Football kicking distance
- Eyeglasses prescription

This list could get absolutely enormous. As a system designer, you can choose the level to which you want to simplify and how much you want to include. A good rule of thumb is to include at least twice as much on this list as you think you might need. This will create a large pool of choices for you to pick from when you decide on a final list.

Researching Attributes in Past Games

Next, you can research past games to see what attributes they use to define a person; when you find any that you have not yet included on your list, write them down. For this research, you should start with games in similar genres that include objects that are the same as or similar to objects that you are creating. This step intentionally occurs late in the brainstorming process because it is the step that is most likely to narrow your creative focus. It is easy to copy what others have done and very tempting to stop there. In doing this step late in the process, you ensure that you have already come up with new and creative ideas before delving deep into what others have done.

When you research attributes from games, you might list the following attributes for your generic person:

- Constitution
- Wisdom
- Acrobatics
- Arcana
- Insight
- Intimidation
- Spirit
- Melee
- Ranged accuracy
- Leadership
- Movement rate
- Encumbrance

Referring to Your Own Personal Attribute Bank

The final step in the process of listing attributes is to refer to your own personal “attribute bank.” When you are new at listing attributes, you won’t have such a bank yet, but it will grow quickly. Every system designer, whether formally or informally, has a bank of attributes that they refer to whenever creating new objects. This is a list of attributes that the designer has used multiple times for different objects across many games. Since there are no specific criteria for what attributes go into your personal bank, it is likely that most of them will not be applicable to any given new object you are trying to define. However, some of them may be, and it is a useful final step to refer to an attribute bank to make sure that you have not forgotten any attributes that you know work well.

Note

An attribute bank is similar to an idea buffet and could even be included in the same document. However, the attribute bank includes only attributes and not ideas in general.

Defining an Attribute

Notice that in all of the attributes lists so far in this chapter, each attribute is a single word or a very short phrase. This is what you want when coming up with new ideas, as trying to define every attribute during the brainstorming phase would slow down the process and likely stifle creativity. To make attributes useful, however, you must go further and define them.

So how do you define an attribute? For example, what is strength? This seems like a pretty straightforward question. By looking at different people, you can often guess whether one person is stronger than another. When you get into the realm of designing game mechanics and attributes, however, you must be exact. For the sake of making a game, you must also abstract the concept because a real-world attribute is likely to be too complex to contain in a single value.

As of this writing, the world bench press record, set by Kirill Sarychev, is 738.5 pounds. Obviously, Sarychev is very strong. The world squat record, set by Jonas Rantanen, is 1,268 pounds. Obviously, Rantanen is also very strong. But which one is stronger? Sarychev can't squat as much weight as Rantanen, and Rantanen can't bench press as much as Sarychev. So there is not one simple attribute that measures strength. In a game, you need to decide what exactly the attribute strength means.

To flesh out the definition of the attribute strength for a game, you need to know what the attribute is being used for in the game. Let's say that in your game, characters are doing a caber toss (see [Figure 9.2](#)). A caber tosser needs strong legs but is even more reliant on upper body strength. When you define the strength attribute, you need to define it specifically as it pertains to the game. In this example, you might define strength as follows:

The maximum weight of a caber that the player can successfully flip.



Figure 9.2 Caber toss

Credit: Vineyard Perspective/Shutterstock

This definition is fairly short because the game is very simple and mechanically limited in scope. With a more complex game involving more interactions, you would need to further define the attribute and how it interacts with the mechanics of the game. In this simple game, however, you want to keep the definition of the attribute as simple as possible.

You can now use your definition of strength to determine which of two characters is strongest. For example, if you wanted to compare Sarychev and Rantanen using this definition of strength, you would consider that although both of them are very strong, the one with the stronger upper body, Sarychev, would be more successful in a caber toss. With this in mind, you might give Rantanen a strength rating in your game of 99 out of 100 (again, he is very strong), but you would give Kirill a 100 out of 100 because he is the strongest at the designated game mechanic. If the game were changed to focus on pushing heavy things with the legs, these players' attribute scores would likely be flipped.

Considerations When Defining an Attribute

Through your brainstorming and research, you might discover dozens or even hundreds of attributes that could be assigned to a data object. Early in the process of listing attributes, you want to come up with as many attributes as possible. When the list of possibilities is fleshed out, it's time to start being more selective about which attributes are actually going into the game.

- **Every attribute should have a use.** Every chosen attribute should do something in the game. In [Chapter 7](#), “Distilling Life into Systems,” I said that games are abstractions of life. Attributes are also abstractions and simplifications of life. Therefore, many attributes that could be listed for a given object are unneeded for the specific game being made.

A simple test to check attributes is to consider what a specific attribute will be used for in your specific game. For example, if you are making a car racing game, you could assign an attribute for the volume of the trunk, but would that ever be used? Probably not, so it is not needed in the game and should not be included. This is the first and most

important test to apply to all proposed game object attributes. The intended mechanics usage of the attribute should also be formally listed in the description of the attribute.

- **Attributes should be intimately related to mechanics.** Every attribute should have related mechanics. For instance, if you have a character with a strength attribute, you need to relate that attribute to the mechanics that will use it, which might be combat, encumbrance (that is, ability to move heavy loads), endurance, or resistance to damage. When narrowing your list of possible attributes down to the final list, it's a good practice to tie each proposed attribute to the mechanics that will use that attribute in the game. In fact, you can often generate more attributes by thinking about what attributes you need to drive your mechanics.

Grouping Attributes

Once you have finished listing and defining attributes, you are likely to notice that many of the definitions and attributes are redundant, and you don't need them all for your game. The next step in the process is not to eliminate the attributes you don't need but rather to group them. This way, you get a much clearer idea of what your abstracted attributes actually mean.

For a very simple action game, you might want to deal with very few attributes to make it easier for the player to comprehend the game. Say that you decide to use just the following attributes:

- Strength
- Dexterity
- Health
- Speed

You can now go through all of your earlier research and decide which attributes will be grouped. Your list might come out something like this:

- Strength

- Bench press weight
- Biceps curl weight
- Squat weight
- Baseball pitching speed
- Football kicking distance
- Melee
- Encumbrance
- Dexterity
 - Hearing ability
 - Eyeglass prescription
 - Acrobatics
 - Ranged accuracy
- Health
 - Blood pressure
 - Heart rate
 - Blood oxygen level
 - Pancreatic function
 - Kidney function
 - Stomach pH
- Speed
 - 100-meter dash time
 - Marathon time
 - Movement rate
- Unused
 - Height
 - Weight

- Waist size
- Wingspan
- Shoe size
- Hat size
- Age
- Body fat percentage
- Intelligence
- Wealth
- Magic aptitude
- Magnetic personality
- Humor
- Stench
- Clumsiness
- Luck
- Neurons in the brain
- Constitution
- Wisdom
- Arcana
- Insight
- Intimidation
- Spirit
- Leadership

Note that the attributes listed under “unused” are not bad attributes, but they don’t fit our current needs. It would be a good idea to put them in your attribute bank for later use in other games.

Something else to notice in this final list is that the attribute melee (that is, the ability to perform hand-to-hand combat) is listed in the strength category, and the attribute acrobatics (that is, the ability to jump, flip, and otherwise move) is in the dexterity category. In the real world, these two attributes are actually fairly closely related. Many melee moves require acrobatic-like ability, and many acrobatic moves are very similar to those you would see in a melee combat. So, why are they grouped into two separate categories? When creating and organizing object attributes, you often encounter conflicts like this. The real world is very complex, and often even simple activities can be categorized in multiple ways. As a system designer, you often have to make calls about how attributes are categorized. When making a game, you are creating your own little universe of rules, and you are the deity of that universe. If you were to try to be perfectly accurate in organizing attributes, you would have an endless task that would make no practical sense for your game. As a system designer, you have to make calls on how to abstract elements of life into a game.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the process of creating attributes for games:

- Analyze the attributes of several games in a similar genre, such as RPGs or FPSs. List every attribute for every character and item. Which attributes show up the most? For unique attributes, why do you think the designers felt the game needed them?
- Do a practice brainstorming session to create attributes for some game objects. This could be any objects of your choosing, but here are a few examples to get you started: a cow, a stick, a spaceship, and a banana.
- Start your own attribute bank with the attributes you found in the preceding exercise. Write a short definition of each attribute and how various mechanics may be able to use it in a game. Keep this bank in a place you can easily access from now on and continue to expand it as you come up with more ideas.

Chapter 10

Organizing Data in Spreadsheets

As you create attributes for your game objects, you need to organize and eventually analyze them. The best place to do this is in a spreadsheet. A very small board game may not require the use of a spreadsheet, but even slightly more sophisticated games can almost always benefit from having their data organized and analyzed in a spreadsheet. By following some best practices when creating and using a new workbook, you can make the process of creating game data much less painful. This chapter discusses several of these best practices.

Create a Spreadsheet to Be Read by an Outsider

One of the biggest mistakes that new designers make when they create a new spreadsheet workbook is to dive in headfirst, making large lists of characters, weapons, vehicles, or other objects without thinking about how the spreadsheet will be viewed by others (or even themselves in future). By taking a few moments to plan what you want to do with a spreadsheet and decide how it will be used, you can save many hours later sorting out masses of data that have become a jumbled mess.

One way to know that you are on the right track in making a usable workbook is to think of it from the perspective of an outsider. If you brought on someone new to the team, how would they view the workbook and data it contains? Would they understand what data is in it? Would they understand why the attribute numbers are what they are? By taking the time to organize your data as you go, you can reduce the confusion others experience when they look at the spreadsheet. When someone is seeing your spreadsheet for the first time, you don't want them to see a list of unlabeled names and numbers or, even worse, multiple data tables for different kinds of objects patched together on a single sheet.

Avoid Typing Numbers

When possible, use references in spreadsheets and avoid typing in numbers. For example, let's say that in your current game, there is a gravity setting. Each time you refer to any momentum of a character jumping, or a projectile traveling, or a vehicle moving, it is affected by the gravity setting. Even if the variable for gravity is a simple single-digit number that is easy to remember, you shouldn't type in that number for all the listed calculations. Instead, put the variable in a labeled cell somewhere, such as on a reference sheet (see the section “[Reference Sheet](#),” later in this chapter). Then, everywhere that variable is needed, refer to the variable by name. By doing this, you can guarantee that if the number associated with the variable changes, all the equations that use it will be updated automatically. Also, don't fool yourself into thinking that number will never change. Through the process of testing and iterating, pretty much any number in any game may have to change for some reason or another. It's better to be prepared to change it and not need to than to need to change it and not be prepared to.

In addition, avoid doing calculations—even simple ones—in your head. For example, say that you have a goblin who has a strength of 2. You also have an ork, whose strength should be double that of the goblin. It might be tempting to type 4 in as the strength attribute for the ork, but don't do it! Instead, take the extra few seconds to write a reference for the goblin strength attribute and a formula that multiplies goblin strength by 2. If possible, go a step further and create a cell that contains an “Ork to goblin strength multiplier” with the value 2.

Then you can multiply the contents of the multiplier value cell by the goblin strength to arrive at the ork strength. Taking the extra time to use references will ultimately make the process of revising data dramatically faster and less prone to error.

When you do a first pass on data, it is highly unlikely that it will be the final pass. A system designer should expect to revise and iterate data five or more times; revising and iterating over a dozen times is quite common. If you have to go back into your data and find every variable that is affected by new updates, you will face a cascade of numbers becoming out of date, branched, or just plain broken. Therefore, using the extra few steps of

references and labeled variables as described above will have some initial investment time, but will save great amounts of time in the revision and iteration process.

Label Data

Figure 10.1 shows a difficult situation you might face when trying to debug a problem. You can see that something is wrong with this formula, but what is it? What is this formula trying to do? What part of the data is it used for? Is this formula actually useful?

The screenshot shows a spreadsheet interface. Cell A1 contains a very long, complex formula starting with "# Skills: Attack, Use, Interact, Equip, Collect, Give item, Move, Break Hold" followed by multiple nested IF functions. Cell B2 contains the error "#REF!". The formula in A1 is wrapped in a single-line formula bar at the top of the screen.

	A	B	C	D	E	F	
1							
2		#REF!					
3							
4							

Figure 10.1 A run-on nested function

There are multiple reasons this formula is problematic. First, the cell B2, which holds the problematic formula, is not labeled. It could have been labeled in A2 or C2, or with a note, or even with a comment. The label could have provided much-needed context to help figure out why it is returning a “REF!” error—or at least what it was intended to do in the first place.

Second, there is way too much going on in a single cell. Any one of the many parts of this formula could be the culprit, but it is hard to debug all of them at once. The spreadsheet can only tell you if the formula or function in total is correct; it cannot let you know about individual parts. To debug a giant run-on formula like this, you need to either do a lot of trial and error

or break it up into pieces and test each one. By breaking the formula into parts and labeling them, you could make it much easier to debug problems with each of the individual cells. You would then also be able to get better feedback from the spreadsheet to help pinpoint the locations of problems.

Note

Remember that, if needed, you can always hide intermediary columns or rows and show only the final calculation.

Validate Your Data

As much as you can, validate your data. For example, if you have character names listed in a column, you can validate the data in that column by ensuring that cells accept only text. If you have a column for the strength attribute, you can validate the data in that column by ensuring that it accepts only numbers. Ideally, you should go even further and make sure the strength column accepts only numbers within the range of numbers you have determined work for the strength attribute. If you can equip a character with a named weapon, you can validate that cell with a list built from the names of the weapons. It is incredibly easy to misspell words or to slightly rephrase them in a way that looks correct but that won't work for analyzing the data later. Validating the data in a spreadsheet can help you avoid problems that occur with such errors.

Validation is needed if you are making all the data for a game yourself, and it is even more important if you are working on game data with a group. It is vital that all the people entering data know what is expected in any given cell and that they are limited to inputting only technically acceptable data. While this can't completely eliminate typos and bad data entry, it can help reduce the number of times that entire data sheets need to be reworked or that single mistyped attributes cause the game to crash.

Use Columns for Attributes and Rows for Objects

[Figure 10.2](#) shows a list of attributes for a selection of fantasy creatures. Note that column A contains a list of the creatures, which are the data objects. Each column starting with B lists an individual attribute (abbreviated for the sake of organization in this sheet).

	A	B	C	D	E	F	G	H
1	Character Type	PLR	STR	DEX	HP	AV	DI	MV
14	Lizardman	y	3	7	7	2	4	7
15	Ogre	y	8	3	17	5	1	3
16	Ork	y	6	4	10	5	1	5
17	Rat Men	y	2	8	8	2	8	8
18	Wood Elf	y	3	8	8	1	6	6
19	Hellhound	n	5	4	12	3	3	6
20	Werewolf	n	7	5	7	1	1	5
21	Blob	n	7	3	20	7	2	2
22	Giant Ant	n	6	5	10	6	2	5
23	Warg	a	7	4	8	3	3	6
24	Clay Golem	m	7	5	12	6	3	4
25	Swamp Man	m	5	5	25	2	2	4
26	Mummy	u	10	4	20	6	2	2

Figure 10.2 Game data

While it is completely possible and mechanically sound to put the data objects in columns and the attributes in rows, it is not advisable to do so. The reason for this is simple tradition. Some time ago, likely in the 1980s, game designers started using computer spreadsheets to keep track of data objects. The intention was to eventually print out those data tables on paper. Most games made in those days had more data objects than they did attributes per data object. It made sense to organize the sheet so that each data object got its own row and could flow down onto multiple portrait-orientation pages, with attributes limited to a page width. This tradition

stuck, and since then, it has been common practice to put data objects in rows and attributes in columns, even though modern computers are completely capable of working either way. In addition, as game engines developed to import data objects into a game, they used the existing convention of rows for objects in their importers, cementing this tradition as the “right way” to do it. This means that while there is no technically right way to lay out a spreadsheet, there is a practical right way: with data objects in rows and attributes in columns.

Color Coding

There is no official color coding method or game industry standard for spreadsheets. Because color coding isn’t exported into game engines, it is just something you can use to help organize and visualize your data. Because there is no official standard for color coding, you have to come up with your own rules for color coding.

The most important factor with color coding for a spreadsheet is that it be labeled and consistent. [Figure 10.3](#) shows one example of a color coding scheme, but keep in mind that this is just an example. Many color coding schemes can be effective. As long as you set up a color coding scheme in the early stages of building the workbook and stick to it, yours is likely to work.

	A	B
1		
2	White on Black	Headers
3	Tan	Input data here
4	Green	Formula, DO NOT input data
5	Blue	Sub-label
6	Black on White	Static Text
7		

Figure 10.3 Sample color coding scheme

Figure 10.3 shows a color coding scheme in which only a few colors provide the level of detail needed for most games. Let's look more closely at the scheme used in this example:

- **White on black for headers:** Headers, which usually run along row 1, are the major labels for attributes or categories. White on black is very visually distinct, and it renders well on a black-and-white printer.
- **Tan for input data:** Data input cells are meant to be changed for the purposes of tuning. The data in these cells is always numbers or text and never formulas or functions.
- **Green for formulas:** In this scheme, green cells contain formulas or functions that refer to data in other locations in the workbook, or can do standalone calculations, like NOW(). Typing over data in these cells will break the calculations.
- **Blue for sublabels:** Sublabels are used for minor categories. The names of individual data objects are often color coded blue in column A to help the viewer distinguish the name from the rest of the data.
- **Black on white for static text:** Static data has often been brought in for reference from somewhere else. It should not be changed for tuning or any other purpose. For example, if you needed a list of mountain heights around the world, you could pull in static data. It would not be tan because you can't change it, and it would not be green because it is not formulas.

Figure 10.4 shows this color coding scheme in use. In this example, the color coding makes it possible for a viewer to tell, at a glance, roughly what is happening in the sheet:

- Row 1 is formatted as headers. The first header is for character type, and the rest of the headers are for attributes that govern all the characters in the game.
- The individual character names in column A are color coded as sublabels.

- Next, there is a group of attributes that you can change to balance the data, all colored tan.
- A number of analytical calculations are done on each character to get a weighted balance. Since these formulas are driven by the attributes, they are color-coded green.
- Finally, the column “Loot Cards” was copied from another data source. Although it is not a formula, it should not be changed, so it is coded as static text.

1	Character Type	STR	DEX	HP	AV	DIMV	LVL	Loot XP Cards	DPT	APT	TTK	OPT VAL	TTK VAL	MV VAL	Total VAL		
2	ROUS	3	3	3	1	5	3	0	1	0	0.90	0.5	0.32	1.80	0.39	1.35	21.27
3	Goblin	3	4	5	1	3	5	0	1	1	1.20	0.3	0.62	2.40	0.64	2.25	21.77
4	Zombie	4	2	6	0	0	2	0	1	1	0.80	0	0.60	1.60	0.75	0.90	24.50
5	Dark Wolf	4	4	7	2	3	6	0	2	0	1.60	0.6	0.74	3.20	0.93	2.70	40.99
6	Snapping Turtle	6	4	10	10	2	1	1	2	1	2.40	2	1.25	4.80	1.56	0.45	45.88
7	Barbarian	7	3	13	2	4	5	1	3	1	2.10	0.8	1.41	4.20	1.77	2.25	51.30
8	Dwarf	6	4	13	3	2	4	1	3	1	2.40	0.6	1.38	4.80	1.73	1.80	53.97
9	Gnoll	3	7	8	4	6	6	1	3	1	2.10	2.4	1.05	4.20	1.32	2.70	54.29
10	Halfling	3	7	6	1	8	5	1	3	1	2.10	0.8	0.65	4.20	0.82	2.25	53.59

Figure 10.4 Color coding example

Avoid Adding Unneeded Columns or Rows or Blank Cells

Another best practice is to avoid the temptation to add blank lines, rows, or cells to visually divide information in a table of information. In the example in [Figure 10.5](#), the designer wanted to show a difference between playable creatures and monstrous creatures. To make the distinction, the designer added a blank row and colored it black. This should be avoided for several reasons.

1	Character Type	STR	DEX	HP	AV	D	MV	LVL	XP
14	Lizardman	3	7	7	2	4	7	1	3
15	Ogre	8	3	17	5	1	3	1	3
16	Ork	6	4	10	5	1	5	1	3
17	Rat Men	2	8	8	2	8	8	1	3
18	Wood Elf	3	8	8	1	6	6	1	3
19									
20	Hellhound	5	4	12	3	3	6	2	3
21	Werewolf	7	5	7	1	1	5	2	3
22	Blob	7	3	20	7	2	2	2	3
23	Giant Ant	6	5	10	6	2	5	2	3

Figure 10.5 Unneeded row separator

The first problem with the blank row in this example is that it breaks the computational flow of the spreadsheet. Even if they are blank, those cells are recognized by the spreadsheet. If you try to formfill down a column in this spreadsheet (as described in [Chapter 5, “Spreadsheet Basics”](#)), the blank line will halt the operation. If you try to filter a column in the spreadsheet, those rows will be filtered out, making them meaningless. The blank line also throws off visual counts. If we have 20 data objects, the last one should be 20 rows after the header. With a bunch of blanks, the last object will appear in an arbitrary row number. Finally, blank lines don’t translate into data. If this sheet were exported into a game, those blanks would mean nothing to the engine and might even cause errors.

To distinguish between playable creatures and monstrous creatures, you should instead use another attribute in its own column, as shown in [Figure 10.6](#).

	A	B	C	D	E	F	G	H	I	J
1	Character Type	PLR	STR	DEX	HP	AV	DMV	LVL	XP	
14	Lizardman	y	3	7	7	2	4	7	1	3
15	Ogre	y	8	3	17	5	1	3	1	3
16	Ork	y	6	4	10	5	1	5	1	3
17	Rat Men	y	2	8	8	2	8	8	1	3
18	Wood Elf	y	3	8	8	1	6	6	1	3
19	Hellhound	n	5	4	12	3	3	6	2	3
20	Werewolf	n	7	5	7	1	1	5	2	3
21	Blob	n	7	3	20	7	2	2	2	3
22	Giant Ant	n	6	5	10	6	2	5	2	3

Figure 10.6 Adding an attribute column for the type of character

Now, not only is it possible for humans to see that there is a difference between the character types, it is possible for the computer to distinguish playable creatures from monstrous creatures. This setup will allow you to filter based on the PLR attribute if you want, and it will also allow you to export the PLR attribute so that a game engine can use it. In [Figure 10.6](#), you can see that the column labeled PLR is color coded white. This means the data in this column is static and should not be changed for tuning purposes.

Separate Data Objects with Sheets

There is a temptation to mash all the data objects for a game into a single giant sheet so that all the data can be seen and accessed at once. While this is technically possible from the perspective of the spreadsheet, it would make using the data much more difficult. Instead, you should separate the data objects by type. Data objects that have different attributes should go on

different sheets. While some differing data objects will share some attributes (“cost of item” is a classic example), if they have significantly different sets of attributes, the data object types should be separated onto different sheets. For instance, weapon and armor data objects should most likely be on separate sheets.

You can use a number of particular type of sheets to separate data objects, as discussed in the sections that follow.

Reference Sheet

In most games, the same variables are used over and over again in many different calculations. The best way to store these variables in a workbook of game data is to put them all into a single reference sheet (often called a *ref sheet*). The ref sheet is a bank of inputs that you plan to use multiple times on multiple sheets. You can also use a ref sheet to store variables needed for a single sheet that would otherwise clutter the sheet.

A ref sheet can contain variables like the following:

- Attribute weights
- Minimum and maximum attribute values
- Conversion numbers, such as game engine units per meter
- An overarching damage multiplier for difficulty adjustment
- Maximum and minimum character counts per level

[Figure 10.7](#) shows a ref sheet that provides notes to help the user understand what a number of acronyms stand for. This ref sheet contains various multipliers, weight scores, and other globally used numbers, as well as descriptions.

	A	B	C	D	E	F	G	H
1								
2	DPT Weight	2.00						
3	TTK Weight	1.25						
4	MV Weight	0.45	Move value weight for calculating character total value					
5	Total Val Weight	6	gets the numbers closer to 100pt scale					
6	Ave DPT	10	Average Damage per turn for doing total character value calculations. 2.4 was original value					
7	LVL Mult	10	Affects how spread out level advancement is					
8	Lvl. variance	40	normalizing level weights so the difference between levels is larger					

Figure 10.7 Reference sheet

Ref sheets can look different in different situations. The best way to start one is to simply create a blank ref sheet when you create a new workbook for game data. As you work through the game, you may notice that you are typing the same referenced number repeatedly. When you find this happening, you can convert this number from something you manually type to a named variable and store it on the ref sheet.

Introduction Sheet

If a workbook is going to be shared by multiple people, or even if only you will use it, but you anticipate that it will become complex, you might want to use an introduction sheet. An introduction sheet is a sheet to the very beginning of the workbook that is labeled “Instructions” or “StartHere” or “Help.” On this sheet, you can show your color coding, explain what each of the sheets is for, and record any instructions needed for use.

Figure 10.8 is an introduction sheet from a workbook made to create and track teams in the game *Blood Bowl*.

Worksheets: Description and usage:	
Roster	The main page of the workbook. It's the roster for your team and is set up to print all needed information for a game. You will need to make a copy of this workbook to make changes and create your own roster.
Log	Keep track of your games played. The results are also updated on the Roster sheet
Ledger	When you make purchases for your team, use the ledger to track what you are purchasing.
Inducements	If you are playing against a team with a higher rating, go to this page to get inducements (which help you)
Skill Descriptions	Skill descriptions from LRB5. The "improvement" column indicates if a player on your team has that skill, including star players or mercenaries. You can filter by this column and print all skills that are relevant to your team to have as reference during a game.
Passing	Chart that shows passing range
Color Code	Reference for how to color code your team positions so everyone knows who is who.
Hidden Sheets	Don't change anything on any of these sheets as it is imported from a source document and any changes will break this entire workbook.
Notes on usage	General note: Don't type in any cell that is not light tan, like this one. Many cells contain formulas that will break if you type in them. The tan indicates a field that it is expected you will fill out. Note 2: Many cells have a drop down arrow next to them. Use this! Lots of formulas are based on the exact text entered and all the values in the drop down are accurate. If you type in invalid values, many formulas will break!!!

Figure 10.8 Introduction sheet

This introduction sheet provides instructions for using the workbook as a whole and notes on individual tabs. It is a help sheet in a workbook created for novices, so it is more thorough than would be needed for a workbook used by a professional team.

Note

The instruction sheet in [Figure 10.8](#) provides clear instructions on what new users should be changing and what they should be leaving alone. It is relatively easy to break a spreadsheet, so giving clear instructions to new users is very important.

Output/Visualization Sheets

Once you have made large masses of data objects, you will almost certainly want to analyze the data by using functions and formulas or visualize it by using charts. In general, it's better to create charts on separate sheets than to try to add them on the same sheet as the data in different rows or columns.

In the rather complex example shown in [Figure 10.9](#), the probability of combat outcomes for the game *Blood Bowl* has been calculated using character attributes in conjunction with the rules for combat. This chart illustrates how large and cumbersome data analysis charts can become. Having these kinds of tables on the same sheet as the game data makes both aspects much more difficult to use and much more likely to break. [Chapter 15, “Analyzing Game Data,”](#) provides much more information on how to handle such analyses. For now, the import takeaway is that you should separate the data from the analysis.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	
1	Dice	picker	Att Skill	Def Skill	DefDwn	Push	Null	BthDwn	AttDwn	Def Down	Push	Null	Bth Dwn	Att Down	Pos	Neg		Bth Dwn	Att Dwn	Bth Dwn	Att Dwn	neg		
2	3	Def	None	Bodge																				
3	3	Def	None	Block																				
4	3	Def	None	Dodge																				
5	3	Def	None	None																				
6	3	Def	Block	Bodge																				
7	3	Def	Block	Block																				
8	3	Def	Block	Dodge																				
9	3	Def	Block	None																				
10	2	Def	None	Bodge	1	17	0	0	58	3%	47%	0%	0%	50%	100%	50%	50%							
11	2	Def	None	Block	6	12	0	0	58	17%	33%	0%	0%	50%	100%	50%	50%							
12	2	Def	None	Dodge	1	23	0	3	5	3%	64%	0%	0%	25%	100%	67%	33%							
13	2	Def	None	None	6	16	0	5	9	17%	44%	0%	14%	25%	100%	61%	39%							
14	2	Def	Block	Bodge	1	15	11	0	5	3%	42%	31%	0%	25%	100%	75%	25%							
15	2	Def	Block	Block	4	12	11	0	9	11%	33%	31%	0%	25%	100%	75%	25%							
16	2	Def	Block	Dodge	4	23	0	0	5	9%	64%	0%	0%	25%	100%	75%	25%							
17	2	Def	Block	None	11	16	0	0	9	31%	44%	0%	0%	25%	100%	75%	25%							
18	1	Att	None	Bodge	1	3	0	0	2	17%	50%	0%	0%	33%	100%	67%	33%							
19	1	Att	None	Block	2	2	0	0	2	33%	33%	0%	0%	33%	100%	67%	33%							
20	1	Att	None	Dodge	1	3	0	1	1	17%	50%	0%	17%	33%	100%	67%	33%							
21	1	Att	None	None	2	2	0	1	1	33%	33%	0%	17%	33%	100%	67%	33%							
22	1	Att	Block	Bodge	1	3	1	0	5	17%	50%	17%	0%	33%	100%	67%	33%							
23	1	Att	Block	Block	2	2	1	0	1	33%	33%	17%	0%	33%	100%	67%	33%							
24	1	Att	Block	Dodge	2	3	0	0	5	33%	50%	0%	0%	33%	100%	67%	33%							
25	1	Att	Block	None	3	2	0	0	5	60%	33%	0%	0%	33%	100%	67%	33%							
26	2	Att	None	Bodge	11	21	0	0	4	31%	68%	0%	0%	11%	100%	89%	11%							
27	2	Att	None	Block	20	12	0	0	4	56%	33%	0%	0%	11%	100%	89%	11%							
28	2	Att	None	Dodge	11	21	0	1	3	31%	68%	0%	25%	8%	100%	91%	9%							
29	2	Att	None	None	20	12	0	3	1	56%	33%	0%	1%	2%	100%	91%	9%							
30	2	Att	Block	Bodge	11	21	2	0	1	31%	58%	17%	0%	25%	100%	97%	2%							
31	2	Att	Block	Block	20	12	3	0	1	56%	33%	0%	0%	2%	100%	97%	2%							
32	2	Att	Block	Dodge	20	15	0	0	1	56%	42%	0%	0%	2%	100%	97%	2%							
33	2	Att	Block	None	27	8	0	0	1	75%	22%	0%	0%	3%	100%	97%	2%							
34	3	Att	None	Bodge																				
35	3	Att	None	Block																				
36	3	Att	None	Dodge																				
37	3	Att	None	None																				
38	3	Att	Block	Bodge																				
39	3	Att	Block	Block																				
40	3	Att	Block	Dodge																				
41	3	Att	Block	None																				

Making a roll

Avoiding a turn over

Figure 10.9 Complex analysis

Scratch Sheet

Very commonly while laying out data objects or working in spreadsheets in general, you will need to do quick one-off calculations. To keep your workbook tidy and organized, it's best to confine all those messy bits in a single place. You can create a scratch sheet at the very far right of all the other sheets for such bits. It is okay to delete or overwrite anything on the scratch sheet. If you find that you are using a calculation from the scratch sheet frequently, you should pull it over to the ref sheet or make a new sheet that can act as a permanent container for the information.

Spreadsheet Example

If you follow the best practices described in this chapter, when you create a new workbook, you will ensure that it contains several sheets right from the beginning: help, ref, data, analysis, and scratch sheets. [Figure 10.10](#) shows an example of what this might look like. By starting with a solid structural framework, you set up the flow of data from idea to reality in a cleaner and more organized way.

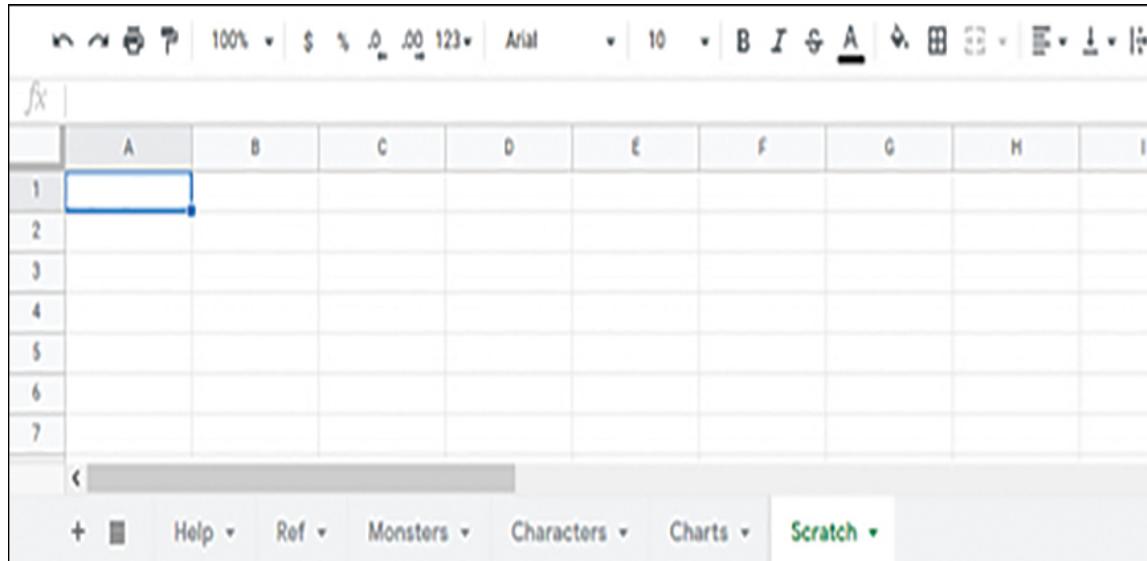


Figure 10.10 Best practice spreadsheet organization

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore data organization:

- For many modern games, you can find a lot of data online. Take some time to look up the data for your favorite games and see how it is organized. What attributes did the game designers use? In what order are the objects listed? How have the game designers separated out their data object types? By reviewing how others have done this for several different games, you will gain a better understanding of how to do it yourself.
- Create a template spreadsheet for yourself. Create a color coding system and record it in an introduction sheet. Once you have created this template and ensured that it works well, you can use it to create new workbooks and be ensure that they are properly formatted from the start.

Chapter 11

Attribute Numbers

So far in this book, you have created objects and their attributes. You have also created a spreadsheet to organize all your data. The next step in bringing your game ideas to life is to start putting in numbers for all of those attributes.

Getting a Feel for Your Attributes

Before trying to assign numbers to attributes, you should start by getting a feel for what you want to get from those attributes. For example, if you were making a racing game and wanted to create the speeds and acceleration attributes for three different vehicles, you could start with some descriptions of what the speed and acceleration feel should be:

- **Sports car:** Good acceleration and good top speed
- **Muscle car:** Fastest top speed but with less acceleration than a sports car
- **Motorcycle:** Fastest acceleration but lowest top speed

While you have assigned no numbers to these attributes yet, you now have a guide that will help in determining what numbers fit the feel you want.

Determining the Granularity for Numbers

After you come up with attributes for game objects, you need to assign numbers to the attributes. Because you are making up all the attributes and numbers for your game, technically you could use any numbers you want. The granularity of the numbers you use can have a dramatic impact on how a player perceives the game. The following sections provide some to help you determine the granularity of your numbers.

Numbers Should Relate to Probability

Numbers should have a visible impact on the game. The larger the possible outcome of a random event, the larger the corresponding numbers of the game must be. For example, if a character has 10 HP, it doesn't matter if the character receives 11 damage or 5,000 damage, as either one will be a one-hit kill. Say that you know a character is rolling 1D6 (a single six-sided die) for damage, and you always want the character to survive at least three hits. In this case, the minimum hit point value would be 111.

Let's consider backgammon as an example. (Do a search for "official backgammon rules" if you need to familiarize yourself.) In backgammon, the maximum number of moves a piece can take at one time is 24. The maximum is 24 because even the largest roll possible can have a use and not be wasted. In addition, 24 is the number of spaces on the board (see [Figure 11.1](#)). The relationships between the number of needed movement spaces and the potential outcomes of the dice are intertwined. If you were to expand the board, you would likely need larger potential rolls to keep the game moving. Conversely, if you were to shrink the board, you would want to reduce the amount of possible movement.

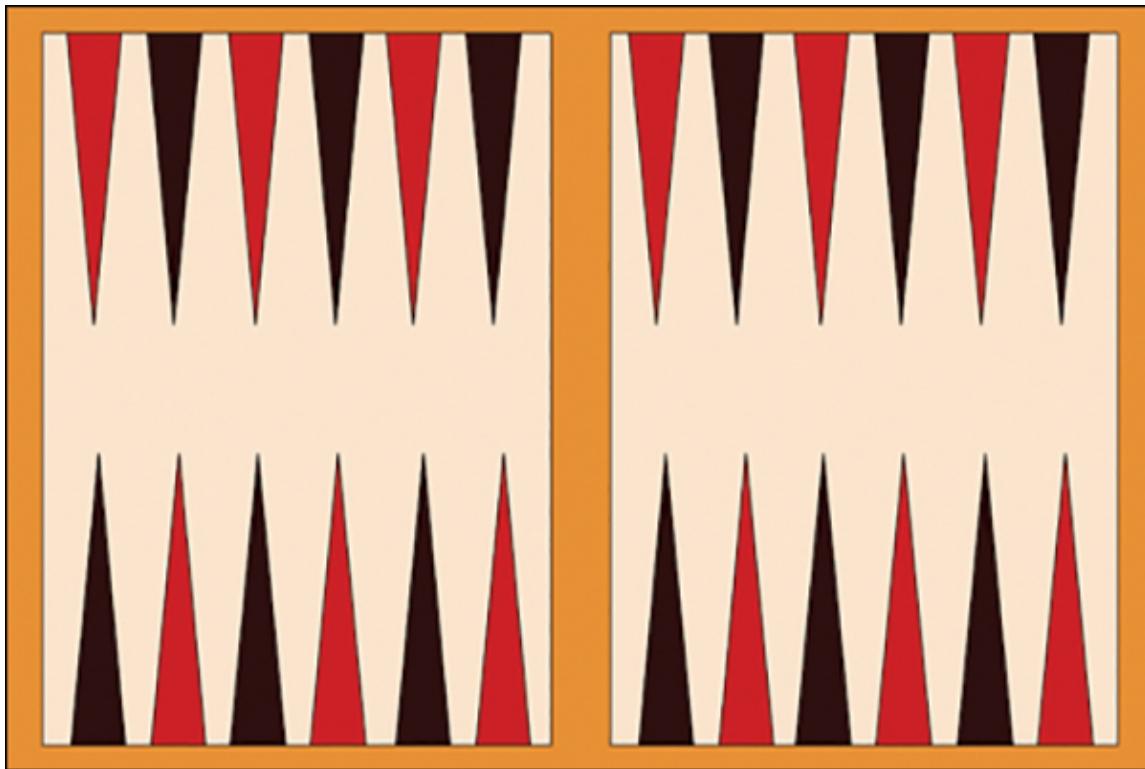


Figure 11.1 Backgammon board

Some Numbers Need to Relate to Real-World Measurements

Some numbers, such as height, weight, and speed, are analogs of the real world. The scale of those numbers has already been decided for you. Even if it is better for your game to use three-digit numbers than to use smaller numbers, you can't decide that every person in your game is going to be measured in hundreds of feet (or meters) in height. Players have incoming knowledge of fixed scales and expect you to play along with the real world. So, if being taller in your game is better, then you will need to adjust your scale. There are a few ways to do this:

- Use a smaller unit of measurement so you get larger numbers.
- Adjust your scale of numbers to fit a fixed attribute.
- Convert the real-world scale to a game scale.

For example, you might list attributes for a basketball player as follows:

Example 1

Strength: 150

Height: 6 (feet)

Speed: 220

Dexterity: 180

This looks odd because the height attribute is a single digit, while the rest of the attributes are triple-digit numbers. In addition to looking odd, this would create the need to use fractions or decimals. Here's another example of attributes for a basketball player:

Example 2

Strength: 150

Height: 182 (centimeters)

Speed: 220

Dexterity: 180

This scale is much better. All the attribute numbers are triple-digit numbers and within a similar range.

Here's another example of attributes for a basketball player:

Example 3

Strength: 50

Height: 72 (inches)

Speed: 73

Dexterity: 60

This scale is also better than the first one. Changing to a more granular measurement of inches and switching all attributes to be two-digit numbers makes them line up nicely.

Now consider this final example of attributes for a basketball player:

Example 4

Strength: 150

Height: 165 (game units)

Speed: 220

Dexterity: 180

This scale also works because you have ditched reality and made your own scale that enables the attributes to all be three-digit numbers in a similar range. Making up your own units may lead to a bit of confusion as a player won't initially know how to picture a height of 165 game units, but you can overcome this difficulty with art.

User Smaller Numbers for Easier Calculations

A player needs clear numbers for each individual calculation and for repeated calculations. If you are asking players to do calculations in their head in the game, then you need to limit the complexity of the numbers. Further, if you are asking players to do many calculations or frequently recurring calculations, you need to further restrict the complexity of those calculations. It is easiest for players to process simple numbers—that is, small whole numbers.

In very old games, attribute numbers are all very small. The number of pieces a player has, the faces of the dice, and total points for a game tend to be no more than two digits. Often they are single digits. Old games use small whole numbers to make the numbers easier for players to remember and use in calculations in their heads. The more frequently a player is required to do calculations, the simpler the calculations tend to be and the smaller the numbers involved are.

Think again about backgammon, for example. Players need to be able to calculate rolls and results in their heads, and complex systems of multiplication or addition would cause unneeded confusion. For each turn in backgammon, a player rolls 2D6 to determine how much movement their pieces get for that turn. A player gets double that movement with a roll of doubles. (Rolling double 6s, for example, allows the player to move a total

of 24 spaces.) On every turn, the player uses the individual rolls of the dice, or adds together the rolls of two six-sided dice, and turns go by in a matter of seconds. Fortunately, adding together the rolls of two six-sided dice is a very easy calculation and does not slow the pace of the game. In addition, the results are all small numbers. The results also tie into the physical space of the game. The board contains only 24 spaces, so any more movement than that would be useless.

Let's now consider scoring in the game spades. Spades has a rather sophisticated scoring system, where players guess their score at the beginning of the game and then, at the end of the game, compare their final results to their initial guess. They then use a scoring system to interpret their results and calculate the final score. This is a somewhat complex calculation, and players often use paper or a calculator to do the scoring—but it is only done once during a game. The numeric results are also much larger than in backgammon, with scores in the hundreds or even up over 1,000. Because this calculation occurs only once a game, it's an event and can even build some tension as a game is calculated, but if it were done every turn, it would completely bog down the game.

Early and even many modern tabletop games and pen-and-paper RPGs continue to use attribute numbers in the single digits and low double digits. For example, a sample fifth edition Dungeons & Dragons character could start with the following attribute scores:

STR 10 DEX 13 CON 14 WIS 19 CHA 14

Note that all of these numbers are in the low two-digit range. Also, while this is a modern, fairly sophisticated game, it is working under the same limitations as backgammon in that the players are needing to do calculations in their head. Whereas in backgammon, players do calculations every few seconds, in an RPG they do calculations every few minutes.

As you can see from these examples, the less frequently calculations are made, the more complex they can be and the larger the numbers involved can be. When assigning numbers to attributes, you should think about how much calculation you expect your players to do in their heads. The more calculations, the smaller the numbers should be for attributes. The more

frequent the calculations, the smaller and simpler the calculation and numbers must be.

Use Larger Numbers for More Granularity

If small numbers are easier for players to understand, why not use single-digit numbers for everything? Small numbers do not allow for much granularity or variety. Say that you are assigning strength to five fantasy characters. These are the five characters, and the feeling you want to convey through the strength attribute for each of them:

- **Human:** Middle-of-the-road guy
- **Ogre:** Much stronger than anyone else
- **Ork:** Stronger than humans but significantly weaker than ogres
- **Goblin:** Weakest by far, but not so weak that they can be ignored
- **Dwarf:** Stronger than humans but notably weaker than orks

Here's how you might turn these feelings into numbers if you want to constrain the numbers to 10 and below:

- **Human:** Middle of the road leads you to choose the halfway point, which is 5.
- **Ogre:** Because this is the strongest character, it is 10. Note that there is no longer room on the scale for stronger characters like dragons or giants. While this might be fine within the scope of your game, it does limit your ability to expand the game.
- **Ork:** You might assign an ork a strength of 7 because an ork is much weaker than an ogre but is not that much stronger than a human.
- **Goblin:** A goblin is the weakest character, so you assign it 2, but 2 might be too weak.
- **Dwarf:** You are now stuck. If you assigned a dwarf 6, then this character would be stronger than a human but not notably weaker than an ork.

As you can see, even with just five characters and a few criteria, you start running out of space in the scale to properly translate your feelings about character strengths into numbers. As you add more characters and more criteria, the scale will get even more crowded, and characters will start to feel too similar. To fix this, it is tempting to make all the values considerably larger, allowing more granularity to work with.

Very Large Numbers Are Confusing

Given the problems discussed so far with small numbers, it might seem like a good idea to go to the opposite extreme in a computer game. If you were to use four- or five-digit numbers, you would have plenty of space to make a large variety without ever crowding your range. Further, given that the computer will be doing all the calculations, you don't need to worry about players doing lots of math on big numbers, as they would need to do with a board game. But calculations are not limited to just what a player must do to make the game progress; they also tie in to how well the player can understand what is going on in the game. We humans are, in general, not designed to calculate large numbers in our heads. For example, try to calculate the final hit point score for each of the following scenarios in your head:

- 5 hit points, taking 2 points of damage
- 100 hit points, taking 27 points of damage
- 34863298 hit points, taking 456321 points of damage

It's clear that the smaller the numbers, the easier the calculations.

The takeaway is that you need to find the right amount of granularity for your game. In general, you want to use numbers that are just large enough to accommodate all needed variety but no larger than absolutely necessary.

Humans Hate Decimals and Fractions, but Computers Don't Mind Them

It is exceedingly rare, outside of educational math games, to ever show a player a decimal score or a fraction. It's not that they aren't valid numbers,

but people just don't like seeing or (worse) calculating them. Games typically show players only whole numbers.

However, behind the scenes, computers have absolutely no problem calculating decimals. This means you can feel free to use as many decimal places as you want for computer calculations as long as you can present whole (rounded) numbers to the player in a way that is not confusing.

Numbering Example

[Figure 11.2](#) provides an example in which each column presents a pair of values: one for Attribute A and one for Attribute B. In each pair, the ratio of A to B is the same: 94%. Because each pair has the same ratio, for a computer, they would all work exactly the same way. However, players would be able to comprehend some of these numbers easily and others with great difficulty. If the players are going to see the numbers, you should use just the two-digit numbers, if possible, or the three-digit ones.

Attribute A	1.230769231	16	160	4592
Attribute B	1.307692308	17	170	4879
Ratio	94%	94%	94%	94%

Figure 11.2 Number granularity example

The Tension Trick

There is a trick that systems designers can use to cause a wide variation of tension in a game by manipulating a few related numbers. The basic rules for tension are as follows:

- Using numbers that are not easy to calculate creates dissonance for players.
- Dissonance creates tension, fear, and other heightened negative emotions.
- These emotions can heighten an experience, if used properly.
- Using numbers that are easy to calculate creates calmness for players.
- Use easy-to-calculate numbers to give the players a calm, easygoing experience and use numbers that are difficult to calculate to cause more heightened emotions.

For example, say that a player character (PC) has 20 HP, and an enemy character should kill the PC in 4 hits. You could assign these numbers for the least tension:

Enemy does 5 damage per hit, so the PC is at 5 HP after 3 hits and at 0 HP after 4 hits.

You could assign these numbers for the most tension:

Enemy does 6 damage per hit, so the PC is at 2 HP after 3 hits and at 0 HP after 4 hits.

In both of these cases, the PC is alive after 3 hits and killed on the fourth, so functionally they are the same. But they can feel very different to a player. Why?

Let's look at it graphically and then break it down further. Imagine that the PC has taken 3 hits. [Figure 11.3](#) shows two options for the health bar for the PC at this point.

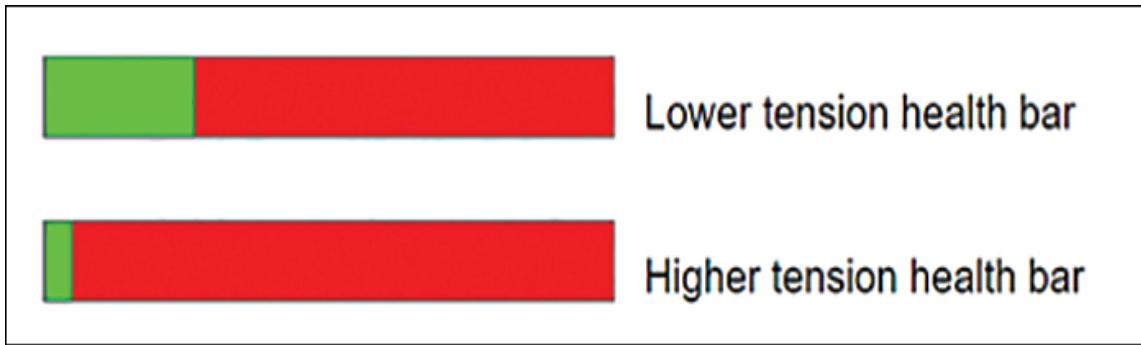


Figure 11.3 Lower- and higher-tension health bars

In both cases, the PC will be killed with the next shot, but which one looks scarier? Players know that more red on a health bar is generally a bad thing. The fact that the lower of the two bars is more red signals to the player, subconsciously, more danger, even though numerically the danger is identical with the two health bars.

Let's look at another example. Say that, in a farming game, the player plants a field that is 20 square meters in 1-square-meter units, so there are 20 total spaces in which to plant. The player has the following resources:

- 5 corn
- 10 beans
- 5 wheat
- 10 rice

In this example, it is fairly easy for a player to calculate the division of crops to plant. All the numbers are easy to grasp and can easily fit in 20, which is also the total number of squares. Young or inexperienced players should be able to quickly figure out what to do in this scenario, with little stress.

To increase the tension in the same farming game, you can change the units to something more difficult to grasp and also change the amounts to numbers that are more difficult to calculate. This time, say that the player

has 2.5 acres to plant and plants in units of 100 square yards. This alone makes the calculations much more difficult for anyone who is not already familiar with converting square yards into acres. In this case, the player would have 121 things to plant. The player has the following resources:

37 corn

63 beans

58 wheat

29 rice

In this revised example, it is very difficult for the player to do the planting calculations in their head. This difficulty will cause a sense of stress and tension. In an action game, this can heighten the player's experience, but in a farming game, it might create stress in what should be a relaxing activity.

There are no universal right or wrong answers about inducing tension in a game through use of numbers, but there are situational rights and wrongs based on the feeling you want the player to have at any given time.

Searching for the Right Numbers

Once you decide on the granularity of the numbers you are going to use, it's time to start plugging in numbers. If you have already described the feel you want with the numbers and determined the number of digits and ratio you want to use, you can do a rough pass immediately.

Keep in mind when doing a first pass at data numbers that they will almost certainly not be what you end up with. This is okay and to be expected. Until a game is tested, it is impossible to know the exact effect numbers will have on the game. Don't think of this as failure; instead realize that you can take the pressure off the first pass. If you approach the first pass knowing that the numbers will be wrong, you don't have the stress of trying to guess right the first time. Instead, you can just get some numbers in there. Use the targeted number of digits and rough ratios for each object and just plug them in.

Let's go back to our racing game example from the beginning of the chapter. Say that you want to make a very simple, new-audience-friendly game, so you want to stick to single-digit numbers. This is what you came up with earlier for what the speed and acceleration should be:

- **Sports car:** Good acceleration and good top speed
- **Muscle car:** Fastest top speed but with less acceleration than a sports car
- **Motorcycle:** Fastest acceleration but lowest top speed

Based on this list and the fact that you want to use single-digit numbers, you might assign the numbers shown in [Table 11.1](#). Are these numbers right? Almost certainly not. But they're a start.

Table 11.1 Basic data table

Car	Acceleration	Top Speed
Sports car	8	8
Muscle car	6	10
Motorcycle	10	6

When testing numbers, it's a good idea to go beyond reasonable, expected numbers. To find the extents of a range, you must exceed those extents during testing. You want to try making something with too much acceleration or a speed that's too low; for example, you might experiment with your numbers as shown in [Table 11.2](#).

Table 11.2 Experimental data

Car	Acceleration	Top Speed
Sports car	8	8
Muscle car	1	15

These numbers are undoubtedly wrong—and, again, that's fine and expected. You are not trying to get the numbers right at this point. Instead, you are trying to understand your game and game engine. Can the engine handle an acceleration of 200? Does this number cause the game to crash? Does collision still work? By testing unreasonable numbers, you can understand the game and engine better, which will make it more likely that you will find interesting and exciting new results.

The great news is that with game data, there is nothing you can do in testing that can't be undone. You can use this aspect of game making to your advantage for wild and interesting tests. Once you have broken the game in interesting ways and understand the mechanical workings better, it's time to home in on the balance you truly want.

The next step is to test and test and test—and then tune and test more and then do more tuning and testing. On this first round of testing, the goal is to get the numbers to emulate what you wrote in your original list of what you feel you want from the numbers. Does that motorcycle feel like it has great acceleration? Does the sports car feel like it has slower acceleration but can eventually top out at the highest speed? Eventually you will find the right balance with the numbers.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the numbers that populate game data:

- Look online for data for your favorite games—in a variety of genres—and analyze the scales used in those games. Take note of the kinds of numbers used for each game and how the games compare with each other in terms of the numbers.
- Take the preceding exercise a step further and redo the values for each of the games by changing their values proportionally. Try doubling them, or multiplying by 10, or multiplying by 0.1. Describe how the

feel of the game changes when you change the scale of the data numbers.

Chapter 12

System Design Foundations

Once you have decided on the attributes you want to quantify and the scale of numbers you would like to use, it's time to look at more factors in creating a mass amount of data objects. In the next steps, you learn how attributes of different quality get different numbers and how to properly name all the data objects you create.

After you have assigned numbers to every attribute, it is a good idea to add them all up so you can compare the different data objects against each other. By getting a total score, you can start to get some insights into the balance of all the data objects in comparison with each other.

Attribute Weights

This section shows how to find attribute weights for the racing example from [Chapter 11, “Attribute Numbers.”](#) [Table 12.1](#) shows how you would get a total score for each car, based on its attributes (refer to [Table 11.1](#)).

Table 12.1 Adding the attributes for each car

Car	Acceleration	Top Speed	Total Value
Sports car	8	8	16
Muscle car	6	10	16
Motorcycle	10	6	16

Now, you need to test these numbers in a game. For this example, you would race all three cars several times and find out which one wins most often and by what margin. Let's say that in this testing, the muscle car wins

every race by a significant margin. You are now seemingly stuck: The data is clearly not balanced if one car is winning every time. But if you lower either attribute value for the muscle car, its total won't be as high as the totals for the other two cars, so you will no longer be able to compare them in the spreadsheet.

This kind of situation happens with game data all the time. It's actually fairly rare for attributes to add up across balanced game objects to be the same. It often turns out that one attribute is simply more important in the context of the game than others are. In the case of our racing example, the more important attribute appears to be top speed.

If we add one more attribute to our cars, we can get an even better idea of how badly skewed raw attribute numbers can be. [Table 12.2](#) shows an additional attribute, horn sound, and one more car.

Table 12.2 An additional car and an additional attribute

Car	Acceleration	Top Speed	Horn Sound	Total Value
Sports car	8	8	4	20
Muscle car	6	10	4	20
Motorcycle	10	6	4	20
Clunker	1	1	30	32

If you simply look at the total raw scores for the car, it seems like the clunker is far and away the best vehicle. However, when you look at the attributes in the context of what they do in the game, you see a very different picture. This is a racing game, so the loudness of the horn is not important to the game, even if it is an attribute you can control. The problem is that attribute value is throwing off the total value of each car.

The solution to this problem is to use attribute weights. An *attribute weight* is a multiplier you apply to each attribute that compensates for some attributes being more important than others. When setting up the data at first, using 1.0 is a nice neutral starting point. It indicates that all attributes

are equally important and lets you get through the data entry process more quickly. When you begin testing, you are likely to find that the attributes are not of equal value, and they need to be modified with weights. In the racing example, testing shows you that top speed is more important than acceleration, and horn loudness is not very important at all. You can use these observations to inform your first pass with weights, which might look something like this:

- **Acceleration:** 1.0 (Neutral. Not very important but not unimportant.)
- **Top speed:** 1.5 (Top speed is therefore 150% as important as acceleration.)
- **Horn sound:** 0.0 (This essentially removes horn sound from the total value equation since it is not an important factor for the game.)

You can now recalculate the weighted total for each car, as shown in [Table 12.3](#).

Table 12.3 Weighted attributes

Car	Acc	Speed	Horn	Acc	Speed	Horn	Total Value
<i>Weights</i>	1.0	1.5	0.0	<i>Weighted</i>	<i>Weighted</i>	<i>Weighted</i>	
Sports car	8	8	4	8	12	0	20
Muscle car	6	10	4	6	15	0	21
Motorcycle	10	6	4	10	9	0	19
Clunker	1	1	30	1	1.5	0	1.5

With the new weights applied, the total value looks a lot more like the test results you see in the actual game. The muscle car always wins because it has a better total real value in game. The goal of the weights should be to represent the effect of each attribute in the game as realistically as possible.

The next step is to go back to the data and adjust it so that each car has the same total weighted value while retaining its own unique character. Given

these goals, you can adjust the attributes on the other vehicles as shown in [Table 12.4](#) to make them feel different from each other, hit your intended goals for the feel of the game, and use balance attribute weights.

Table 12.4 Modified attributes

Car	Acc	Speed	Horn	Acc	Speed	Horn	Total Value
<i>Weights</i>	1.0	1.5		0.0	<i>Weighted</i>	<i>Weighted</i>	<i>Weighted</i>
Sports car	8	8	4	8	12	0	20
Muscle car	5	10	4	5	15	0	20
Motorcycle	11	6	4	11	9	0	20
Clunker	1	1	30	1	1.5	0	1.5

With these new numbers, it appears as though you have reached all your goals—but there is only one way to find out. After doing a balancing pass with weights, you need to go back and test the feel of the game. If the game now feels more balanced, you can conduct some specific tests to find out how many times each car wins and by what margin. When you have that information, you can be more confident that your game is more balanced, and you will be able to add new objects to your spreadsheet and quickly get them into balance before ever actually placing them in the game.

One more factor we need to look at here is that the clunker is still way out of balance. This type of imbalance is not always a bad thing or something that needs to be fixed. This might be a joke car that was placed in the game, or it could be a punishment for a player who wrecks a car too often that will set up the player for almost certain failure. It could even be a challenge car that a very good player might pick to show that, even with the quantifiably worst vehicle in the game, they can still win. These are just some of the valid reasons to have data objects that are well out of balance. It is just as important that you know they are out of balance and by how much.

It is important to note that weighting can require some counterintuitive thought. In [Table 12.4](#), you can see that speed is weighted 1.5 and

acceleration is weighted 1.0, which means that speed is more important than acceleration in this game. It also means that the raw numbers for speed are likely to be smaller in total than the raw numbers for acceleration.

Let's consider a more extreme example. Say that you are making an archery game where each archer has the attributes power and accuracy. Because a missed shot does no damage, you have determined that accuracy is twice as valuable as power. In this case, your attributes would get the following weights:

- **Accuracy:** 2.0
- **Power:** 1.0

This means two equal archers would have raw attribute numbers where the power attribute is higher than accuracy in most cases. It's only once the weight is applied to the raw values that you can see which of the players is the better archer. For example, look at [Table 12.5](#).

Table 12.5 Archer attributes

Archer	Power	Accuracy
Archer 1	10	10
Archer 2	20	5

At a glance, if you don't know the weights, Archer 2, who would have a higher total attribute summed score, looks like the overall better archer. But when you apply the weights, you can see that the two archers are, in fact, balanced (see [Table 12.6](#)).

Table 12.6 Weighted archer attributes

Archers	Power	Accuracy	Power	Speed	Total Value
Weights	1.0	2.0	<i>Weighted</i>	<i>Weighted</i>	
Archer 1	10	10	10	20	30

Archer	2	20	5	20	10	30
--------	---	----	---	----	----	----

Be patient when searching for proper weights. It can take dozens of tests—or more—in varying circumstances with multiple playtesters to determine the proper final weights to apply to your attributes. While this process should start very early in the production of data objects, it’s likely that it will continue throughout production and almost up to the end.

DPS and Intertwined Attributes

Very often, a single attribute tells only part of the story of the in-game interaction. The most common example of this is damage per second (DPS), where a series of characters have both a damage attribute and an attack rate attribute. Each of these has its own scale, with a minimum and a maximum. However, looking at only one part of the pair—that is, looking at only damage or only attack rate—does not explain accurately how powerful the character is. For example, say that you have a pair of weapons—a dagger and a battle axe—which have the damage scores shown in [Table 12.7](#).

Table 12.7 Weapon damage scores

Weapon	Damage
Dagger	6
Battle axe	14

Which one of these weapons does more damage? Without any other context, the answer is fairly clearly the battle axe, but let’s look a little deeper. [Table 12.8](#) shows how often each weapon can attack in a minute.

Table 12.8 Weapon damage per minute

Weapon	Damage	Attacks per Minute
Dagger	6	17

Now which weapon do you think does more damage? While individual hit damage has not changed, the total accumulation of damage output has changed considerably. Instead of asking how much damage each weapon does, it would be more helpful to consider how much damage each weapon does in a single minute. When you take into consideration both the amount of damage done and the frequency at which damage is done, you can get a much clearer idea of which weapon does more damage in a practical setting in the game. Because video game combat goes fast—usually in seconds rather than in minutes—we commonly express this value as DPS (damage per second).

But how do you determine DPS for two weapons that can't attack every second? To do this, you need to take an intermediate step and calculate using a scale at which both weapons can do damage multiple times. For this example, we can use minutes, but any amount of time will work, as long as it's enough for both data objects to have multiple attacks. To get the value, you need to multiply the damage of each shot by the frequency:

$$\text{Weapon damage} \times \text{Attacks per minute} = \text{Damage per minute}$$

[Table 12.9](#) shows the results for the dagger and battle axe example.

Table 12.9 Damage per minute calculation

Weapon	Damage	Attacks per Minute	Damage per Minute
Dagger	6	17	102
Battle axe	14	7	98

You can now clearly see that the dagger does slightly more damage over time. To translate damage per minute to DPS, you divide the damage per minute value by 60, using this formula:

$$(\text{Weapon damage} \times \text{Attacks per minute} = \text{Damage per minute})/60$$

[Table 12.10](#) shows the results for the dagger and battle axe example.

Table 12.10 DPS values

Weapon	Damage	Attacks per Minute	DPS
Dagger	6	17	1.7
Battle axe	14	7	1.6

With this calculation, you can now compare two weapons meaningfully, and in [Table 12.10](#), you can see that over time the dagger will do slightly more damage than the battle axe. Note that the final DPS value has a decimal portion, so it is likely going to be used only for internal evaluation. If you wanted to represent this to players, it might be better to leave it as damage per minute or show it graphically as a bar chart.

The calculation becomes slightly more complicated when the frequency is measured in delay, as opposed to attacks per time. For example, for the dagger and battle axe, you might have a cooldown delay between attacks. To convert a cooldown into a per time measurement, you divide 60 seconds by the cooldown to determine how many times the cooldown can trigger in a minute. For example, a 3-second cooldown would be $60/3 = 20$ attacks per minute.

While DPS is the most common intertwined set of attributes used in games, it's far from the only one. The good news is that any given set of intertwined attributes can be treated in the same way as DPS for the sake of comparison. Let's look at one more example that at first seems to be quite different. Say that you have forklifts that can carry pallets. In this game, quickly moving pallets is advantageous, so you want a forklift that can carry a lot of pallets. The other part of the equation is how many trips the forklift can take in a minute. The more trips, the better. [Table 12.11](#) summarizes the capabilities of two forklifts.

Table 12.11 Forklifts

Forklift	Pallet Capacity	Trip Duration, in Seconds
Speedy	5	20
Big Brutus	9	30

Which forklift is the better one in terms of this game mechanic? To compare the two, you need to first calculate how many trips per minute each can make:

$$\text{Speedy: } 60/20 = 3$$

$$\text{Big Brutus: } 60/30 = 2$$

These are now your trips per minute scores. Next, you need to multiply the capacity by the trips per minute to get a total pallets per minute score. This gives you a final comparison that looks like what is shown in [Table 12.12](#).

Table 12.12 Compared attributes

Forklift	Pallet Capacity	Trip Duration, in Seconds	Pallets per Minute
Speedy	5	20	15
Big Brutus	9	30	18

In this case, Big Brutus is the better of the two forklifts over the long run.

When using intertwined attributes, you often use the final DPS comparison score to evaluate weight in the total comparison of data objects instead of the individual attributes.

Binary Searching

Binary searching is a mathematical method of searching that enables someone to find an exact unknown number in the fewest possible guesses. Why is this important to you as a game system designer? You are constantly trying to find the right numbers for your data objects. By using binary searching, you can home in on the number you need more quickly, saving valuable time when creating large amounts of data.

How Binary Searching Works

The way binary searching works is slightly unintuitive, but once you understand how it works, it becomes simple and fast to use. To perform a binary search, you need just a few pieces of information:

- **Range of viable numbers:** For a strict binary search, you need to know the largest and smallest possible numbers that could be correct. There are workarounds for this, as discussed later in this chapter, but they don't fit the strict definition of binary searching.
- **Feedback response:** This lets you know if your current number is too high or too low. Without knowing this, you really are flying blind. But if you can at least get this one clue, binary searching can work.

Let's consider a basic example of a higher/lower guessing game. This guessing game starts with someone saying, "I'm thinking of a number between 1 and 100. Guess a number, and I will tell you if the actual number is higher or lower." This game is the basis for all binary searches, and there is a "correct" answer to this opening question: In this case, the correct answer is 50. Why is this the correct answer? This is where the method becomes slightly unintuitive. The goal for the guess is not actually to find the right answer. The odds of doing that on the first try are 1%, which are far too low to use this as a viable method. Instead, the goal is to eliminate as many wrong answers as possible. To see why there is a correct answer, let's look at two scenarios involving the same guessing game:

- **Scenario 1:** You are trying to guess the right number, so you guess a favorite (or maybe random): 88. The response to this guess is "lower." From this, you can deduce that no number in the range 88–100 is the correct answer. So you have eliminated 13 of 100 options, leaving a

possible 87. Using a random guess method, you have no idea how many options you might eliminate and, therefore, how many options will be left. The goal of the game is to get the list of options down to 1—or as few as possible—so you can guess correctly. The fewer options left in the list, the higher the probability that the next guess will be correct.

- **Scenario 2:** Instead of guessing a random or arbitrary number, you can divide the total possible pool by 2, which means you guess 50. This way, if the answer comes back as “higher,” you have eliminated 50 options. If the answer comes back “lower,” you have also eliminated 50 options. By always guessing exactly in the middle of the remaining options, you can always guarantee that you will eliminate 50% of the incorrect options. You also might get lucky and guess correctly, which would save even more time, but that is not the goal.

By using the method described in Scenario 2, you can tackle even very large numbers of options with very few guesses. Let’s take the above example to its conclusion:

Range: 1–100

Guess 1: 50

Response: Lower

New range: 1–49

Guess 2: 25

Response: Higher

New range: 26–49 (where the halfway point is 36.5, which is not a whole number)

Guess 3: 36 (though 37 would also be a valid choice here)

Response: lower

New range: 26–36

Guess 4: 31

Response: Higher

New range: 32–35 (where the halfway point is 33.5, which is not a whole number)

Guess 5: 33 (though 34 would also be a valid choice)

Response: Higher

New range: 34 and 35

Guess 6: 35 (though at this point you can guess either and know that there is only one more option)

Response: Lower

Guess 7: 34

In this example, out of 100 possibilities, where no guesses were lucky enough to be correct, it was possible to find the exact answer in seven guesses. This is the maximum number of guesses possible for this range using a binary search. Using this method is dramatically faster, on average, than guessing arbitrary numbers in the range, and it saves over 90% of time needed to guess through the possibilities. If you apply binary searching everywhere you possibly can in a game, you are likely to find the answer you need in as little as one-tenth the amount of time it would take to guess the right answer.

[Table 12.13](#) shows the range of possible guesses and the maximum number of guesses needed for that range when using a binary search.

Table 12.13 Binary searching guesses for various ranges

Options	Maximum Guesses
1	1
2	2
4	3
8	4
16	5
32	6

64	7
128	8
256	9
512	10
1,024	11

You can see in [Table 12.13](#) that adding twice as many options to the range increases the number of guesses needed by only 1. The more options there are to guess from, the more powerful the binary search method becomes.

How does binary searching apply to system designers? You actually play the higher/lower game at work all the time. When tuning data numbers, you are constantly guessing what data numbers are right for each attribute given to the data objects. The higher/lower feedback you get comes from the feel of the changes in testing.

Binary Searching Example: Boss Fight

Let's say you are designing a boss character for a difficult fight that is going to take place in a game. You know what hit points the player character has and how much damage the player character can do. This gives you a frame of reference for how difficult to make the boss character. You also know how often you want players to fail at the fight. This provides the frame of reference for whether you have guessed too high or too low. So you can guess a range of total hit points that you might want the boss to have, from a minimum you would safely know is too weak to a maximum you can safely know is way too much. Then you apply the binary search method to make a guess and test. If players are beating the boss too often, this equates to the response "higher." If the players are losing more often than you want them to, you have received the feedback "lower."

Binary Searching Example: Jump Distance

One of the most classic mechanics in platformer games is having the player character jump over a pit to get to the other side. If player characters are easily able to jump the pit each attempt, the pit loses its challenge and

becomes a chore; on the other hand, if the player characters are not able to clear the pit in a jump, the game becomes impossible to complete and breaks testing. The difficulty in this case shows you the range. The minimum is the smallest amount of jump distance needed to clear the pit by only a pixel. The maximum is the jump distance that so easily clears the pit that players fail to see any challenge in the obstacle. With this range, you can again apply a binary search and find the exact amount of jump distance that allows players to feel challenged by jumping over the pit without being frustrated by it.

Lacking a Viable Range

For a true binary search, you need to have the exact minimum, exact maximum, and accurate higher/lower feedback. However, you don't always get all these factors when trying to find the right numbers for data, but you can still apply the principles of binary searching to situations where you have very little information. If you don't know the minimum or maximum of a range, you can still test for the feeling of too much or too little, and you can use the rule "double it or halve it."

For example, say that you are working with brand-new game data, and nothing has been tuned yet, so you don't have a point of reference for determining a range that is too low or too high. You have an archer that you want to be able to shoot targets from far away—but what does "far away" mean? You have no idea to start with, but you do have an attribute variable called shot momentum that tells the physics engine how much momentum to apply to the arrow when the bow is fired. At this point, you would want to dig for any possible clues to what a good range might be. How big are the levels? How far away are enemy targets? You might or might not have answers to these questions, but for the sake of making the challenge more difficult, let's say you have none of this information at all.

To solve this challenging problem, you can start by making a very quick test level in the game engine with a single archer and a single target. Take some time walking the character up to the target and then further away. How far away feels like a practical distance? All modern game engines have methods for measuring game distance, and you can feel out the possible distances and make notes about them. At some point, the target will be too

far away to render onscreen, and that is likely too far. Take the extra time to work out the variables that are available to try to find some kind of range. When you know that range—or at least have a guess—you can move on to shooting at the target.

Because in this case you have no idea what your units of “shot momentum” actually mean to the game engine, you have to take a completely wild guess. Say that you guess 100, knowing you will certainly be far off, and then try it out to see what happens. In this example, imagine that you put in the value 100 and take a shot, but the arrow barely goes a meter. Now, you have some information to work with, and you double the shot momentum to 200 and test again. This time the shot goes further, but it is still too short. You double again to 400. This time the shot flies out of the game world. Congratulations! You have found your range and can now use a binary search between the known too-low value 200 and the known too-high value 400.

Taking the other possible branch in this example, you can put in the value 100. The arrow flies out of the world, far into the distance. For your next guess, you cut the value in half, to 50, and try again. Again, the arrow flies out of the world. Next, you cut the value in half again, to 25, and repeat. By doing this task repeatedly, it is entirely possible that you will find that the shot momentum was intended to be a decimal number, and your actual range might end up being 0.01 to 0.2.

Both of these very nebulous scenarios show how to use binary searching when you don’t know the top and bottom bounds to quickly home in on an appropriate range of values. As long as it is possible to test to see if a value is too large or too small, you can use binary searching to quickly determine what numbers will fit in your attribute range.

Naming Conventions

When making even moderately complex games, you will generate numerous game objects—and which could easily get into the hundreds and possibly into the thousands. One of the most basic requirements of any game object is that it needs a name. Coming up with hundreds or thousands of names for objects can be a tedious task if you don’t have a standardized

convention. Even more complex is tracking down and then working with a specific object that you have previously created that is floating in the sea of game objects. Even more challenging than that is finding and working with a specific object that someone else on the team has made. Try to imagine sifting through hundreds of objects that are numbered or named gibberish to try to find the one you are working on. It sounds awful—and it is awful. Now imagine doing this dozens of times a day, every day, for several months or even years. If you don't have a good naming convention, this task becomes miserable and dramatically slows down work on the game.

You know what the problem is, but what is the solution? In the end, the most important thing about naming is to have one naming convention—and only one—for the team. There are many ways to name game objects, and many conventions can work. The important fact is that the entire team needs to agree to the rules of the naming convention it uses.

Let's look at a naming convention that is data designer centric. It is made to help a data designer, who is usually the person most often modifying the data objects, find what they need quickly and keep the entire group of objects organized. To understand the system, first let's talk about the English language a bit. English is inherently ambiguous in the way it describes objects. Let's start a sentence in English as an example:

My game object is a big...

This is the start of a description, but it is not enough to give someone an idea what the object is. Let's add another more:

My game object is a big, fast...

This is still not enough for someone to know what the object is; it could be any of millions of different things. Let's add another word:

My game object is a big, fast, blue...

We are now three words into the description of the game object, and although we're narrowing down the range of possibilities, it's still

impossible to know what the object is. One final word clears it all up:

My game object is a big, fast, blue narwhal (see [Figure 12.1](#)).

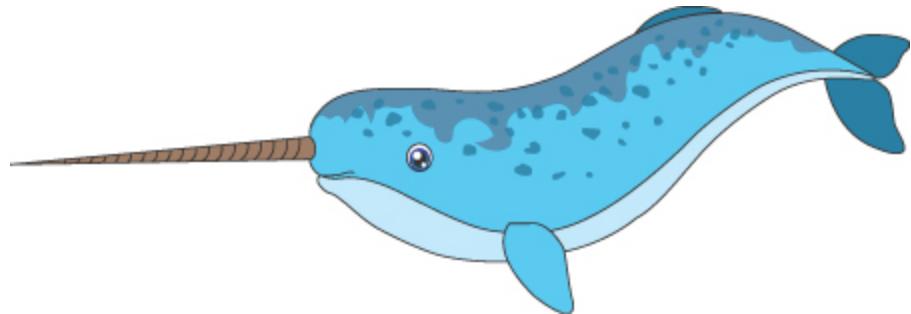


Figure 12.1 A narwhal

Aha! Four words into the description, and we finally have it: We are talking about a narwhal. The English description “big, fast, blue narwhal” does allow someone to picture what the object is, but it would not work well as a name in data design. Given that files automatically sort alphabetically on a computer, and most databases sort alphabetically by default, this name would sort by the first adjective, instead of by what the object is. Therefore, a basic English description like this won’t work very well as a name in a game.

In a very large collection of game objects, you would usually want to start names with a parent category. For example, you might start with the category sea mammals or, if you were playing the game as a narwhal, character. It is common to shorten the category name to a few letters for the sake of brevity and easier reading. If all category names are three letters long, for instance, then everyone knows that the object name starts at the fourth character. For this example, say that you are going to be playing as a narwhal, and you want to use the category prefix Cha (for character).

The following are some of the game object categories that are commonly used in games:

- **Cha:** Character, for playable characters

- **Npc:** Nonplayer friendly characters
- **Bad:** Enemy characters
- **Pow:** Power-ups, or useful boost items
- **Wep:** Weapon items
- **Amr:** Armor item (Note that this category name is Amr, not Arm, because *arm* is already a real word with a meaning.)
- **Vhc:** Vehicles

One of the first jobs of a system designer to create game objects is to create and maintain the names and definitions of object categories. Any lengths or convention will work, but for ease of reading, try to keep names fairly short and easy to understand. In creating a naming system, you want to find a good balance between getting very specific (with too many categories) and being too vague (and lumping together too many objects).

If your naming convention is for a large game with thousands of objects, you might follow the category with the subcategory. In this case, let's assume the game is smaller and follow the category abbreviation with a noun that describes the game object. If the object is a narwhal, its name might therefore look like this:

ChaNarwhal

or

Cha_Narwhal

These two naming methods—using CamelCase and using an underscore—are both valid. CamelCase tends to be a bit shorter, but underscores can be a bit easier to read. For a small game, you don't really need to worry that your game object name lengths will get out of hand, so you can use underscores. Both CamelCase and underscores work well, so you can choose either one; the important thing is to stick with the convention you choose.

Note

Why don't you just use a space in an object name? While many game engines can handle spaces in names, some can't. When referring to game objects in code, spaces can actually break many coding languages, and the compiler will interpret the space to indicate a new object. Therefore, you should avoid using spaces in any names as a generally good practice.

If you are making a rather simple game, the object name `Cha_Narwhal` might be good enough. If, for example, there is only one type of narwhal, you would not need to go any further. However, say that you are creating a game that has several narwhal objects to choose from. In this case, you need more information to distinguish them from each other. At this point, you need to decide exactly how much granularity you need in order to describe your set of data. For the sake of this example, say that you have more than a dozen narwhals, so you need a name that is quite detailed. In this case, the next part of the name should be one word that describes the most important attribute of the object. It's important that it be a description, and not a value, because it's very likely that attribute values will change multiple times. You would not want to rename all your objects each time an attribute value changes. Instead, you want to describe objects in a relative manner. For this example, say that you have two class sizes of narwhal: big and small. For an object in the larger class, the name might now look like this:

`Cha_Narwhal_Big`

Now you can add the next most important attribute in the game. For this example, you might determine that speed is the next most important attribute and split the group into two classes: fast and slow. For a big narwhal character in the fast group, the name would now look like this:

`Cha_Narwhal_Big_Fast`

Note

How do you know which attributes are the most important? That's up to you as the system designer to determine. In many games, it is obvious which attributes are most important. For example, in a racing game, speed is likely to be the most important attribute. However, there are a lot of ways you could go that would not be wrong. The important thing is to make a decision and then stick to it.

The name is now very close to a full description, but in this game, the characters can come in a variety of colors, so you want to add a final descriptor to make each name unique. For a blue narwhal with all the preceding characteristics, the name looks like this:

Cha_Narwhal_Big_Fast_Blue

This name may not, and likely will not, reflect all the attributes associated with this object. For this naming convention, you are not trying to cram in each and every attribute; instead, you want to pick just enough of the most important attributes to distinguish the specific game objects from each other. Cha_Narwhal_Big_Fast_Blue is quite a different name than the original description “My game object is a big, fast, blue, narwhal,” but it contains the same information, presented in a way that is easier for computers and data designers to work with. The order that this naming convention makes possible becomes apparent when you look at a larger group of objects, all names using this naming convention:

Cha_Narwhal_Big_Fast_Blue

Cha_Narwhal_Big_Fast_Green

Cha_Narwhal_Big_Fast_Red

Cha_Narwhal_Big_Slow_Blue

Cha_Narwhal_Big_Slow_Green

Cha_Narwhal_Big_Slow_Red

Cha_Narwhal_Small_Fast_Blue

Cha_Narwhal_Small_Fast_Green
Cha_Narwhal_Small_Fast_Red
Cha_Narwhal_Small_Slow_Blue
Cha_Narwhal_Small_Slow_Green
Cha_Narwhal_Small_Slow_Red
Cha_Whale_Small_Fast_Red
Cha_Whale_Big_Fast_Blue
Cha_Whale_Big_Slow_Green
Cha_Whale_Small_Slow_Red

This naming convention groups together all the narwhal characters so they're separate from the whales. The fast ones are separated from the slow ones, the big ones are separated from the small ones, and so forth. Having a strict naming convention like this makes it dramatically more efficient to find the information you need and then to pass that along to anyone else who might need it on the team.

Note

Although we have not talked about what the characters in this game actually do, the names provide some pretty big clues. In addition, the names give you enough information to allow you to think about how this theoretical game might play. You can guess that it's likely not a racing game since speed is not the most important attribute. You can guess that it's not a color-matching puzzle game since color is the least important of the attributes listed. Looking at just the name, you might guess that this is a game about surviving at sea or possibly some form of battle. In both of these cases, size would be very important, with speed coming in a close second.

This is just one example of many different ways to form a good naming convention, and it is a solid starting point from which you can create the

right convention for your game. In the end, the most important thing about a naming convention is that your team has one and that all the team members stick to it strictly.

Naming Object Iterations

Static, single-use names are not the only names in a game. Many objects need to have new names as they are revised. To properly organize all those names, you need to include a system of naming specific iterations of objects. It might be tempting to label a new version of an object *new*, but doing so is highly problematic. The good news is that there are better ways to get the desired effect.

The Problem with “New”

When a designer remakes an object, there is a temptation to name the new object with the word *new* (for example, `New_Cha_Narwhal_Big_Fast_Blue` or `Cha_Narwhal_Big_Fast_Blue_New`). The problem is that there is also a very real possibility that this overhaul will be repeated, perhaps many times. What happens when you redo an object whose name was already changed to include the word *new*? Do you name it `New_new`, `new2`, or something else? Such options are not clear, so it’s best to avoid using the word *new* in names altogether. Instead, it’s better to use a system of iteration notation, if needed.

Note

Many source control software packages handle iteration naming for you. For example, Perforce keeps track of multiple iterations of an object with the same name by using timestamps.

Note

Just as you should avoid using *new* when naming iterations, you should avoid *rework*, *modified*, *updated*, *changed*, and any other description that is vague.

Iteration Naming Method 1: Version Number

Many programs do not keep track of timestamps and do not keep old iterations of objects or files. What if you want to keep an older object or file but use a newer version? The easiest way to iterate an object or file in this scenario is to add a numeric suffix that indicates the version of the object. For example, if you were to rebuild your fast blue narwhal character three times, you might end up with the following game objects:

Cha_Narwhal_Big_Fast_Blue_01

Cha_Narwhal_Big_Fast_Blue_02

Cha_Narwhal_Big_Fast_Blue_03

By using this method, you can keep the old versions for reference but use the most recent version. You can also clearly explain to anyone on the team which version of the object you are referring to by using its number. This method works well if you are planning on doing a small number of versions over the course of a project.

Iteration Naming Method 2: Version Letter and Number

If a project is going to be large and the object or file in question is likely to get remade many times, you can use a more detailed versioning method than the version number method just described. For this method, you use a letter that represents major changes and a number that represents minor changes. *Major change* is a subjective term that you need to define for a specific project, but it generally means a big, potentially problematic change that should stand out at a glance through a list. If you were to take a character in a new direction or delete mechanics from a weapon or completely rebalance a vehicle, these could be considered large changes that warrant new letters. Small value tweaks or superficial changes would get new numbers. Using this method, you could get this list of iterations:

Cha_Narwhal_Big_Fast_Blue_a01

Cha_Narwhal_Big_Fast_Blue_a02
Cha_Narwhal_Big_Fast_Blue_a03
Cha_Narwhal_Big_Fast_Blue_a04
Cha_Narwhal_Big_Fast_Blue_b01
Cha_Narwhal_Big_Fast_Blue_b02
Cha_Narwhal_Big_Fast_Blue_c01
Cha_Narwhal_Big_Fast_Blue_c02
Cha_Narwhal_Big_Fast_Blue_c03
Cha_Narwhal_Big_Fast_Blue_c04

In this example, it is easy to spot where major changes occurred. This is helpful in tracking the changes made to an object, and it is also helpful if problems arise in a major change. Instead of sorting through versions, you would know to look back at the last major revision, such as c01, to start your search for the culprit.

Special Case Terms

Special case terms are used for temporary objects or files during testing or development. Some special case terms illustrate the purpose of the keyword.

DeleteMe

You can probably guess what the special case phrase `deleteMe` means. You are likely to very often need to make game objects or files in order to run tests or sketch out ideas. You don't want those objects hanging around forever, taking up space or causing confusion, so it is a good idea to name them in a way that clearly illustrates their temporary nature. Many game designers use `deleteMe` for this purpose. For example, if you wanted to test the idea of a giant character for the narwhal game, you might name it like this:

Cha_Narwhal_Gaint_Fast_Blue_deleteMe_a01

Many other keywords would work just as well as `deleteme`, but this word also has instructions for use built in, so it does a fantastic job of being short and clear.

Deprecated

Like `deleteme`, the special case word `Deprecated` also means exactly what it says. However, whereas `deleteme` indicates that the object or file is a temporary file that should be deleted, `Deprecated` indicates that this file or object has been replaced by a newer version, and you want to keep it around (perhaps as a backup). For example, if you updated your narwhal completely, you could eliminate all other parts of the naming conventions and use this:

`Cha_Narwhal_DDeprecated`

Test

Another special case word that you can use in object names is `test`, which also does what it says. `Test` works in a similar fashion to `deleteme`, but it does not indicate that the object should be deleted. When you see the word *test* in an object name, you know that you should keep the object but should not refer to it for any finalized information.

Date or Time

In some instances, you might want to add the date to an object to show a progression of iterations at specific times. In such cases, you can add the date at the end of the name. How specific you get with the date depends on how exactly you need to know when the object was created. Regardless of the specificity you choose, you should present a date in descending order, starting with year and working down through smaller and smaller units of time (such as month, day, hour, minute). Doing so ensures that dates will be ordered properly when sorted. For a test narwhal object created October 2, 2020, you might use this name:

`Cha_Narwhal_Test_2020_10_02`

Using the Handshake Formula

After you start creating game objects, you need a method of comparing them. For very simple games with few objects, this is simple: You take each variation of an object and compare it to every other variation. For example, if you were making a fighting game, you would have the characters fight against each other to ensure that they are all balanced in a fair way. If you were making a car racing game, you would race each car against all the other cars. If you were making an action game, you would want to compare each enemy AI type against every other enemy. This method of “brute force” testing and balancing works fine for very small games with a small number of variations. However, problems start cropping up when the number of game objects you create increases. Doing a full round of testing and balancing between two objects can take an hour or more for even a simple pairing—and that’s just for one round of testing. Making a full game often involves dozens of rounds of testing, tweaking, balancing, and rebalancing.

The first step in knowing how to deal with testing, tweaking, balancing, and rebalancing is knowing just how big the problem is. This is where the handshake formula comes in. The name *handshake formula* is based on an old mathematical riddle that goes like this: “A group of doctors are in a room together. To say hello, each doctor needs to shake the hand of every other doctor. How many handshakes does it take for any given number of doctors to shake every hand needed?”

You might wonder what this riddle has to do with balancing game objects. If you think of the handshakes in the riddle as a metaphor for how you test and balance objects against each other, it might start to make sense how useful this formula is to game designers. We can rephrase the riddle into language that is closer to home for game system designers: “A group of combatants are in a fighting game. Each combatant must fight every other combatant in the game to determine who is the best.” or “A group of cars need to race against each other to determine which is the best car. How many races would it take for all cars to race against each other?” While the outcome of the handshake formula might, on the surface, seem like nothing more than a bit of trivia or a curiosity, it is something system designers do constantly.

The handshake formula can be extended to larger examples (three-way races, group combat, and so on), but for now, we will focus on just two parties at a time. You need to think about how you will use the formula not for a single specific number but as a formula that you can apply to any number.

Any time you need to compare a number of objects to a group, you need to start with the size of the group. For the current example, let's start with four. To compare any two of the group of four, how many combinations would you need? The very simple way to find the answer would be to combine each of the four with all four. This, mathematically means, squaring the number. You can also do this graphically in a spreadsheet, as shown in [Figure 12.2](#); this is called a *possibility grid*.

	A	B	C	D	E
1	1	2	3	4	
2	1	X	X	X	X
3	2	X	X	X	X
4	3	X	X	X	X
5	4	X	X	X	X

Figure 12.2 Possibility grid

Here you can see that if you list all objects in a row and also in a column, you can check off how many combinations there will be; this is the same as multiplying the number times itself, or finding the square. In this case, 4 squared is 16, so there are 16 combinations. This seems like a lot of combinations needed to do a comparison with just 4 objects, and indeed it is. There are several redundancies in the simplistic view of squaring the

number that you can immediately eliminate. The first of which is a game object compared to itself. You don't need to compare identical cars to know that they are the same, for example. If you look at every point where you would compare an object to itself, you find the clear pattern illustrated in [Figure 12.3](#).

	A	B	C	D	E
1		1	2	3	4
2	1	X	X	X	X
3	2	X	X	X	X
4	3	X	X	X	X
5	4	X	X	X	X

Figure 12.3 Unneeded combinations

Object 1 is compared to itself in cell B2, Object 2 is compared to itself in cell C3, and so on in a diagonal pattern through the chart. In a fighting game, you know that Combatant 1 will be equal to Combatant 1, so you can eliminate the need to test that combination. Therefore, you can eliminate every point on the graph where an object would be compared directly to itself. In the case shown in [Figure 12.3](#), this is 4 combinations. In fact, for any comparison, the number of eliminations will equal the number of objects you are comparing. So, if you were to compare 6 objects, you could eliminate 6 combinations, and if you were to compare 1,000 objects, we could eliminate 1,000 combinations. With this in mind, you need to update your formula to get rid of those unneeded combinations. This is the update formula:

$$(\text{Number of game objects})^2 - (\text{Number of game objects}) = \text{Unique comparisons}$$

or:

$$G^2 - G = \text{Unique comparisons}$$

or, in this example:

$$4^2 - 4 = 16 - 4 = 12 \text{ unique comparisons}$$

While this formula removes the redundancy of comparing objects to themselves, there is one more step you can take to make your comparisons fully efficient. In a fighting game, if you compare Combatant 1 with Combatant 2, do you then need to compare Combatant 2 with Combatant 1? No. You already did that comparison, but in the opposite order, and because order doesn't matter in the comparison, you can eliminate any combinations that you have already done in the opposite order. [Figure 12.4](#) shows this graphically in a spreadsheet.

	A	B	C	D	E
1		1	2	3	4
2	1		X	X	X
3	2	X		X	X
4	3	X	X		X
5	4	X	X	X	

Figure 12.4 Redundant combinations

Both highlighted cells in [Figure 12.4](#) are making the same comparison because they are both comparing object 1 with object 2, and you need only one of these comparisons. You need to know mathematically how many of these redundant comparisons are left in the chart. Because you are comparing every object with every other object, and all the objects are listed in both dimensions of the chart, you will get twice as many matchups of each object as needed. For every time Object 1 is compared to Object 2, Object 2 will be compared to Object 1. The same goes for each other pairing. This means you can eliminate half of the pairings because they are comparing the same two objects but in the opposite order. [Figure 12.5](#) shows the redundant comparisons that you can eliminate.

	A	B	C	D	E
1	1	2	3	4	
2	1		X	X	X
3	2	X		X	X
4	3	X	X		X
5	4	X	X	X	

Figure 12.5 Removing redundancies

Because all the highlighted cells in [Figure 12.5](#) are redundant, you can remove them from your count of combinations that you need to test and balance. Mathematically, this means you can divide your total combinations by 2. This leads to the final formula:

$$(G^2 - G)/2 = \text{Total combinations to test}$$

or, in this case:

$$(4^2 - 4)/2 = 6$$

To validate that the formula is working, you can count the number of *X*s in your chart after eliminating all redundant combinations (see [Figure 12.6](#)).

	A	B	C	D	E
1		1	2	3	4
2	1		X	X	X
3	2			X	X
4	3				X
5	4				

Figure 12.6 Valid remaining combinations

As you can see, 6 is indeed the number of valid combinations remaining. Now that you have the needed formula, you can input any number into it and find out exactly the number of combinations you need to compare your entire set of objects. As one more example, [Figure 12.7](#) shows 20 objects and the combinations you need to test.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
2		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
3			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
4				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
5					X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
6						X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
7							X	X	X	X	X	X	X	X	X	X	X	X	X	X	
8								X	X	X	X	X	X	X	X	X	X	X	X	X	
9									X	X	X	X	X	X	X	X	X	X	X	X	
10										X	X	X	X	X	X	X	X	X	X	X	
11											X	X	X	X	X	X	X	X	X	X	
12												X	X	X	X	X	X	X	X	X	
13													X	X	X	X	X	X	X	X	
14														X	X	X	X	X	X	X	
15															X	X	X	X	X	X	
16																X	X	X	X	X	
17																	X	X	X	X	
18																		X	X		
19																			X		
20																				X	
21	20																				

Figure 12.7 Large set of combinations

You can see that no matter how large or small the number of objects, the pattern repeats. If you plug in the new number to the formula, you get the following:

$$(20^2 - 20)/2 = 190 \text{ combinations}$$

If you wanted, you could count the number of Xs in the chart and arrive at the same answer to verify, but doing so would take more time than using the combination formula. There is another way to do this formula that takes a little longer but that shows a very interesting pattern. To illustrate, let's solve for the numbers 1 through 20 (see [Table 12.14](#)).

Notice that if you add the number of objects with the number of combinations, you get the number of combinations for the next greater count of objects. This means that if you know the formula for any number, you can quickly figure out the result for the next larger number by adding the number of objects to the number of combinations. If you wanted to extend this chart up to 21, therefore, you would add 20 to 190, for a result of 210 combinations.

When you know the number of combinations you are dealing with, you can use this information to more accurately calculate how much effort will be needed to check and balance these combinations. For example, if you know that it takes an hour to properly test and tune a single combination, you can figure out how many hours you need to account for in total.

Table 12.14 Possible combinations

Objects	Combinations
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45
11	55
12	66
13	78
14	91
15	105

Objects	Combinations
16	120
17	136
18	153
19	171
20	190

In addition to using the combinations formula for testing and balancing, you can use it in the following ways:

- To make a round-robin tournament where every player plays every other player
- In a fighting game, to create a special victory animation for each combination of combatants
- In a racing game where a voiceover announcer says a unique line for each pairing of cars before the cars start the race
- In a superhero game where two of heroes combine their powers to form a special power that is unique to that combination of characters
- In a fantasy RPG, where each combination of character classes has a unique bonus when played as a team
- In a combat game, to create a special animation for each combination of attack and defense

In addition to using the combinations formula to manually figure out how many combinations you need to test, you can use a spreadsheet shortcut function that does this math for you. This function, COMBIN, takes two arguments: the number of objects in the dataset and the number of objects in a combination. For all of the examples in this section, that second argument has been 2 because we have been combining only two objects at a time. However you can combine more than two objects with one another to get a significantly larger number of total combinations. Using a spreadsheet

formula makes it possible to easily calculate these larger numbers of combinations.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore concepts covered in this chapter:

- For some real-world examples of game objects, list their attributes and try to assign weights to those attributes. Try to do this for a variety of different objects and use the objects that you have created from previous chapters.
- Create a naming convention for game objects—either objects you have made or preexisting games objects. Take the time to figure out how many attributes need to be contained in a name and in what order they should appear.
- Use the handshake formula to determine the number of combinations in several actual scenarios (for example, your favorite upcoming sport or a game tournament you follow).

Chapter 13

Range Balancing, Data Fulcrums, and Hierarchical Design

This chapter covers three very important data concepts: range balancing, data fulcrums, and hierarchical design.

With these concepts, you can start creating a larger amount of data objects that are balanced and ready to go into your game. By learning these foundational principles first, you will be able to avoid many of the most common pitfalls that new data designers experience when building their first set of data.

Range Balancing

Many numbers you use for data object attributes have physical or virtual constraints. For example, miles per hour and weight in pounds (or kilograms) are known measurements that are well entrenched in our audiences. You want players to easily understand the measurement of attributes in your game, so you are likely to use such known systems of measurement when displaying an attribute of a data object instead of using abstract measurements.

On the other hand, using your own scale that meshes well with your systems and spreadsheets can help you, as a game system designer, do your job more quickly and effectively. For example, if you want to express the weight of an MMA (mixed martial arts) combatant, you can take either of the two following approaches:

MMA combatant: 105–265 pounds

or:

MMA combatant: 0–100 in game “weight units”

Pounds are much easier for players to grasp than “weight units” because players are used to the real-world analog of “weight class” in the sport. However, using pounds makes doing calculations on the scale of weight more cumbersome. Where is the middle? You don’t know. What is a mid-high value? You don’t know that either. You could use a spreadsheet to calculate these values, but it’s not very easy for us to visualize the scale from the least to the most and see how each score will compare to other data objects.

If you use an in-game unit with a standard scale of 0 to 100, you can much more easily do calculations. Where is the middle? It’s 50. What is a mid-high value? It’s 75. What is a mid-low value? It’s 25. You can easily find all these values in your head very quickly. The problem with using game units, as mentioned earlier, is that they do not have meaning to the player. How much does a man weigh if he is 25 in game units? Will this make sense to a player?

An ideal system would allow you to use a standardized range such as 0–100 internally to do calculations and comparisons and provide a different scale that more closely matches a real-world analog for the player—and the two would work well together. Fortunately, this type of ideal system exists, and it is called *range balancing*. Range balancing provides a bridge between the two methods of numbering, and it also has several other benefits:

- Normalizes numbers so they are easier to work with in spreadsheets
- Works around odd units of measurement that are difficult to work with (such as radians and force)
- Eliminates “ripple effect” balancing
- Decouples the work of the system designer and data designer(s)
- Makes balancing easier on a larger scale

How Range Balancing Works

Range balancing starts by finding the data range in the player-facing scale. This range can have any minimum and any maximum, and the range may at

some point change. At first, all you need is an educated guess about the range. For example, if you were making a racing game and you wanted to find the top speed of race cars, you could simply estimate that the slowest race car might go around 110 miles per hour and the fastest would go 190 miles per hour. Don't worry about being accurate at this time, as one of the benefits of range balancing is that you can change it painlessly later on.

After you estimate the minimum and maximum, subtract the minimum from the maximum to find the data range. You can now express every attribute value as a percentage of this range. Let's dive in a bit deeper to better understand all the concepts involved in range balancing.

To better understand the data attribute range, let's look at a very simple example of the height of ogres. Say that you want a variety of ogres in your game, and you want them to be of various heights so they seem more dynamic and natural. However, it would not make sense for this game to have very tiny ogres, and the levels can only hold ogres up to a certain size. For the player-facing portion of this, you can use a real-world unit, the foot, as your measurement. Through discussion with the team, you have estimated that the smallest ogre that can still feel "ogre like" would be 6 feet tall. That is a very small ogre, but it is big enough to be a viable ogre. While the game engine and levels could technically handle smaller ogres, you simply don't want them because anything smaller than 6 feet would feel odd and out of place. So you choose an ogre minimum height of 6 feet.

On the upper end, you need to keep in mind some technical constraints. For example, the level designers have been making doorways with a minimum height clearance of 8 feet. You want ogres to be able to fit through the doors without any special crouching animations, so this can guide you on the maximum height. This also works well with your vision of what a big ogre is, so there is no need to add further complications to production. Finally, if you look at your existing hypothetical hierarchy you find that level geometry ranks above secondary monster AI for this particular hypothetical game. This seals the deal: If ogre height might conflict with level geometry, it is on you to limit the ogre instead of asking for geometry changes.

Once you have the minimum and maximum ogre height as 6 feet and 8 feet, you can get the total range of ogre height by subtracting the minimum from

the maximum—in this case, 2 feet. So you know all ogres in the game will fall somewhere in this 2-foot range. You can see this illustrated in the graph in [Figure 13.1](#).

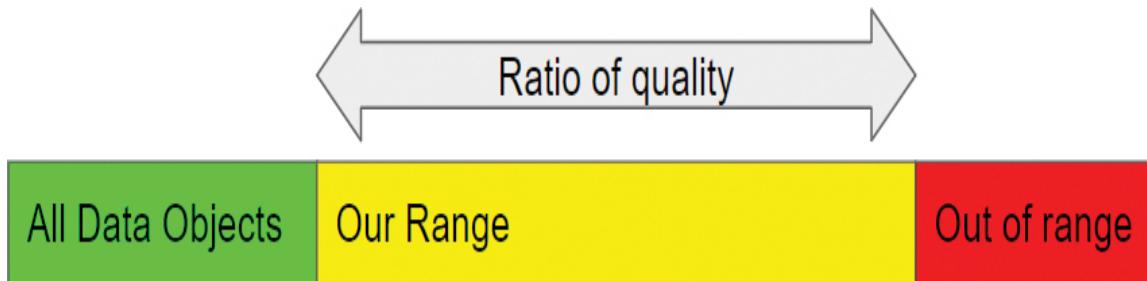


Figure 13.1 Range of acceptable numbers

The next step is to turn this range into a number that is normalized and easier to work with behind the scenes. The most commonly used range is 0% to 100%, and you can stick with this unless there is a good reason not to (see [Figure 13.2](#)).

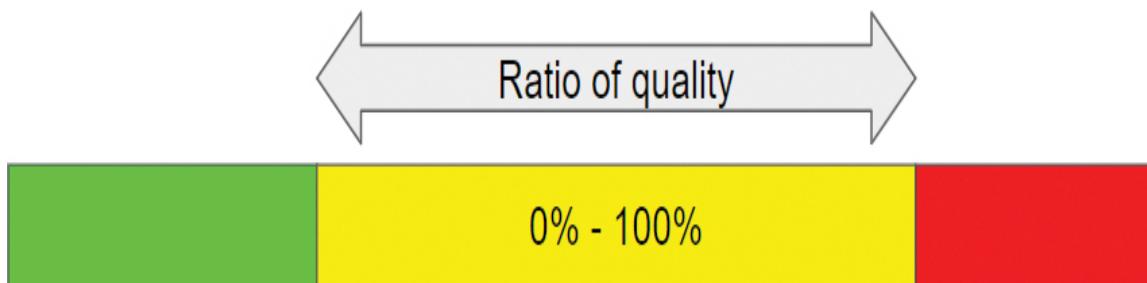


Figure 13.2 Range illustrated as percent

Knowing the maximum, the minimum, the range, and the scale, you now have all the information required to make a wide variety of ogres that fit within your goals. To use this information together for a translation, you need to apply a single formula, and then you can reuse the data many times. The formula for translating between developer scale and user scale is as follows:

$$((\text{Max} - \text{Min}) \times \%) + \text{Min} = \text{Numeric value}$$

This might seem like a complex algebraic formula, but it is much simpler than it looks. Let's break it down using the ogre example. We can start with a very simple example of an ogre that is rated at 50%. You already know the rest of the values:

Percentage = 50%

Range = 2 (feet)

Max = 8 (feet)

Min = 6 (feet)

Value = $((8 - 6) \times .5) + 6$

or

Value = 7 (feet)

Thinking of this intuitively can help to clarify what is happening. If you have a linear range of ogres from 6 feet up to 8 feet, then an ogre exactly in the middle is 7 feet tall, as in [Figure 13.3](#). When doing this intuitive calculation in your head, you are actually going through the range balancing formula. It's likely that you have used this formula countless times in your life but never actually broken it down into parts to examine how it works.



Figure 13.3 Example Ogre height range

To get deeper into the range balancing formula, let's use a larger, more sophisticated example. This example shows how to balance the run speed attributes of several different kinds of people. In this example, you start with descriptions that don't have literal numbers, as shown in [Figure 13.4](#).

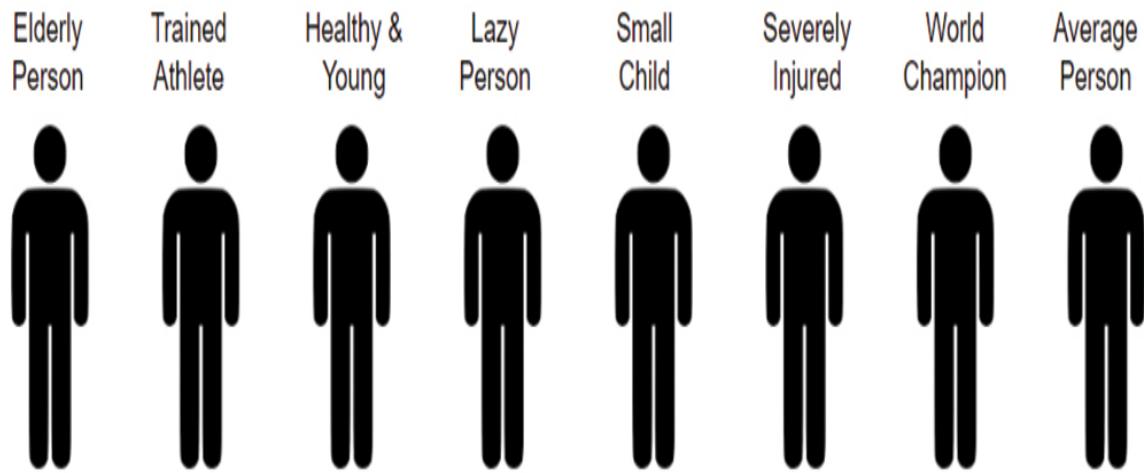


Figure 13.4 Example character types

In [Figure 13.4](#), you can see a wide variety of characters that will have a wide variety of top run speeds. Next, you need to quantify them. Take a moment before moving on and read the descriptions of each of the characters and think about how fast each could run. Write down a percentage from 0% to 100% for each of them, indicating how fast you think they would be in comparison to each other. Importantly, while you can think about this question in terms of miles per hour if it helps, you shouldn't write down actual miles per hour; just consider the relative percentages for the characters for now. When you have your numbers written, take a look at [Figure 13.5](#), which shows the percentages I assigned.

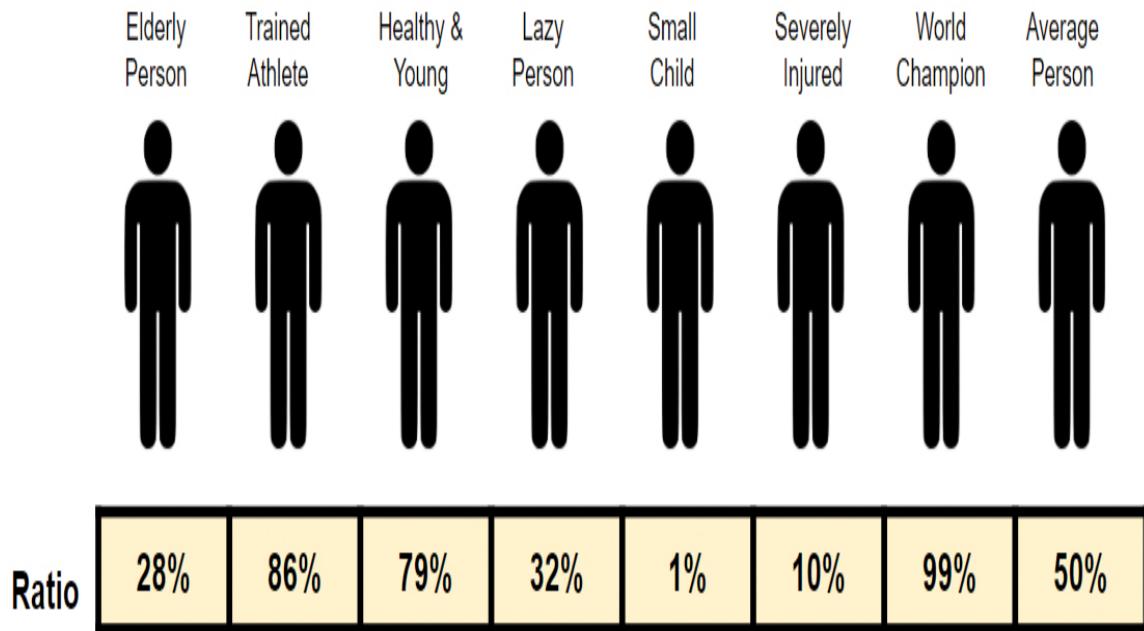


Figure 13.5 Character types with ratio listed

Do you agree with these numbers? Did they closely match yours? More importantly, did you use the same rank order from slowest to fastest? If not, don't worry; you did nothing wrong. There is really only one character here who has what could arguably be a “correct” answer: the average person, at 50%.

While doing this exercise, you might have had questions such as these:

- “Severely injured how? On the legs or arms?”
- “How lazy is this lazy person?”
- “What kind of elderly person?”

If you did, congratulations: You are thinking like a system designer! Translating vague concepts into numbers is a difficult thing to do, and it always involves ambiguity. When groups of system designers go for lunch on a workday, they often bicker and debate these exact topics, which is part of how they do their jobs well.

In some cases, asking follow-up questions is exactly what you want to do. If a level designer asks you to put in character design for a “lazy person,” you

may want to ask “Lazy how? Physically out of shape? Low motivation? Something else?” Other times, you very well might be handed a vague list like this and be expected to fill in the gaps yourself. Fortunately, this system is flexible, and you can go back and tweak it later.

After all the thought and debate, when you have a set of percentages recorded for the characters, it’s time to apply the range balancing formula to these percentages to see what they translate to. To do this, you need a minimum and a maximum. Just making a guess at it, you could say that a very slow person could move at 3 miles per hour. That’s very slow. Next, we could estimate that a very fast person might be able to run up to 20 miles per hour. Very few humans have ever run faster than that in the real world. When you assign your slowest runner the lowest value (min = 3) and your fastest runner the highest value (max = 20), you have all the data you need to apply the formula. Your spreadsheet can easily and quickly calculate the speed, in miles per hour, for all the other characters based on the min, the max, and the percentages. In this case, you get the set of numbers shown in [Figure 13.6](#). Once you have these numbers, it’s time to test them.

	Elderly Person	Trained Athlete	Healthy & Young	Lazy Person	Small Child	Severely Injured	World Champion	Average Person
Ratio	28%	86%	79%	32%	1%	10%	99%	50%
Min= 3, Max = 20	8	18	16	8	3	5	20	12

Figure 13.6 Example character with ratio and real value

Who Adjusts What

Once the data objects go into the testing phase, it becomes possible to break up the responsibility for data tuning into two separate tasks:

- **Systemic balance:** Systemic balance controls the feel and function of the entire set of data objects, without bogging you down with the minutiae of individuals.
- **Individual balance:** Individual balance is responsible for making each data object feel just right, while staying within the confines of the system as a whole.

On small projects, a single person can handle individual balance, but for a large game, it would be impossible for a single person to do all the work of balancing all the data objects. For example, an MMO might have hundreds of different characters and thousands of objects. In such situations, range balancing can help dramatically. When you have the min and max separated out from the individual data, you can adjust them independently of any individual data object. This can have a huge impact on the game and the actual production life of the team.

Say that you are working with human run speed, and testing has revealed that all the characters feel slow. It turns out (and this is actually true in video games) that for the game to feel right, many actual physical traits need to be heightened to unrealistic levels. Even in realistic games, characters run faster, jump higher, hold their breath longer, take more damage, and do many other superhuman feats regularly—and this is what “feels right” to players. What this means, practically, is that there is one person responsible for the minimum and maximum value, since those two values alone drive the feeling of all the objects in the game. This is typically the lead system designer, whose job is to ensure that the game as a whole feels balanced. Say that your testing group finds that the numbers are simply too low. By doing some more in-game testing, you discover that better numbers for minimum and maximum are $\text{min} = 5$ and $\text{max} = 40$. Using the range balancing formula, you can tweak the min and max

numbers, and instantly all of your humans' run speeds are updated to new numbers (see [Figure 13.7](#)).

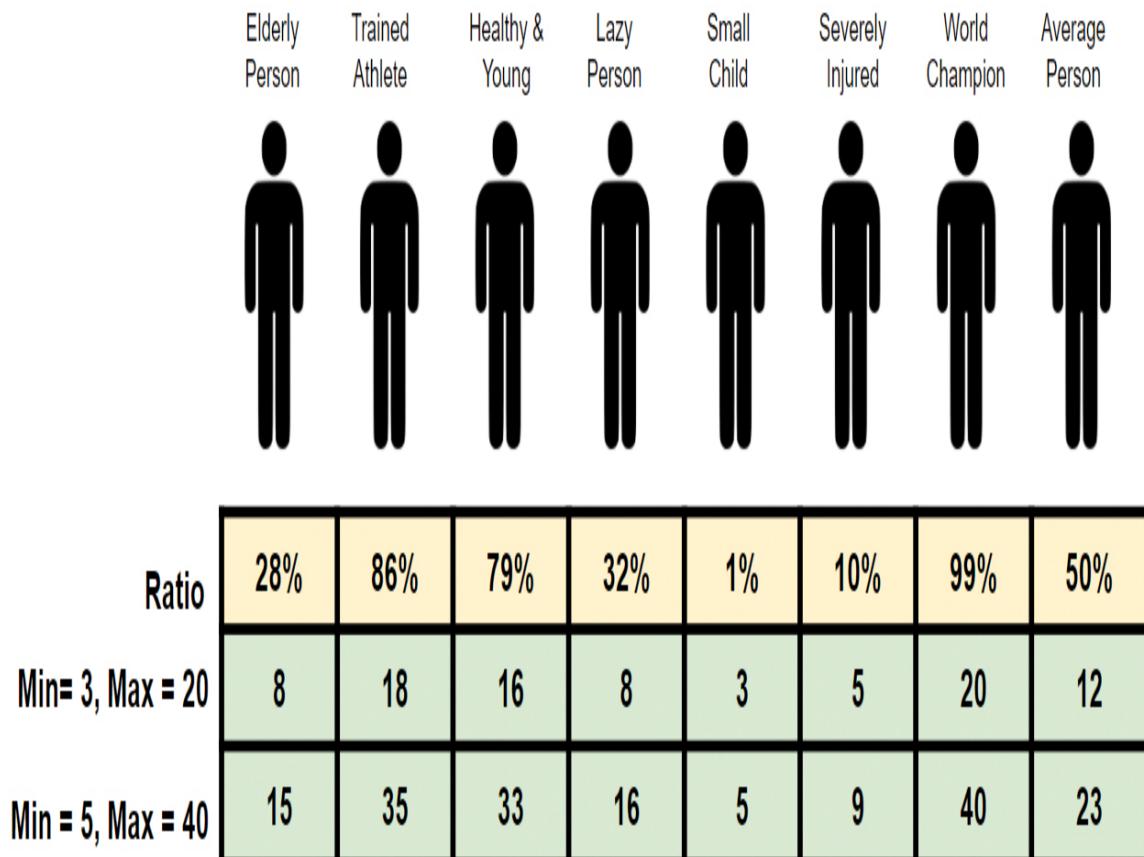


Figure 13.7 Example character with modified ratio

In [Figure 13.7](#), you can see the original percentages, the first pass at numbers, and the newly updated values that you arrived at through testing. There are several things to notice in this example:

- Numbers at the bottom of the scale did not change nearly as much as did numbers at the top. This is fine and accurately reflects that the bottom of the range did not change much.
- Numbers at the top of the scale changed a lot. Clearly, this is where the testers had the issue. They might have been okay with an elderly NPC hobbling about or characters having special conditions (such as

severely injured) that make them slow. But when players heard that a character was a trained athlete, they wanted to see the character zipping around the level.

- The world champion has a percentage of 99% but has 100% of the max value. This discrepancy is due to rounding all the numbers displayed to players. As mentioned in [Chapter 9, “Attributes: Creating and Quantifying Life,”](#) you should avoid showing decimals or fractions to players. You might want to reconsider the percentage for the world champion character as well. He is a world champion, but might there be someone even better out there, lurking in the shadows? By giving this character a perfect score, you have not left any room for improvement.

When the team is happy with the limits of the system, the data designers can further tune and revise individuals. In terms of the game, the min and max always sit above individual characters, and individuals need to be adjusted to fit the scale. The good news is that most of this is done automatically as characters maintain their relative comparisons to each other, regardless of any changes to the min and max. Note that this tuning can now—and forevermore—be done independently of the data range as a whole. Large teams of data designers can work in concert and the lead system designer to create massive amounts of data that all naturally fall within the bounds of the desired system. For example, a level designer who asked for an elderly person might come to the data designer and explain that they want her to be very elderly and barely mobile. With this request, the data designer can easily change the one percentage variable from 28% all the way down to 0%. This still works within the designated range, requires no adjustment of the min or max, and satisfies that level designer.

I’d like to drill this concept home with one more example: In the early days of large-scale video games, every data object attribute had to be entered manually, and there was no such thing as a range balancing system. Imagine being a data designer on a game with thousands of characters and being asked to speed them all up. This would require you to work in a database multiple days, doing individual calculations or guesswork to add a few points to each. If someone then thinks you went too far, and wants you to slow them all down a bit again, you would have to do all the tedious work

again. This was the reality for years in the game industry; a few companies continue to do it this way today.

Data Fulcrums

The ancient Greek mathematician Euclid said, “Things which are equal to the same thing are also equal to one another.” The same method of thinking can be applied to large groups of data objects. If we try and compare data objects to a single, fixed example, we can in a way, compare all of the objects to one another without having to manually do so.

In [Chapter 12](#), “[System Design Foundations](#),” you learned about the handshake formula and saw that adding new objects creates an ever-increasing number of combinations. Creating more combinations requires more effort. Maybe even more importantly, having more combinations means much more testing and balancing time is needed for each iteration. So how are game designers able to make large sets of objects? How can an RPG have 100+ character classes? How can racing games have dozens of cars? Using brute-force creation and testing methods would require an army of system designers to keep up. This type of creation is impractical for even the biggest team, and it is completely impossible for smaller teams. However, many medium and small teams make large varieties of objects in their games. How do they do it? Again, there are many methods, but most of them can be boiled down to variations of a single philosophy: the use of fulcrums.

What Is a Fulcrum?

The dictionary definition for a fulcrum is “a thing that plays a central or essential role in an activity, event, or situation.” In game design, a *fulcrum* or *data fulcrum* is something you use as a single point of comparison so you can avoid needing to use the handshake formula method to create large numbers and varieties of objects. Any time you have more game objects than you can balance with simple brute-force testing, there is another good opportunity to include a fulcrum. A fulcrum is a game object that is going to be the center point of comparison for all other objects of its type. Often, this object doesn’t even actually appear in the game but is used exclusively as a balancing tool for the designer behind the scenes. What’s most

important about the fulcrum object is that it is the balancing center. It should not be the best or the worst. It should not be special in any way. That is, in fact, the entire point of the fulcrum object: It's the most basic example of a creation (or an object or item) to which you can compare every other object in the group.

For example, if you were making a large variety of swords, the fulcrum would be a basic sword. It would not be the best or the worst but the most central, average sword you could create. Some swords might be large and powerful but slow. Other swords might be smaller and faster than most. The fulcrum sword is right in the middle. Some swords might be enchanted or cursed. But the fulcrum sword is neither; again, it is right in the middle.

As another example, in a racing game, you might have cars that are fast but have poor handling. Other cars might have great handling but lower top speed than average. Some cars might be fast with great handling but very expensive. Still other cars might be cheap but slow and might have poor handling. The fulcrum car would be right in the middle of all of these things: moderate speed, handling, and cost. If you had a 0 through 100 weighted rating on all of these aspects—speed, handling, and cost—the fulcrum would be 50, 50, 50.

Note

Most games have more than one fulcrum. In a fantasy RPG, there might be fulcrums for swords, shields, armor, characters, magic spells, and so on.

Creating a Fulcrum

Once you have decided what needs a data fulcrum, it's fairly straightforward to make one. Start with every attribute in the middle of the expected range. For example, if the game you are working on has strength, dexterity, and intelligence attributes for all characters and the expected range is 0–100, the fulcrum character would have the following attribute values:

- **Strength:** 50
- **Dexterity:** 50
- **Intelligence:** 50

But keep in mind that this is just a starting point. It may turn out, for your game, that all 50% attributes are not actually balanced or fun. This is just where you start and not where you finish.

Testing a Fulcrum

Once you have the initial numbers for a fulcrum in place, it's time to test. You should think of the most common situations in which this object will be used and test them. These situations will, of course, vary for different kinds of games, but we can look at an example to illustrate.

For a game that is centered around characters in combat situations, like an RPG or action shooter, then that's the best place to start testing. Put your fulcrum character into a large variety of combat scenarios. If the game is multiplayer, test the fulcrum against itself. Let's say you made an eight-player death-match shooter with one test level. You would need a fulcrum character and a fulcrum gun. You would then use that character and gun eight times—once for each of the players—and play a game session. This session will probably not be the most interesting version of the game, but it should work, and it should also start to show some slightly interesting gameplay. It's likely that it won't even work on the first test. However, it is easy enough to redesign the single fulcrum character and test again.

You need to keep iterating on this fulcrum character until you achieve the goals of having a working game and slightly interesting combat. It's not uncommon for a fulcrum to take a dozen or more iterations to get to a working state. Say that you have a character with these stats:

- **Strength:** 50
- **Dexterity:** 50
- **Intelligence:** 50

During playtesting, you might find that dexterity of 50 is just too low for anyone; characters are constantly missing, and the battle is a slog. You might discover that the actual scale of standard usable range is 70–100, with anything below 70 reserved for special cases like injuries. In this case, you would change the fulcrum character to have dexterity of 85 because that is the halfway point in the 70–100 range. This might or might not be your final number, but it's a good starting guess.

After you adjust the fulcrum, you need to test again. Take note of what has changed and what has stayed the same. It is advisable to change only one attribute at a time so you can more easily see the effects of changing that attribute. You may need to repeat this process many times. Getting the feel of the fulcrum is very important as it is the basis for all other data objects to come. Don't worry if determining the fulcrum takes much more time than you have allotted to balancing an individual object; it is likely to take far longer to get the fulcrum right than it will take to nail down the other objects.

It is also important to test your fulcrum in as many different conditions as possible. For example, you might want to see how the fulcrum does in a one-on-one battle, with three players, with four players, and so on. You should also check different environments, if possible. What's the smallest area? What is the largest? Does the fulcrum work equally well in these areas? What is the most wide-open area you are going to have? What about the tightest area? Try to test the fulcrum in isolation in as many different situations as you can. Doing this testing helps you work out any balance issues not only with that data object but with the game environment. Testing often uncovers limitations in game systems.

Locking a Fulcrum

After you do testing for days, weeks, or even months, a clear picture of what the game is and how the game objects will work should start to emerge. This is rather exhausting, tedious work, but doing it will yield dividends in the end. Once your team feels the fulcrum object is rock solid and you understand the basic mechanics of the game, it's important to lock down the fulcrum, which means making no more changes to it. Unless there

is a catastrophic bug or a major change to the project, the fulcrum needs to be locked before anything else is created.

A locked fulcrum serves a few very important purposes. First and foremost, it is the measuring stick for every other object of its type. So, if you are making a cart-racing game, you will be comparing each new cart with your fulcrum. If it's a fighting game, each new character will be compared to the fulcrum. If it's a weapon, each new weapon will be compared to the weapon fulcrum, and so on.

A locked fulcrum also acts as a reference for other aspects of the game. For example, if you have a game that allows players to mix and match characters with weapons, you can use a fulcrum character to even the playing field for weapons testing. Conversely, you can use a fulcrum weapon to test how each character uses the weapon. Finally, you have the set standard of a fulcrum character armed with a fulcrum weapon; you have a benchmark for the middle.

Finally, you need a locked fulcrum for testing everything around it. With an arena shooter, for example, you can give level designers a rock solid character to play in each of their new level designs. They can be confident that they are building levels that will work with the types of characters being built for the game. You, as system designer, can also be confident that the level design team knows what to expect from the characters you are building and that they will make levels that conform to the needs of the systems. This can be done long before you have all the data done for the game, which means the level design team gets a huge head start. This is also the time to have discussions with the team about expectations. It's better to get everyone on the same page now than to wait until vast arrays of data objects and levels have been created.

Using a Fulcrum for Data Creation

Once the fulcrum object is locked, it's time to start working on all the rest of the data. Let's look again at the arena shooter example. After much tweaking and tuning, say that you have a fulcrum character with the following stats:

- **Strength:** 40
- **Dexterity:** 85
- **Intelligence:** 50

For this design, let's say you have 19 more characters to make. You have already gone through the process described in [Chapter 11, “Attribute Numbers,”](#) and decided what kinds of adjectives to use:

- **Bruiser:** Strongest, very slow, very low intelligence
- **Sniper:** Weak, good dexterity, good intelligence
- **Medic:** Weak, poor dexterity, best intelligence

To start building these characters, you have determined that the bruiser will be your highest priority, and we will start by giving him an attribute score. Here's what you use on the first pass:

- **Bruiser**
 - **Strength:** 40
 - **Dexterity:** 85
 - **Intelligence:** 50

It may seem surprising that the first pass is identical to the fulcrum, but this is the best way to start: Literally copy and paste the same attributes from the fulcrum into every other data object. Now that your bruiser has stats, you can start to test and tweak them. But first, you need to test the bruiser with just the copied numbers. You need to make sure that he behaves exactly like the fulcrum. It's entirely possible that some unanticipated factor might change the object's behavior. The model mesh, animations, a material, some code error, or any number of other factors can cause bugs at this stage. The great thing about using the fulcrum numbers first is that doing so enables you to more quickly isolate the problem because you know the intention of the attribute numbers you have assigned to the character. Once you have tested the bruiser with fulcrum numbers and he checks out, you can start to deviate. These might be the numbers you test next:

Note

At this point, it becomes increasingly important to keep referencing the bruiser to the fulcrum.

▪ Bruiser

- **Strength:** 90
- **Dexterity:** 73
- **Intelligence:** 20

If you are making an eight-player death match, you should use seven fulcrum characters and one bruiser for your test on the first set of deviation numbers. How does the bruiser perform? Test again and make sure the results are consistent. If the bruiser fares on par with the fulcrum, then you have successfully deviated without causing an imbalance. Once this test is done, it would be a great idea to mix it up and do a few more tests, such as two bruisers and six fulcrum characters or four of each or seven bruisers and one fulcrum. It is important to do tests like this to make sure small deviances don't add up and become large imbalances on a larger scale.

When the testing cycle for the bruiser is complete, it's time to make more adjustments to the attributes and test again against only the fulcrum. This can take several iterations, but the process should go much faster than the original tuning of the fulcrum.

Remember that the goal in all of this is a fair fight between the bruiser and the fulcrum. They should have an equal chance of winning. If the bruiser is dominant, then he needs to be adjusted down; conversely, if he is constantly losing, he needs to be adjusted up. But remember that you never adjust the fulcrum once you've locked it.

When the bruiser has numbers that work well against the fulcrum in all the test situations, you can set him aside. Next, you need to repeat the process with the sniper. Importantly, we do not initially test him in comparison to the bruiser. This is where having a fulcrum character really starts to pay off in time savings. Because you know the bruiser is balanced against the

fulcrum, you can make a fair assumption that the sniper will be balanced against the bruiser if you balance the sniper against the fulcrum as well.

When you have finished with the sniper, you can set him aside and work on the medic. Again, you need to test the medic only against the fulcrum, and not against the bruiser or the sniper. You continue this process for all the remaining characters. Your testing and tuning phase turns from an ever-expanding slog into a simple linear test. When you add another character, you don't need to test it against every other character; you only need to test it thoroughly against the fulcrum.

Unavoidable Cross-testing

After completing all 20 characters and testing them against only the fulcrum, you have to do the unavoidable: test them in many combinations. A vast number of combinations are possible in this situation. To find out how many, you can use the spreadsheet function COMBIN that was introduced in [Chapter 12](#). To determine how many combinations of 20 characters you can have in an 8-player game, you use the following formula:

=COMBIN(20,8)

This formula gives you a result of 125,970 possible combinations. It would be impossible to test this many combinations internally at a studio. The good news is that you don't have to test all of them; for the most part, because you tested all the characters against the fulcrum, everything should be balanced from the very first test.

Because you did the extra prep work of making the fulcrum, locking it, and using it to balance all the characters, they will naturally be close to balanced already. Therefore, instead of testing all the characters against each other, you only need to test some combinations, based on human intuition. The system designers and a good QA team should try combinations they think might be problematic. This is a necessary step in finding the final balance. (In [Chapter 17](#), “[Fine-Tuning Balance, Testing, and Problem Solving](#),” we further discuss how to go about finding the right balance in game data.)

Note

On large teams, developers may be able to use automated testing AI or other algorithms in lieu of brute-force testing.

Fulcrum Progression

Up to this point in the chapter, we have been discussing a single slice of game objects, but in more complex games, this might not be enough. If a game has the concept of leveling up or another form of progression among data objects, developing a single fulcrum won't be enough. For example, a player might start out as Level 1 and fight against Level 1 orks, improve to Level 2 and fight against bigger and tougher Level 2 orks, improve to Level 3, and so on. In such a case, the fulcrum character needs to be made for all versions or levels before the rest of the data objects can be built and stratified across the levels.

Which level of fulcrum should be created first? There could be an argument made for starting at the lowest, at the highest, or in the middle. When in doubt, starting in the middle is the safest bet. For example, if your game has objects that go from Level 1 to Level 9, you might first create the fulcrum at Level 5. Once the fulcrum feels good in isolation at Level 5, you can strip away power from the object until it is weak enough to be considered Level 1. After that, you can add power to the fulcrum character until it is at its maximum, Level 9. (For further details on increasing power, see [Chapter 14, “Exponential Growth and Diminishing Returns.”](#))

Once the boundaries and center point have been established, you can test those objects against each other. Is Level 5 better enough than Level 1 that players will be able to feel the progression? Similarly, there should be clearly large amounts of improvements in the fulcrum character at the very top level.

After the fulcrum is created with lowest point, a highest point, and a midpoint, you can create all other objects at the same middle progression, knowing that they will be balanced against the central fulcrum. Once all the levels for the fulcrum are calculated and the central points of all other data objects are established, you can use the same gradient built for the fulcrum

to apply changes to the rest of the data objects. In the end, you will have a series of data objects in their own lines of progression that should be running parallel to each other from lowest to highest, as illustrated in [Figure 13.8](#).

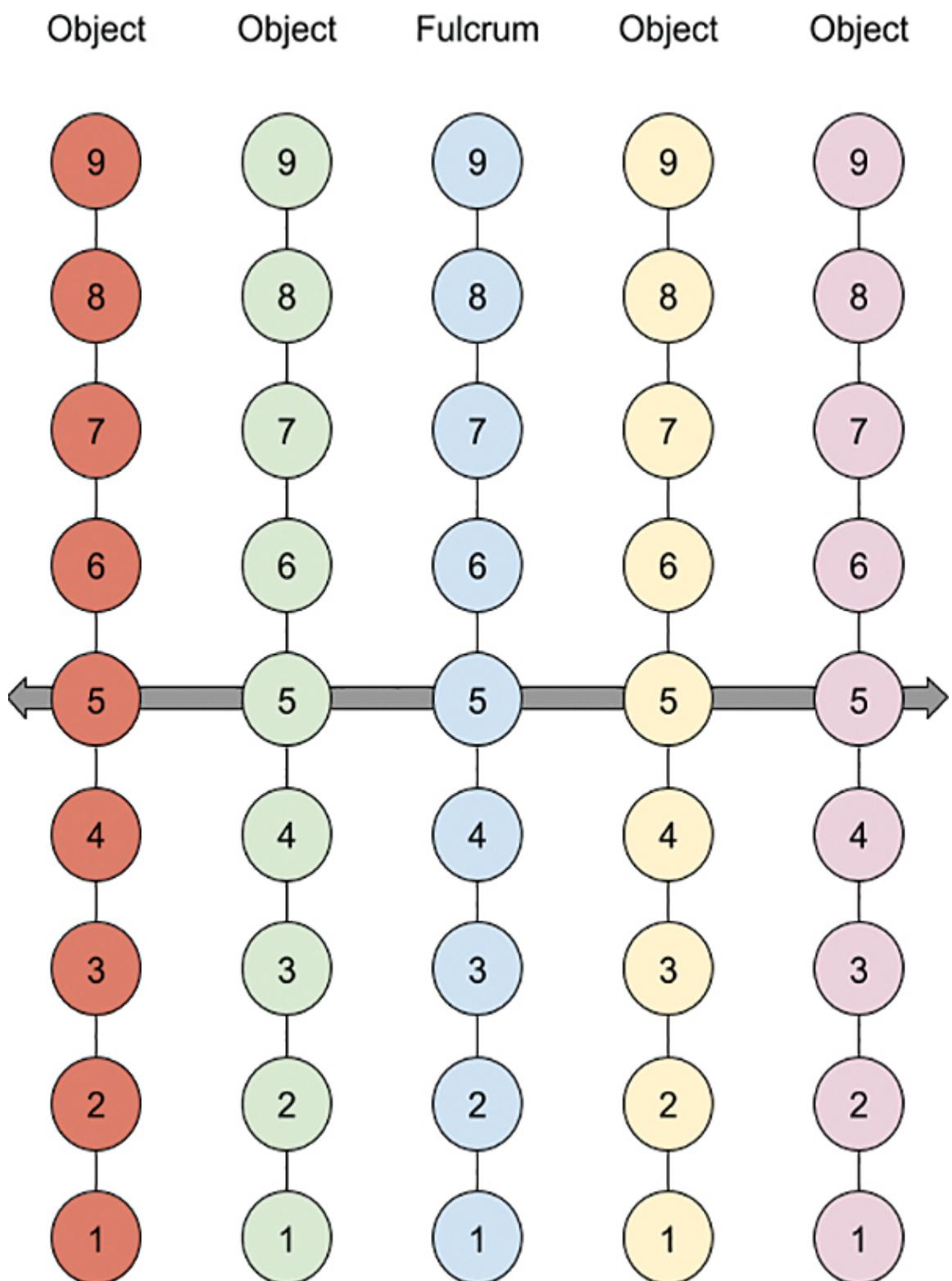


Figure 13.8 Fulcrum in progression

As you can see in [Figure 13.8](#), the center of the data world is the fulcrum object that runs up the middle. At the central point of progression—in this case Level 5—all the other data objects are created and balanced. Then, each additional data object has its own line of progression that runs in lock step with the fulcrum as it increases in level.

Hierarchical Design

There is a common problem that occurs not only in games but in any complex system that contains multiple variables that can all be tuned. Which variable do you tune? For example, if you are working on a game where the player character is doing battle with a monster, and you decide that the monster is too difficult, how do you make the monster easier to battle? Assuming standard RPG combat attributes, you could make the monster do less damage, attack less frequently, or start with fewer hit points. Conversely, you could make the player character do more damage, start with more hit points, or hit more frequently. You could also use any number of combinations of any of these adjustments. But which one should you change? Which ones should be left the same? As discussed earlier, the data fulcrum should be locked by the time you get to balancing characters, so that should not change. What else governs which value to change? This answer will be slightly different for every game, but there is a method you can use for any game to make the decision easier: hierarchical design.

The basic method of hierarchical design is very easy: Write a linear list of the elements in the game from most important at the top to least important at the bottom. Then, if two features conflict with each other, check where each lies on the list. Change the feature that is the lowest on the list. When in doubt, start with the most globally affecting features at the top and work down to the most local features at the bottom. As a general rule, you should tune local features to fit with the global design—and not the other way around.

Starting the Hierarchy

At the very beginning of the game-making process, you can begin to form your hierarchy. Start with very broad concepts like the following:

- Core game vision
- Game engine
- Console platform
- Game controls
 - Game camera
 - Control input buttons
- Game user interface
- Player character
 - Attributes
 - Mesh
 - Animations
- Level design constraints
 - Fulcrum level with max ceiling height, door width, and so on
- Fulcrum enemy character
 - All other enemy characters
- NPC fulcrum character
 - All other NPC characters
- Power-ups
- Individual levels
 - Geometry flow
 - Interactive setups
 - Macro system rewards
- Sound
 - Music
 - SFX

With this basic example of a hierarchy, you already have the answer for part of the earlier question: The player character should not change, but the monster should. Imagine changing the player character's attributes to make this one monster fight better. What might that do to the rest of the game? It is highly likely that it would throw off the balance of combat in many or all other monster fights. That would require other designers to go back and tune them again, and they might even want to change the PC again. This would cause an endless cycle of rebalancing the PC to try to fit a multitude of situations. On the other hand, if you change the monster, all the other fights between the PC and other monsters would remain the same. The changes would remain as local as possible, which is a key to keeping rework time accurate and fast.

To determine which attributes to change on the monster, you could use the fulcrum monster as a guide because it is higher in rank than this individual monster. After that, there could be a sublist of attributes that are in order as well, or an individual designer could be given leeway to make a judgment call at this lower level.

Advantages of Hierarchical Design

There are several reasons for a team or even an individual developer to build a game hierarchy:

- It helps solve disputes. Being able to point at a definitive list makes it easier to make a decision and stick with it.
- It facilitates communication. If the whole team has access to the hierarchy and knows what is most important, team members can better work together, without stepping on each other's toes.
- It makes priorities in production clearer. In the hierarchy example you just saw, the game camera is very high on the list. This means that work on the game camera should start and even finish much earlier in the project than items lower on the list, such as level geometry. These two aspects in particular are often at odds with each other during development, so locking down the higher of the two early on will translate into less rework time.

- It can be a living document. Notice that the example above is rather vague, only listing categories of game features. As the game is developed, the hierarchy should grow to match. Each individual monster or power-up could be listed, if needed.
- It's flexible in scope. While you could list every feature and data object in the game, getting this granular is usually not needed. Working through the project, starting with a high-level overview and adding more detail when needed, the team can organically find the level of granularity that works best for the project. On a very large project with hundreds of developers and thousands of data objects, the hierarchy might need to get very big and detailed. Conversely, on a solo project or small team, the high-level example shown earlier in this section might be enough for the entire project.

Once the initial hierarchy is created, it should be discussed within the team. Everyone needs to know where the features they work on lie in the list. There may be some friction among developers about the “importance” of their features. It’s better to have these conversations early than let them go unspoken until a real problem arises.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the hierarchical design and range balancing:

- Create a spreadsheet with either characters from your own game or a favorite game of yours. Bring in their attribute values and calculate the min and max values. Then use these numbers for a range and do range balancing on the characters.
- Create a new fulcrum character for either a game you are working on or an existing game. Look at all the characters that exist and try to calculate what the exact center values should be for each attribute.
- Create a game hierarchy for a project you are working on. Start with the broadest strokes first and then start adding specific details about your project and see how you need to adjust the rankings as the project

develops. It can be very eye opening to see your own project laid out this way.

Chapter 14

Exponential Growth and Diminishing Returns

In an exponential growth system, a linear set of steps calls for an ever-increasing amount of input or output. A good real-world example of this is in the speed of cars. In general, a faster car costs more money. But the speed and cost of cars almost never increase at the same rate. Instead, you must pay ever-increasing amounts of money to get a car that is only slightly faster. So for each mile per hour faster you want to go, the speed gets increasingly more expensive. This type of increase is called *exponential growth*. On the flip side, the top speed of the car you are paying for gets smaller per dollar. Such a decrease in the marginal (incremental) output of a production process as the amount of a single factor of production is incrementally increased while the amount of all other factors of production stay constant. This is called *diminishing returns*. These two concepts work together.

Simple games—such as chess, backgammon, and checkers—don't need to use an exponential or diminishing returns system. It's not until you start to make campaign-style mechanics that you need growth systems at all. In game systems, exponential growth is consuming ever-increasing resources and giving back only linear incremental change. Diminishing returns is the flip side of this same coin. A simple and common example is earning experience points to level up a character. In an exponential growth system, it takes ever more experience points to increase the character's level by one more level.

Game system designers make exponential growth systems for several reasons:

- They can “hook” the player into the game by providing early reward cycles.
- They can entice the player to play for longer periods of time and form a deeper relationship with the game in the later phases of the game.
- They allow developers to introduce new, simple mechanics quickly to expand play possibilities.
- They make it possible to pace out later rewards at intervals that fit the development and mechanical goals.

Anything that levels up or gets more difficult over time or space can make use of exponential growth charts. The following are some examples:

- Difficulty of progressive levels in a coin-operated arcade game
- Quality of weapons compared to cost
- Speed of a vehicle compared to cost
- Experience needed to level up
- Time played to earn an achievement
- Number of enemies defeated to complete a quest line
- Number of daily pushups done over the course of exercise training
- Number of material items collected to make better-crafted equipment

Linear Growth

Before trying to understand exponential growth it’s good to have a point of comparison in linear growth. *Linear growth* simply means that as a number of iterations gets larger, so does a total amount. For example, if after each turn in a game you add 2 points to a total, then after 10 turns, you would have 20 points, and after 100 turns, you would have 200 points. The chart in [Figure 14.1](#) illustrates this type of growth.

Linear Growth

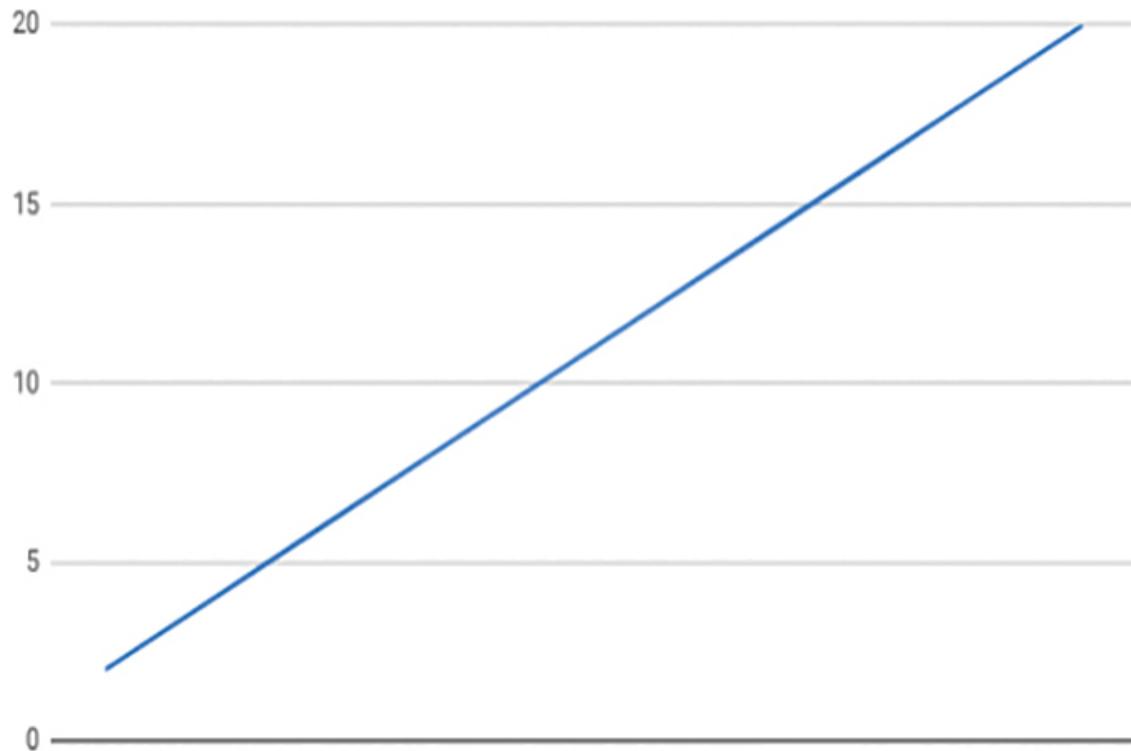


Figure 14.1 Linear growth chart

You use linear growth in games where you want a simple, steady progression. Game designers turn to exponential growth and diminishing returns in more complicated games. These two concepts are flip sides of the same coin, and game designers constantly use both of them.

Exponential Growth

As mentioned earlier in this chapter, *exponential growth* is growth whose rate becomes ever more rapid in proportion to the growing total number or size. [Figure 14.2](#) shows a graph with a curve illustrating exponential growth. There are an infinite number of ways to create exponential growth in a game, and each has strengths and weaknesses. Because this is an introductory book, the formula we discuss here is a great formula to start with; throughout the book, we refer to this formula as the *basic exponential growth formula*. By starting with a simple formula and then testing it with

your game, you will likely find ways to tweak and tune it to exactly fit your needs.

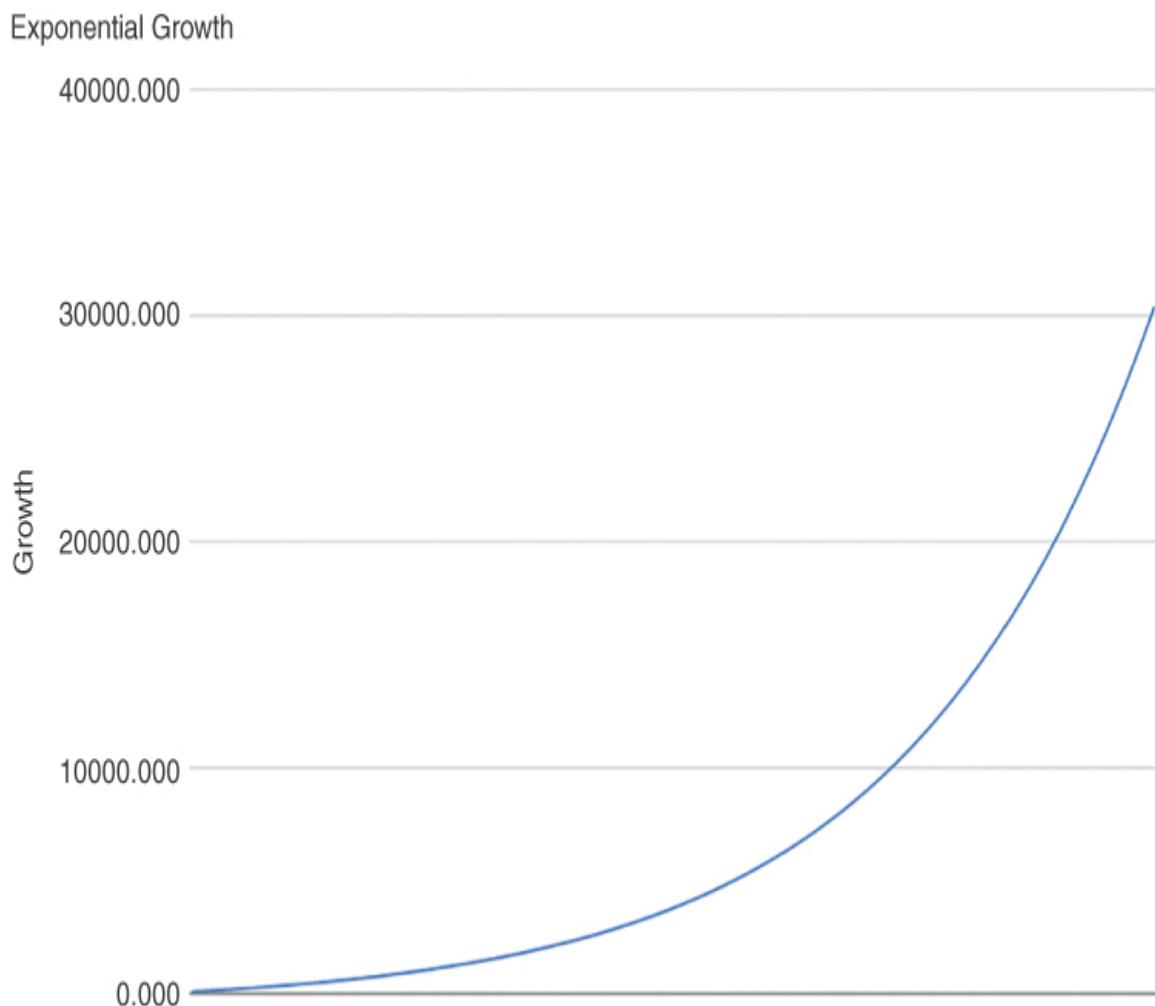


Figure 14.2 Basic exponential growth chart

Note

To get the most out of this chapter, open a spreadsheet and work through all the examples as you read.

Parts of the Basic Exponential Growth Formula

There are only a few adjustable parts of the basic exponential growth formula, but making even slight changes in them can create a vast multitude of variations of growth. These are the adjustable parts:

- Number of iterations
- Initial starting value
- Iteration additive value
- Iteration multiplier value

With just these four variables and a single formula, you can create a curve in a spreadsheet. To start, open up a blank spreadsheet and add some labels for the needed variables, as shown in [Figure 14.3](#).

The screenshot shows a Microsoft Excel spreadsheet with a grid of cells. The columns are labeled A through G at the top, and the rows are numbered 1 through 4 on the left. Cell A1 contains the value '1'. Cell B1 is highlighted in black and contains the label 'Itt'. Cells E1, F1, and G1 are also highlighted in black and contain the labels 'Init', 'Add', and 'Mult' respectively. The other cells in the grid are empty.

	A	B	C	D	E	F	G
1	1	Itt			Init		
2					Add		
3					Mult		
4							

Figure 14.3 Basic input variables

In this example, the variable names have been shortened to fit on the page better. Itt, which stands for “number of iterations,” needs to be a column header because it is going to get a lot of entries. Init (initial starting value), Add (iteration additive value), and Mult (iteration multiplier value) have been color-coded as sublabels and have individual variable cells next to them. The blank columns in this spread sheet leave room for you to add computational columns later.

Note

In a game example, you could change “iterations” to “player” level to find the experience required to level up.

The first decision that needs to be made to fill out the spreadsheet is how many iterations, or “steps,” are needed. This could be how many levels the player can go through until they max out. Or it could be the cost of different swords that a player can buy, or power levels of spaceships, or anything else that should have exponential growth. The basic exponential growth formula is purposely generic enough that it can fit in a wide multitude of applications. In the example in this chapter, you are going to use 100 iterations to see the growth over a long curve. In your own game, you might use a curve that is much smaller or even larger.

To add iterations, start by typing 1 in cell A2 and then formfill down (as discussed in [Chapter 5, “Spreadsheet Basics”](#)) until you have a list of numbers that matches your needs. Next, you need some placeholder numbers for the remaining variables (Init, Add, and Mult); you can input 1 for each of them for now. Your spreadsheet should now look as shown in [Figure 14.4](#).

	A	B	C	D	E	F	G
1	Init				Init		1
2	1				Add		1
3	2				Mult		1
4	3						
5	4						
6	5						
7	6						
8	7						
9	8						
10	9						
11	10						
12	11						
13	12						

Figure 14.4 Creating iterations

Next, you need to add a calculated value for the result of each step. You can start this in cell B2. The very first step will be identical to the value of Init, so you should set cell B2 to reference cell G1 in this case. You should also add the title Value to column B because it will show the calculated value for each step. Then, in cell B3, you can start entering the formula that will govern the entire exponential curve, as described next.

Building Blocks of the Exponential Growth Formula

In each step of the basic exponential growth formula, the previous step is multiplied by the iteration multiplier value, and then the iteration additive value is added to the result:

(Previous step \times Iteration multiplier value) + Iteration additive value =
New step

This formula is repeated for the number of iterations desired.

[Figure 14.5](#) shows a spreadsheet with this formula applied for the first time.

A	B	C	D	E	F	G
1	Itt Value				Init	1
2	1 =G1				Add	1
3	2 =(B2*\$G\$3)+\$G\$2				Mult	1
4	3					

Figure 14.5 Applying the basic exponential growth formula

Notice in [Figure 14.5](#) that the reference to B2 is relative, which means it will be updated with each iteration as it is formfilled down the list. Also notice that the references to the iteration additive value and the iteration multiplier value are absolute, as denoted by the \$ in the cell reference. Because the location of these values never changes, the references need to be locked and unchanging as well. After you enter this formula as shown in [Figure 14.5](#), you can formfill it down to create values for the iterations. With the placeholder numbers you already have in the spreadsheet, you will get a very uninspiring list of numbers from the formula, as shown in [Figure 14.6](#).

	<i>fx</i>	$=(B3 * \$G\$3) + \$G\2					
	A	B	C	D	E	F	G
1	Itt	Value				Init	1
2		1	1			Add	1
3		2	2			Mult	1
4		3	3				
5		4	4				
6		5	5				
7		6	6				
8		7	7				
9		8	8				
10		9	9				
11		10	10				
12		11	11				
13		12	12				

Figure 14.6 Step calculations

This is not quite exponential growth yet because the multiplier is at value 1. Because any number multiplied by 1 is itself, using a multiplier of 1 actually gives you a linear growth chart and not an exponential one. To see this visually, select column B and insert a line chart. You should see a chart that shows basic linear growth, as in [Figure 14.7](#).

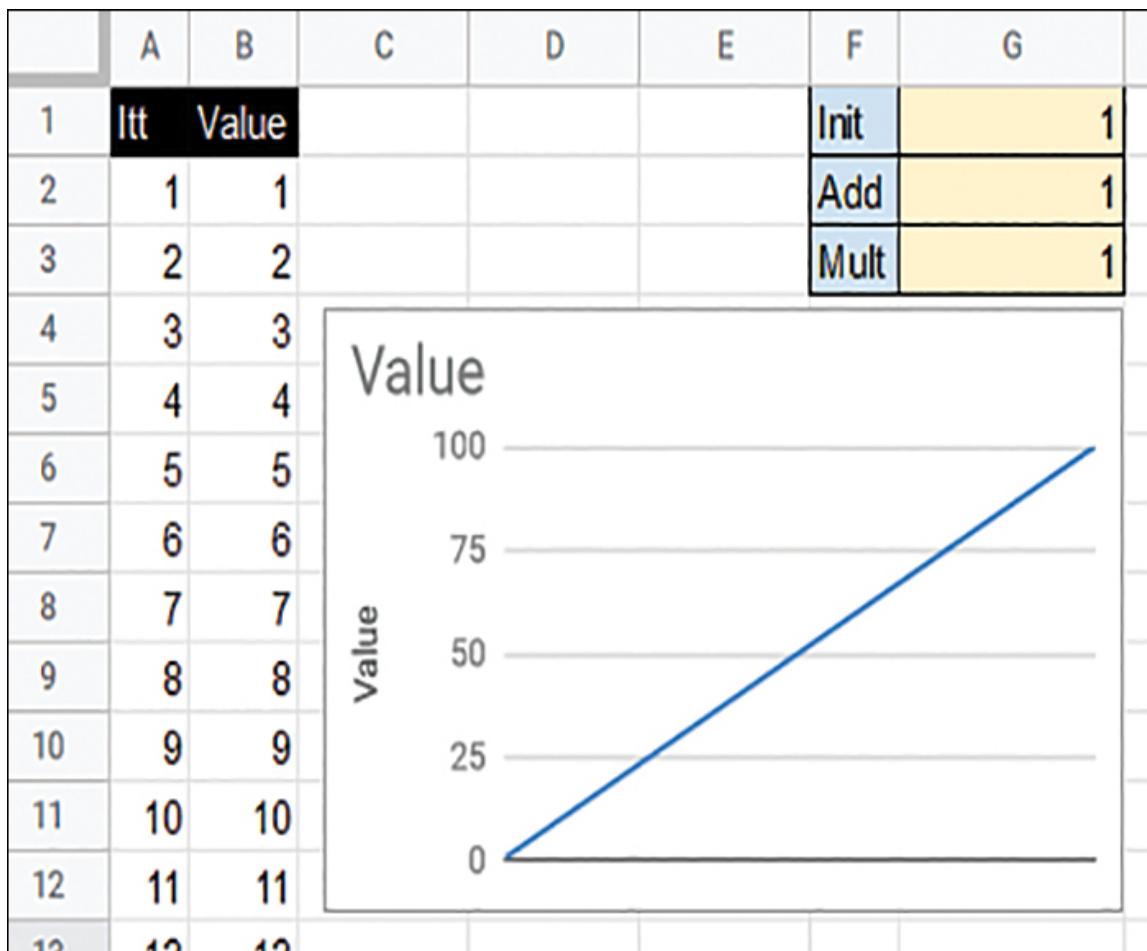


Figure 14.7 A chart showing linear growth

This is not what we ultimately want in this example, but all the needed components are now in place. The key to exponential growth is the multiplier. Without a multiplier, there is no exponential growth. But what number should you enter for the multiplier? Keep in mind that the player will likely never see this variable, so you don't have to worry about it being an integer or anything that is accessible at all. It may be surprising, but over the course of a long iterative run like this, very small multipliers are needed. You can test this yourself by typing in values for the Mult variable (in cell G3) and noting what happens to the chart. With even a modest integer, like 3, the graph shoots off the scale in the later iterations, with numbers so large they can be displayed properly only with engineering notation (see [Figure 14.8](#)).

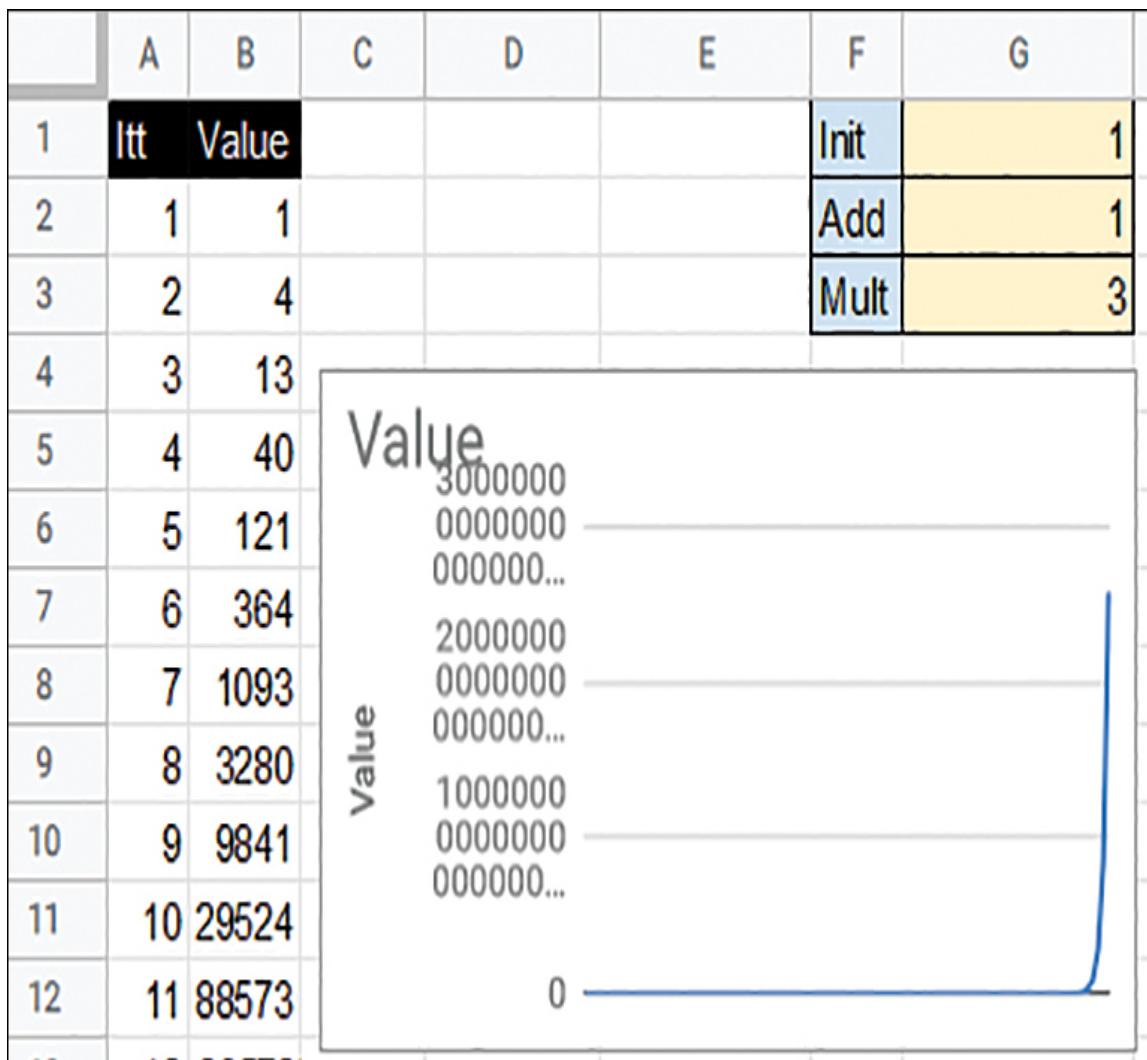


Figure 14.8 Broken growth with 3 as the multiplier

You can think of the multiplier as a percentage—and a small one at that. To get started, 2% growth is a nice middle ground that is not very steep but clearly shows growth over iterations. To write 2% as the Mult value, you would use 1.02; this multiplier yields the graph shown in [Figure 14.9](#).

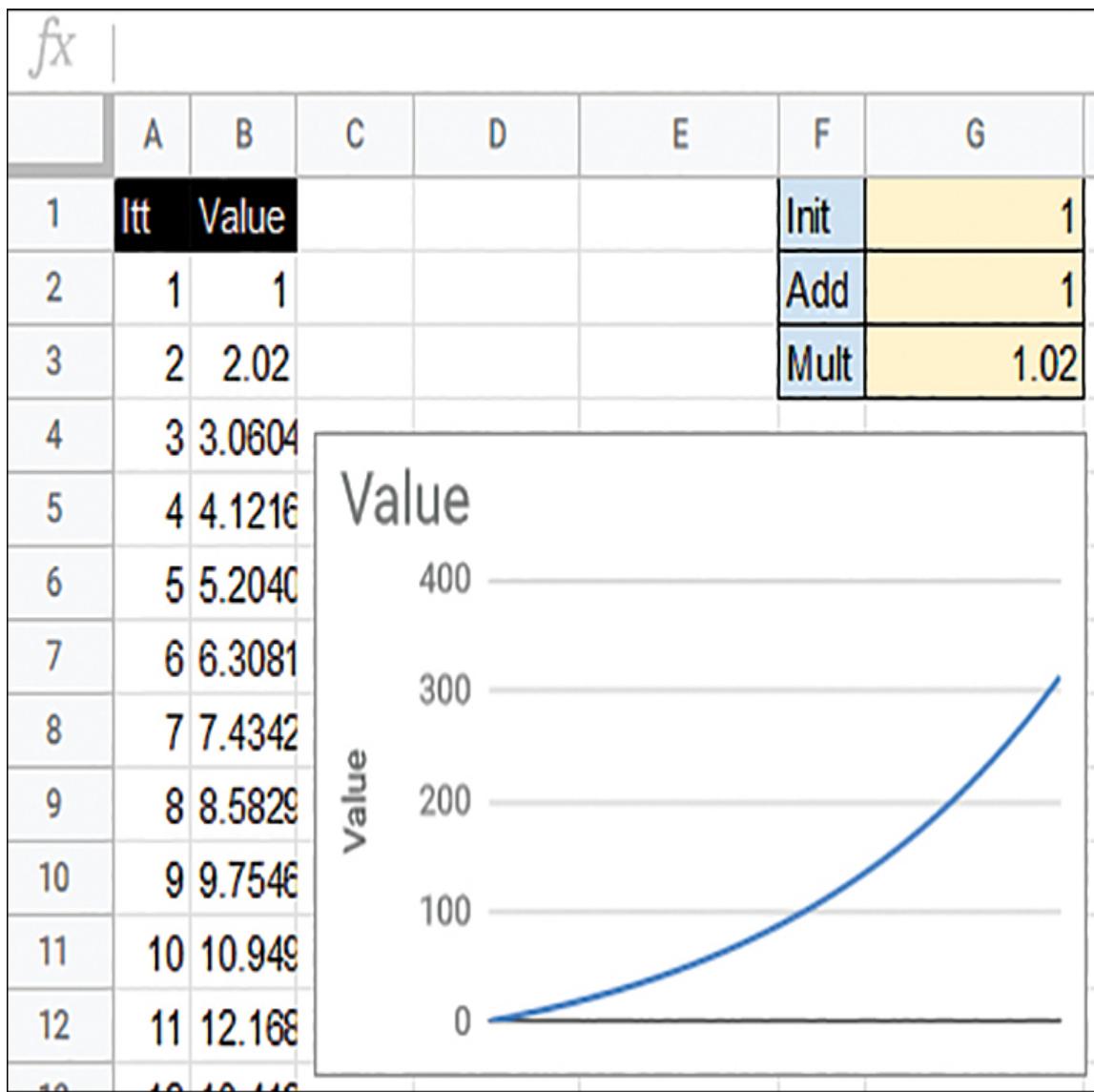


Figure 14.9 Smooth growth with 1.02 as the multiplier

At this point, the graph is showing exponential growth; technically, you could stop here. However, there are some practical problems with this growth. To see the first problem, observe the values in column B and imagine them in the game context of experience points needed to reach the next level. The player would start at Level 1 and be told to gain 1 XP (experience point) to level up. At Level 2, the player would be told to gain 2.02 XP. Here is the first problem: You never want to display decimals to the player. One way to solve the decimal issue is to create a derived value (or “display” value), where the decimal value is rounded using a function to

make it more digestible for the player. [Figure 14.10](#) shows a column added for display value (column C).

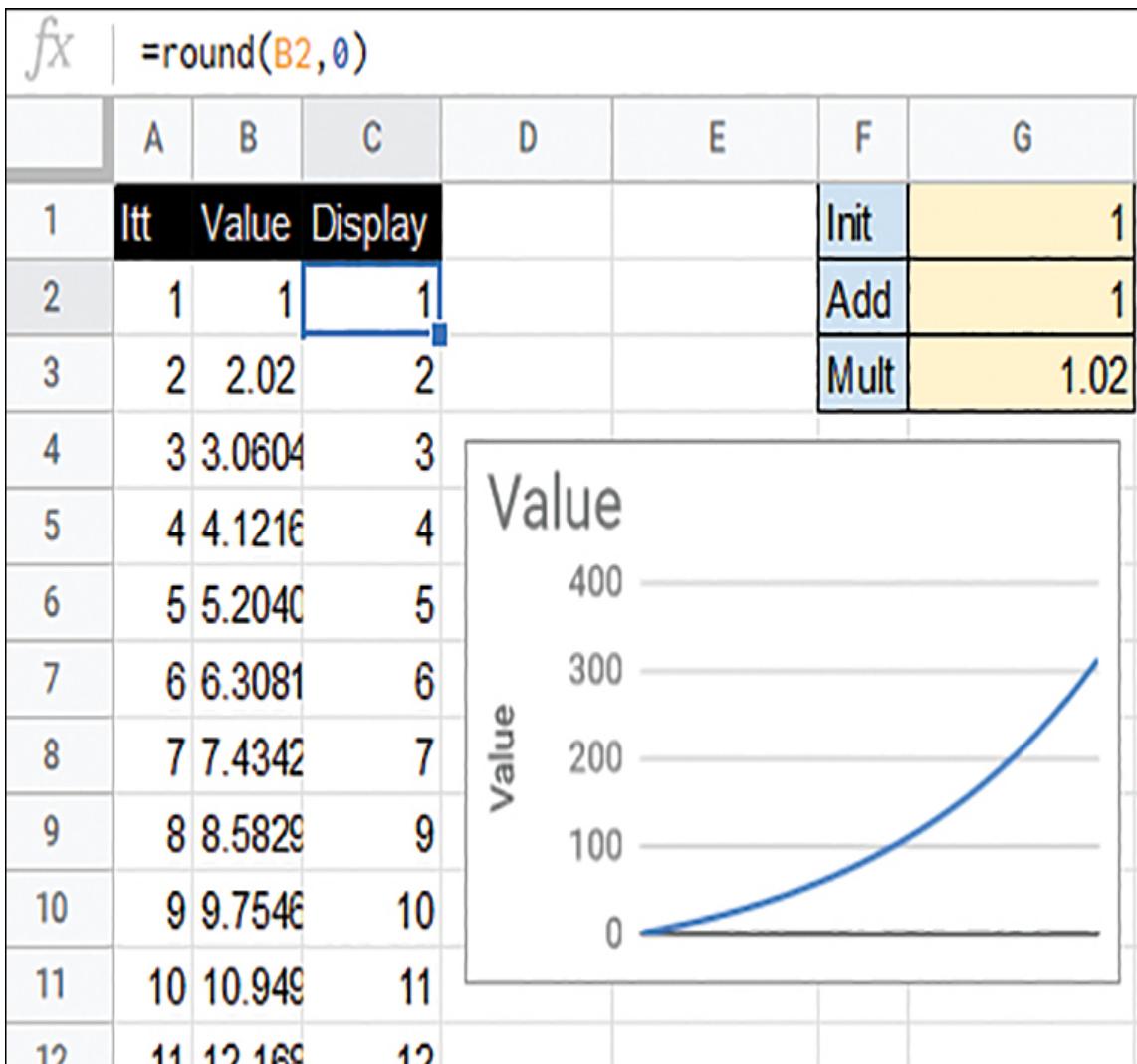


Figure 14.10 Adding a column for display values

The Display column now presents numerals that players can work with more easily, but it exposes another problem with the list. At the early levels, the iterations provide nearly linear growth—which can lead to player confusion and, even worse, present some difficult math problems for systems designers.

For example, in a racing game, to level up a race car, you might have the number of XP needed listed in the Display column. And you might allow the player to earn XP by doing a very simple race to gain 1 XP or a tougher, higher-failure-rate race to gain 2 or more XP. With the values as listed in [Figure 14.10](#), the player would gain levels very quickly and would not be encouraged to ever take on the tougher, higher-failure-rate races. It would be easy to “grind” the easy race repeatedly because the increase per level is so small.

While it would be possible to fix this issue by increasing the multiplier, doing so would lead back to the same problem of very large numbers at the end of the list. Instead, you want to have steady, noticeable growth at the beginning and larger, exponential growth at the end. To accomplish this, you can change the Add value. Because it is additive, it does not affect the shape of the growth curve, but it changes the increments at the start of the curve. For example, changing the Add value to 10 results in the graph shown in [Figure 14.11](#).

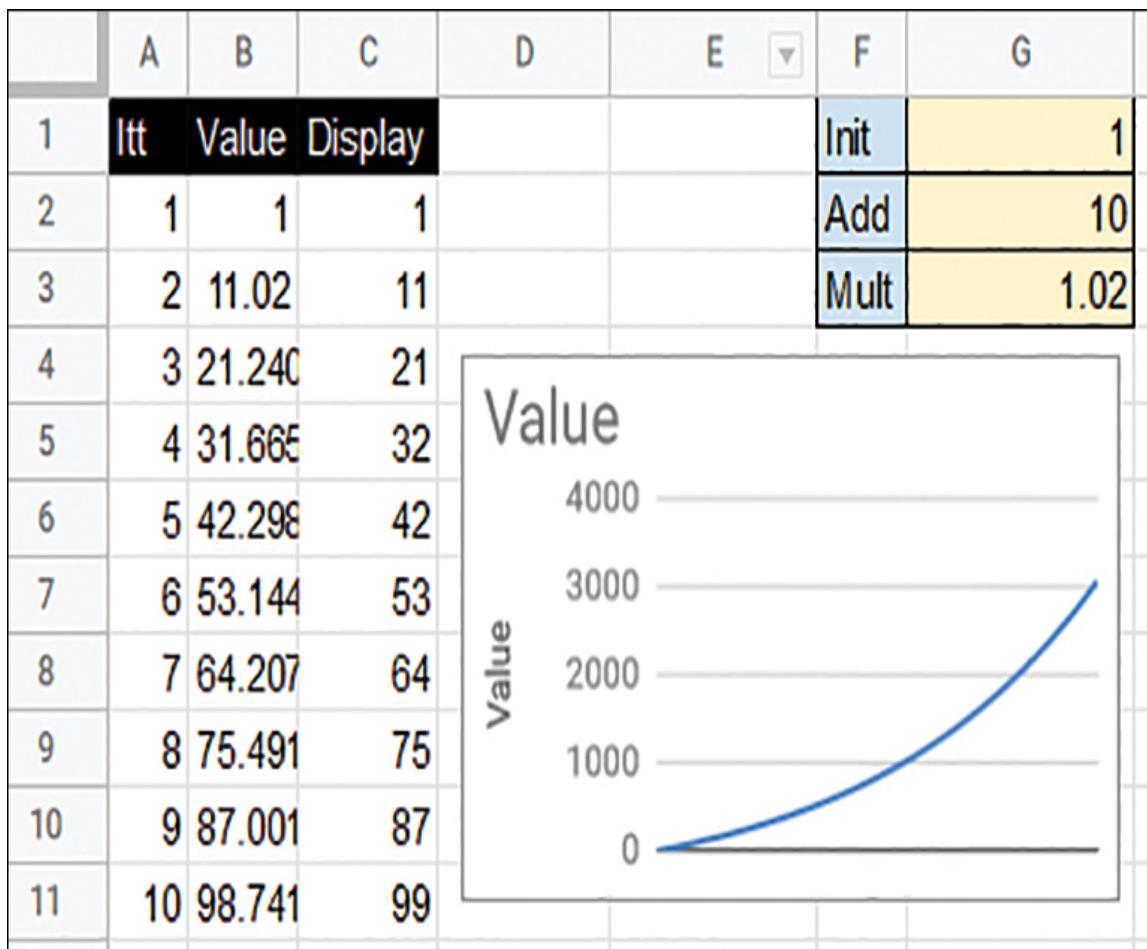


Figure 14.11 Rounded numbers

At this point, the curve is still showing a comfortable growth rate, but the early levels show distinct growth for each iteration. The growth is enough that the players can clearly notice it intuitively, and it will be enough to encourage players to perform tasks to gain more of the resources needed (XP in this example). There is one more problem hidden in here. To see it better, scroll down to the bottom of the iterations and look at the kind of growth shown near iteration 100 (see [Figure 14.12](#)); compare this growth to the growth shown in the first 10 levels.

	A	B	C
1	Itt	Value	Display
91	90	2419.127153	2419
92	91	2477.509696	2478
93	92	2537.05989	2537
94	93	2597.801088	2598
95	94	2659.75711	2660
96	95	2722.952252	2723
97	96	2787.411297	2787
98	97	2853.159523	2853
99	98	2920.222713	2920
100	99	2988.627168	2989
101	100	3058.399711	3058
102			

Figure 14.12 Growth near the end of the iterations

If you are unable to spot the problem, that's fine: It may not be obvious to you unless you see it as a curve or put it into a game and test it. But you can expose the issue here in the spreadsheet by doing one more check: You can determine the growth of each iteration as a percentage of the last iteration. That is, you can determine the size of jump in needed XP for each level. Is it even across the entire run, or does it spike?

To visualize this, you can take the previous value of each iteration and divide it by the new iteration value. This will show, as a percentage, how

much of a jump there is in the value needed to level up. Compare the top of the iterations list (shown in [Figure 14.13](#)) to the bottom of the list (shown in [Figure 14.14](#)).

	A	B	C	D
1	Itt	Value	Display	
2	1	1	1	1
3	2	11.02	11	1102.00%
4	3	21.2404	21	192.74%
5	4	31.665208	32	149.08%
6	5	42.29851216	42	133.58%
7	6	53.1444824	53	125.64%
8	7	64.20737205	64	120.82%
9	8	75.49151949	75	117.57%
10	9	87.00134988	87	115.25%

Figure 14.13 The top of the iterations list

	A	B	C	D
1	Itt	Value	Display	
91	90	2419.127153	2419	102.42%
92	91	2477.509696	2478	102.41%
93	92	2537.05989	2537	102.40%
94	93	2597.801088	2598	102.39%
95	94	2659.75711	2660	102.38%
96	95	2722.952252	2723	102.38%
97	96	2787.411297	2787	102.37%
98	97	2853.159523	2853	102.36%
99	98	2920.222713	2920	102.35%
100	99	2988.627168	2989	102.34%
101	100	3058.399711	3058	102.33%

Figure 14.14 The bottom of the iterations list

As this check shows, to go from Level 2 to Level 3, there is 1102% increase in needed XP! This is a huge increase and will likely cause problems in assigning XP values because so much is needed to move up in these first levels. Compare this with the values at the end of the list, where the final iteration is a 102.33% increase. Notice that this increase very nearly matches the growth multiplier, which means the additive value is no longer having a significant effect.

To fix this issue, you need the amount of added growth per iteration to be a smaller percentage of the initial value. Because the initial value is still

sitting at the default 1, and the additive is at the relatively larger amount 10, the percentage increase is dramatically affected. Because you already changed the additive to get it where you want it, the final tweak to make is to adjust the initial value so that it is large enough that the additive value does not create the dramatic percentage increase at the start of the curve. For this example, you can plug in 100 for the Init value and see what happens (see [Figure 14.15](#)).

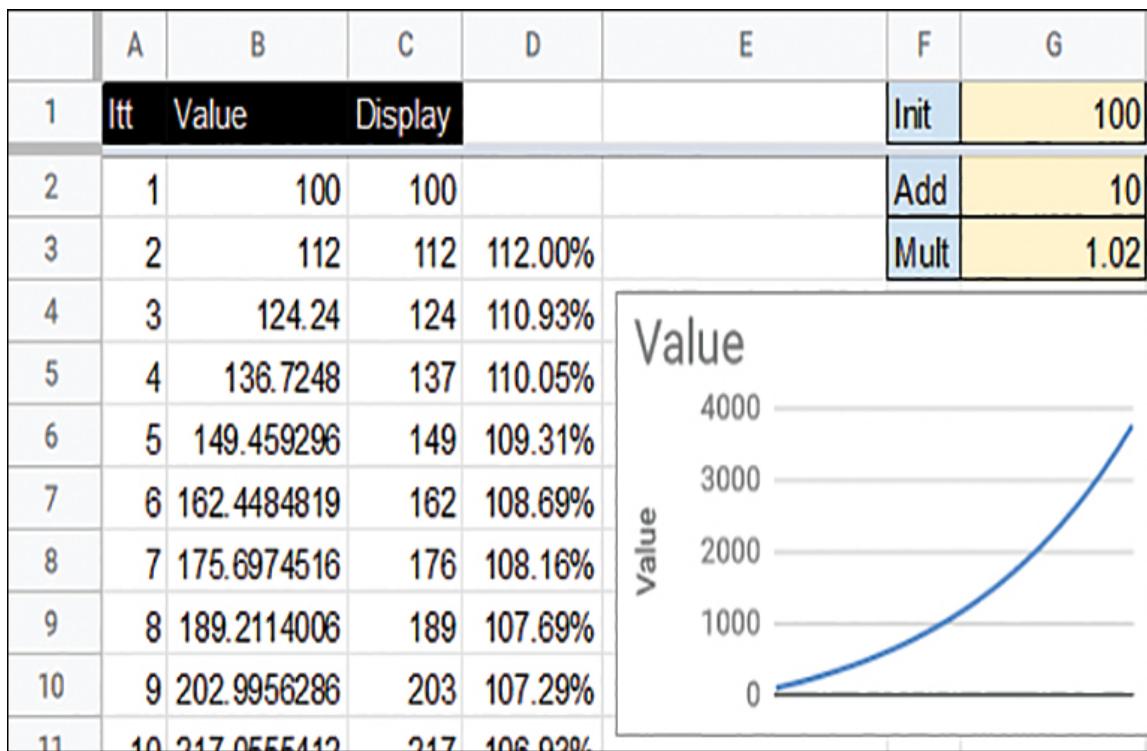


Figure 14.15 Basic input values for each of the variables in the growth formula

The percentage increase has now dropped to 112% at the most and is 102.27% at the last iteration. This might seem like it is still a bit large, but look at the display numbers. The first iterations go from 100 to 112 to 124. A player would not notice a large percentage increase in the number because the actual number is relatively small, at just 12 more points for the second step. This last change has now solved the final common problem of basic exponential growth curves. With these numbers, you have a solid start to work from for almost any exponential growth curve imaginable.

Tweaking the Basic Exponential Growth Formula

You could easily branch off from what you have learned to this point by tweaking these four variables to get a wide array of effects. The best way to learn how to do this is to play with the values in a copy of this spreadsheet. Leave up all of the values as they are here and tweak the iterations, initial value, additive, and multiplier. Also take some time to go further than common sense would warrant. Put in very large numbers for the variables and see what happens. Put in decimals. Then go even further and put in negative numbers, formulas, exponential powers, and anything else you can think of. By interactively playing with the variables and observing the results, you can quickly learn how to get effects you want and avoid those you don't.

A Note on Iterations

In the example in this chapter, you did not change the number of iterations, but this variable is important. There is a basic rule you should keep in mind for the number of iterations: The more iterations in a chart, the smaller the multiplier must be. To illustrate this, [Figure 14.16](#) shows exactly the same list of iterations as in [Figure 14.15](#), but the chart includes only the first 10 iterations.

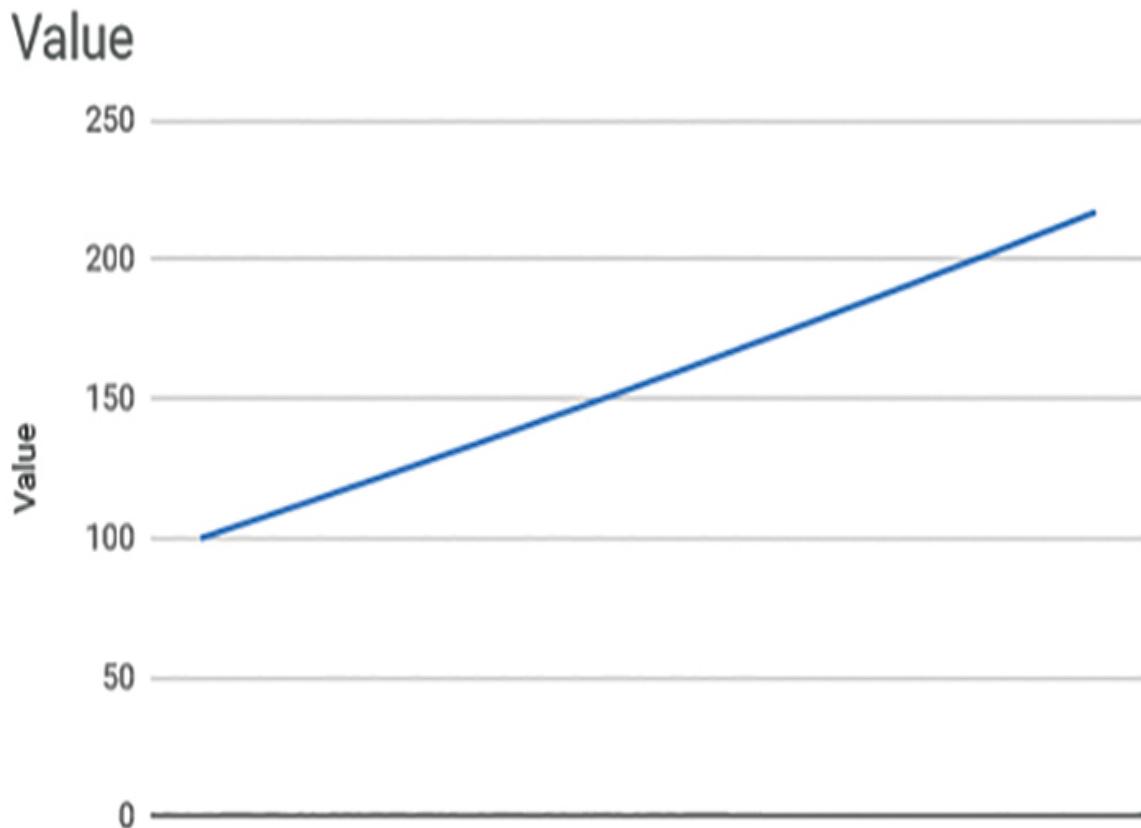


Figure 14.16 Using short iterations

Notice that this curve looks almost completely linear. The 2% growth does not have enough iterations to start to take significant effect over the course of the chart. As you can see, it is important to know the number of iterations needed for total growth before building a chart like this. Fortunately, it's also easy to change the variables quickly to account for any change of iterations.

Exponential Charts and Game Hierarchy

It may be fast and easy to completely change the character of a game by adjusting an exponential growth chart, but doing so also has dramatic ripple effects for the rest of the game. By changing a single variable in the chart, it is quite possible to completely change the balance of every level and every encounter in the game. Because of this, it is highly recommended to place each needed growth chart high in the game design hierarchy (as discussed

in [Chapter 13, “Range Balancing, Data Fulcrums, and Hierarchical Design”](#)) and to lock them down early in the game development process.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the Exponential growth charts in all of their many variations:

- Create a spreadsheet and make the exponential growth chart shown in [Figure 14.15](#). Be sure to use the chart functionality to visualize your data. Start by making 200 iterations and using 100 as the Init value, 1.02 as the Mult value, and 10 as the Add value. This chart will provide a good starting point that you can use to start experimenting.
- After you create the chart described in the preceding exercise, make a copy of the sheet in the same workbook and label the copy “Play.” On the Play , change the input values until you break the chart. (Really, that’s the goal here.) Try putting in very large or very small numbers. Put in negative numbers, or decimals, or even formulas and functions. Keep trying the most bizarre and creative things you can think of and observe what happens to your chart. Can you get it to make a cone? How about a solid bar? Can you get it to look somewhat like a chart of a heartbeat on a medical device? Only by pushing systems beyond their intended bounds can you find where the true bounds are. Note that even when you are purposefully breaking the data, you are not actually doing any damage. Because you made a copy and are not working on the original, your data and structure are completely safe. This is one of the great benefits of working in the digital world.
- When you are comfortable breaking the data, make another copy of your original sheet and see if you can come up with a few varieties of charts that you might like. How gentle can you make the curve before it is functionally linear? How steep can you make it before it breaks? Again, getting comfortable with doing these extreme exercises will help you know what your boundaries are.

Chapter 15

Analyzing Game Data

Up to this point in the book, we have been focusing on making data objects one at a time and then evaluating them individually against other data objects. The next step in understanding a game as a whole is to evaluate all the objects together, whether it's a small set of 10 objects or tens of thousands of objects. You can do this for data as it is created or with existing game data that is fully complete—or anywhere in between.

There are many ways of viewing and analyzing game data, but this chapter focuses on two methods to provide an introduction. The first method is overview analysis, which involves clues and characteristics of the data setup. The second method is comparison analysis, which involves comparing two different data objects and analyzing them in more depth.

Overview Analysis

Overview analysis involves looking from a high level at trends in data to get a larger picture of what is happening. A standard way of getting a high-level view is to find an average of values. However, on its own, an average does not provide much information—and it can actually provide misinformation. Let's look at a simple example of how averages can go wrong. What is the average of the following list?

10

10

10

10

10

10
10
10
10
999,910

The average is 100,000.00. But does 100,000 actually give insights into what is going on in this list? No, it doesn't. If you look at the list, you intuitively understand that most numbers in the list are low numbers (10), but a single number is huge and throws off the average in the data as an outlier. In the end, 100,000 is too small to represent the outlier and too large to represent most of the data points, so it does not give much true insight into how this data works. In order to counterbalance this problem, you can find both the average and the median and then compare those two values. In this example, the median is 10. So this is what we know about the list:

Average: 100,000

Median: 10

By looking at these two numbers, you know that something strange is going on in the data. In a uniform and even distribution of numbers, the average and median are fairly close together, but in this example, they are different orders of magnitude. When analyzing data, this difference would warrant a deeper dive into the data to find out what is going on.

[Table 15.1](#) shows a scenario that you might encounter in making a game: a list of fantasy creatures, each of which has a set of attributes.

Table 15.1 Creature data

Character Type	STR	DEX	HP	AV	DG	MV
Goblin	3	4	5	1	3	5
Zombie	4	2	6	0	0	2
Dark wolf	4	4	7	2	3	6

Character Type	STR	DEX	HP	AV	DG	MV
Snapping turtle	6	4	10	10	2	1
Barbarian	7	3	13	2	4	5
Dwarf	6	4	13	3	2	4
Gnoll	3	7	8	4	6	6
Halfling	3	7	6	1	8	5
Protoman	6	4	9	3	3	5
Kobold	3	8	5	3	5	6
Legionnaire	4	6	10	1	5	5
Lizardman	3	7	7	2	4	7
Ogre	8	3	17	5	1	3
Ork	6	4	10	5	1	5
Rat man	2	8	8	2	8	8
Wood elf	3	8	8	1	6	6
Hellhound	5	4	12	3	3	6
Werewolf	7	5	7	1	1	5
Blob	7	3	20	7	2	2
Giant ant	6	5	10	6	2	5
Warg	7	4	8	3	3	6
Clay golem	7	5	12	6	3	4
Swamp man	5	5	25	2	2	4
Mummy	10	4	20	6	2	2
Giant scarab	8	5	15	9	2	4
Razorback	6	6	15	6	3	5
Serpent	6	6	15	6	6	4
Giant scorpion	6	5	15	9	4	5
Black bear	7	6	15	6	4	6

Character Type	STR	DEX	HP	AV	DG	MV
Harpy	6	8	15	6	5	7
Gargoyle	7	5	15	6	5	6
Stone golem	8	5	25	8	3	4
Giant crab	8	5	15	12	5	6
Gorilla	9	7	15	3	3	4
Iron golem	9	5	30	10	3	4
Hill troll	12	4	20	5	1	4
Sasquatch	9	6	15	4	6	5
Cave gobbler	10	8	3	1	1	4
Sabertooth	10	6	20	7	5	6
Giant spider	8	7	15	6	5	6
Gorgon	6	9	20	4	6	4
Minotaur	10	7	20	5	6	5
Crocodile	10	7	20	8	3	4
Grizzly bear	10	8	20	7	5	6
Tigerman	11	8	18	7	6	6
Lionman	11	8	15	7	6	6
Rhinoman	13	7	20	9	3	4
Cave bear	12	8	20	8	5	6
Hydra	12	6	30	9	6	5
Cyclops	11	8	30	9	7	5
Treeman	15	5	50	20	3	1
Dragonling	11	7	40	10	7	10

Let's look at one individual attribute from this table—STR (strength)—and do an overview analysis of it. The simplest way to get an overview is to get

the average of the STR values for all of the characters. After that, you can find the median and compare it against the average. [Figure 15.1](#) shows these two calculations.

A	B
Attribute	STR
Average	7.42
Median	7

Figure 15.1 Finding the average and median of the STR attribute for a list of characters

This comparison provides a little bit of insight. You can see that the average and median are less than one whole point off from each other, so it is likely that there is a somewhat even distribution of values for this attribute (although that is not guaranteed to be the case). You can also get a feeling for the attribute numbers this list probably contains; in this case, medium to large single-digit numbers are likely.

The next step in understanding the data is to know the size of the data set (that is, the number of items in the data set). For this, you can use the COUNT function to count the number of STR attribute values included in the data set (see [Figure 15.2](#)).

	A	B
1	Attribute	STR
2	Count	52
3	Average	7.42
4	Median	7

Figure 15.2 Finding a count of items in the data set

Now you know how big the data set is, and you can make a few determinations based on this knowledge. The smaller the data set, the more outliers will affect the data. If you only have a list of 10 items, for example, even 1 being an outlier will easily throw off the results, but if you have 10,000 items, the data set is so large that a few outliers are likely to go unnoticed. In this example, the count of 52 means that large outliers would likely be visible in the comparison of average to median, but small ones might not.

Another observation you can make in this comparison is that the average is indeed a bit larger than the median. This means there are likely a few more large numbers in the data set, or there may be one or more rather large numbers that skew the results. You have already gleaned quite a bit of information from only three checks.

To dig even deeper, it's useful to know what number comes up in the data the most frequently. This is the *mode*, and you can find it with the MODE function. The mode also has some limitations, so let's look at one more small example before applying it to the RPG characters. In the following list of numbers, what is the mode?

2	9	15
3	9	16
4	10	17
5	11	18
6	12	19
7	13	

This example is simple enough that you can figure it out without a spreadsheet. The number that appears most frequently in this list is 9—but it is also the only number that is duplicated. So, is knowing that 9 is the mode important for understanding this list? Probably not. The list contains only a few numbers, and almost all of them are unique. The mode can be useful, but as with the average, you typically need another check to determine how much of a statement the mode value is making. In this case, you want to know what proportion of the numbers in the list are the mode. If very few are the mode, then the mode value is not particularly important. If the percentage of values that are the mode is higher, then the mode is a more important factor to watch. In the case of this simple list, 2/20 numbers are the mode; this works out to 10%. This percentage is small enough that the mode is not a very important factor when evaluating this data. [Figure 15.3](#) shows a number of formulas applied to the RPG character data.

	A	B
1	Attribute	STR
2	Count	52
3	Average	7.42
4	Median	7
5	Mode	6
6	CountMode	10
7	Mode %	19.2%

Figure 15.3 Several of the most typical methods of doing high level analysis.

In [Figure 15.3](#), you can see that the mode is smaller than the median and average. You can also see that 19.2% of the characters in the game have a STR value of 6. While 19.2% is not a huge percentage, it does provide some information. In this case, it shows that a sizable portion of characters have a STR value that is lower than the average or median, signaling that there are likely a few higher outliers in the data.

Next, you need to find the extent of outliers that exist in the data. To do this, you can find both the minimum and maximum values for that attribute. To take it one step further, you can do a count of how many are the minimum value and how many are the maximum value. [Figure 15.4](#) shows a set of functions you can use to get helpful numbers from the character data, which is on another sheet.

	A	B
1	Attribute	STR
2	Count	=counta(ChaData!B2:B)
3	Average	=AVERAGE(ChaData!B2:B)
4	Median	=MEDIAN(ChaData!B2:B)
5	Mode	=mode(ChaData!B2:B)
6	CountMode	=countif(ChaData!B2:B, B5)
7	Mode %	=B6/B2
8	Max	=max(ChaData!B2:B)
9	Max Count	=countif(ChaData!B2:B, B8)
10	Min	=min(ChaData!B2:B)
11	Min Count	=countif(ChaData!B2:B, B10)

Figure 15.4 Functions used for this calculations; exposed

Figure 15.5 shows the results of these functions. In these numbers, you can see how common the most extreme outliers are.

	A	B
1	Attribute	STR
2	Count	52
3	Average	7.42
4	Median	7
5	Mode	6
6	CountMode	10
7	Mode %	19.2%
8	Max	15
9	Max Count	1
10	Min	2
11	Min Count	1

Figure 15.5 Adding Max, Min, and counts of values for a deeper dive

When you look at all the information provided in [Figure 15.5](#), the picture of what is going on in the data becomes a bit more clear. You can make some commonsense observations like the following:

- This attribute has a fairly wide possible distribution, but most of the numbers are clustered around 7.

- The mode, 6, belongs to 10 characters, where the minimum and maximum each only have 1. This suggests that there is likely to be a bell curve, where most of the values fall in the center of the range, and just a few are at the high and low ends.
- Given that the average is only a little bit larger than the median, but both are much smaller than the maximum, it's likely that there are very few characters at the upper end.

All these observations together paint a picture of a fairly standard set of characters where most are around average, and there are a few stand outs on the high and low ends.

The final step, which is also the most important step, is to go back to the raw data and observe it. The STR attribute, at least, should start to make a lot more sense when you scroll through that data. It doesn't really matter that you don't yet know what the attribute actually does in the game. This is also a great time to start pairing names (and possibly descriptions) with the data. What names of characters are associated with larger numbers, and which ones are associated with the smaller ones? By doing this type of analysis, you can start to form a picture of what each of these character types will be like in the game.

By doing so much analysis for STR, you have created a format to do the same type of analysis for each of the other attributes. If you run the same functions shown in [Figure 15.4](#) for all the other attributes, you get the results shown in [Figure 15.6](#).

	A	B	C	D	E	F	G
1	Attribute	STR	DEX	HP	AV	DG	MV
2	Count	52	52	52	52	52	52
3	Average	7.42	5.77	15.81	5.50	3.90	4.90
4	Median	7	6	15	6	3.5	5
5	Mode	6	5	15	6	3	6
6	CountMode	10	11	12	9	13	15
7	Mode %	19.2%	21.2%	23.1%	17.3%	25.0%	28.8%
8	Max	15	9	50	20	8	10
9	Max Count	1	1	1	1	2	1
10	Min	2	2	3	0	0	1
11	Min Count	1	1	1	1	1	2

Figure 15.6 Expanded attribute analysis

From this view of the data, you can make even more observations:

- HP is on a larger scale than the other attributes. MV and DG are on a smaller scale.
- All of the attributes are tending toward middle-sized single-digit numbers, and there are some outliers.
- The average and median scores for all attributes are close together, indicating a consistency in the spread of character data for this game.
- The mode is more telling for some attributes than for others. There is likely an in-game reason that so many characters have 6 MV. When games use single-digit numbers or small digit integers, it's likely that

the mode will be a higher percentage of the data objects because there simply are not enough numbers at that scale to have large amounts of variation.

- None of the min count or max count values have many instances. This indicates that most of the character attributes cluster more closely toward the center, and a few attributes stand out as being extreme.

For a basic high-level view of a game, this analysis might be enough. This is the level of data analysis that would work for playing a game at a serious level or evaluating a simple game. In some cases, you need to do further analysis, as described next.

Next-Level Deep Analysis

For more complex or larger games, you may need to dig in a bit deeper. The next step in this research is to find the numbers that are involved specifically for each attribute. The goal of this analysis is to find out what numbers are being used for each attribute and how often each of them is being used. You can find this information by using a spreadsheet and the functions SORT, combined with the function UNIQUE. This compound function can look at a list of repeating numbers and pull out only one instance of each number and sort those instances into an ascending list. [Figure 15.7](#) shows how you use this compound function to find the STR attribute distribution in column B of the ChaData sheet.

fx | =sort(unique(ChaData!B2:B),unique(ChaData!B2:B),true)

	A	B	C	D	E	F	G	H	I	J
1	Attribute	STR	DEX	HP	AV	DG	MV		STR	STR #
2	Count	52	52	52	52	52	52		2	1
3	Average	7.42	5.77	15.81	5.50	3.90	4.90		3	6
4	Median	7	6	15	6	3.5	5		4	3
5	Mode	6	5	15	6	3	6		5	2
6	CountMode	10	11	12	9	13	15		6	10
7	Mode %	19.2%	21.2%	23.1%	17.3%	25.0%	28.8%		7	7
8	Max	15	9	50	20	8	10		8	5
9	Max Count	1	1	1	1	2	1		9	3
10	Min	2	2	3	0	0	1		10	6
11	Min Count	1	1	1	1	1	2		11	4
12									12	3
13									13	1
14									15	1

Figure 15.7 Using SORT and UNIQUE to create a list

You already knew from the earlier checks that the maximum value is 15, and the minimum is 2. The procedurally generated list also finds the same thing. Notice also that there is no 14 in the list because no character had an attribute value of 14. In this list, all of the same things are apparent as in the first breakdown of the attribute in column B, but this time you see more granular results. In column J, you see a COUNTIF function that does a check for how many times each of the possible values shows up in the entire list of data. Just as you would expect, 6 is the most common value; you already knew that from the mode. Interestingly, you can see that this is not exactly a uniform bell curve distribution. There are very few 5s and 4s, and there are more 3s. On the upper end, the numbers taper off more, as expected, with a few minor exceptions. [Figure 15.8](#) shows this data charted.

STR # vs. STR

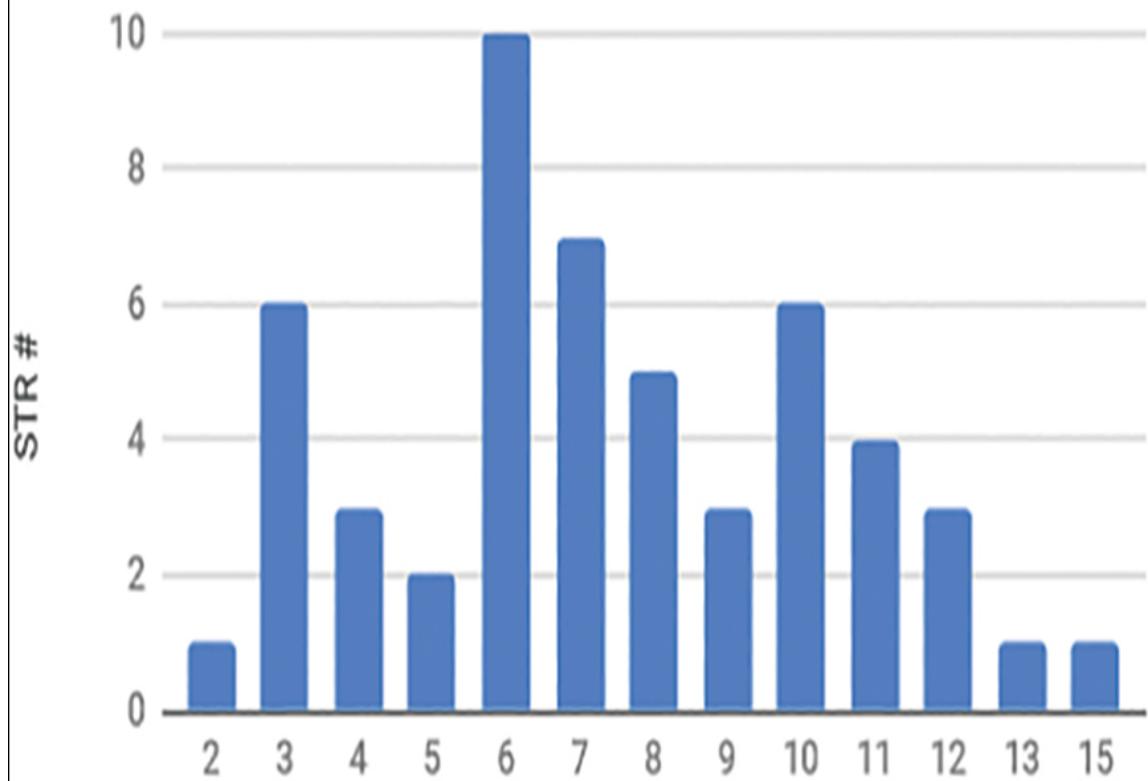


Figure 15.8 Chart of data

The chart gives you even more insights into how the data works in the game. It looks like there are three peaks of frequency that then taper off into the next peak. This indicates that there might be “classes” of STR among characters—perhaps low, medium, and high or weak, average, and strong—with variations in those groups. You can apply similar analyses to all the attributes to get a full readout of the distribution of each attribute value set (see [Figure 15.9](#)).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Attribute	STR	DEX	HP	AV	DG	MV		STR	STR#	DEX	DEX#	HP	HP #	AV	AV #	DG	DG #	MV	MV #
2	Count	52	52	52	52	52	52		2	1	2	1	3	1	0	1	0	1	1	2
3	Average	7.42	5.77	15.81	5.50	3.90	4.90		3	6	3	3	5	2	1	6	1	5	2	3
4	Median	7	6	15	6	3.5	5		4	3	4	10	6	2	2	5	2	7	3	1
5	Mode	6	5	15	6	3	6		5	2	5	11	7	3	3	6	3	13	4	13
6	CountMode	10	11	12	9	13	15		6	10	6	7	8	4	4	3	4	4	5	14
7	Mode %	19.2%	21.2%	23.1%	17.3%	25.0%	28.8%		7	7	7	9	9	1	5	4	5	9	6	15
8	Max	15	9	50	20	8	10		8	5	8	10	10	4	6	9	6	9	7	2
9	Max Count	1	1	1	1	2	1		9	3	9	1	12	2	7	5	7	2	8	1
10	Min	2	2	3	0	0	1		10	6		13	2	8	3	8	2	10		1
11	Min Count	1	1	1	1	1	2		11	4		15	12	9	5					
12									12	3		17	1	10	3					
13									13	1		18	1	12	1					
14									15	1		20	10	20	1					
15												25	2							
16												30	3							
17												40	1							
18												50	1							

Figure 15.9 Expanded attribute analysis

With this fuller picture, even more patterns emerge:

- You can see that MV has a very tight grouping around 5.
- DG has one value, 3, that seems to be very important for this game because it is much more common than the other numbers.
- AV has a more traditional peak in the middle and then tapers off, but it has one very high outlier on the high side and one very low outlier on the low side. It would be easy to miss these outliers because they counterbalance each other in the full list of values.

- DEX has a weaker pattern than the other attributes and a more linear distribution.

You could easily keep going deeper with analysis, but with just the tools used so far, you already have a much better understanding. Also, the deeper the analysis goes, the more individual it becomes to the game, the data, and the attributes. The methods discussed so far can apply to any game that has lists of data objects that have attributes.

Practicing Data Analysis

If you had a hard time following along with the observations in the previous sections, keep in mind that those observations are the most difficult part of analysis. The good news is that it is fairly easy to practice making such observations.

One of the best ways to practice is to try out the previous methods of analysis on games you already know. Both official websites and fan-made sites are full of data sheets containing lists of data objects and their attributes. You probably want to start with a game whose feel you already know. Then you can do an analysis of the game and note what patterns in the game caused that feeling to emerge to you as the player. What mechanics and technical limitations do you already know about the game that may have driven the numbers you find in the data attributes? By going through this process for several games, you might start to see the connections between what the data is doing and what the game feels like.

Comparison Analysis

With complex games that have large lists of data objects, it can be helpful to focus on a comparison of two objects in isolation. For example, with the earlier example of the RPG characters, you might want to look at two characters at a time and compare their stats directly. To do this, you can use data validation (as described in [Chapter 5, “Spreadsheet Basics”](#)) to create a drop-down list of all the data objects and then use VLOOKUP to pull up each of the attribute values for the two data objects. [Figure 15.10](#) shows an example of this.

	<code>=VLOOKUP(\$B2,ChaData!\$A:\$G,7,false)</code>	A	B	C	D	E	F	G	H
1	# Character	STR	DEX	HP	AV	DG	MV		
2	1 Zombie	4	2	6	0	0	2		
3	2 Barbarian	7	3	13	2	4	5		

Figure 15.10 Using VLOOKUP for Data object comparison

Here you can see the zombie character in direct comparison to the barbarian character. Once you have this comparison set up, it is easier to go further and more closely examine the data. The following are a few examples of what you might want to look at:

- A difference in attributes between the objects
- A plot of where each character falls in comparison to the minimum and maximum
- Charts that further show detailed comparisons between the objects.

When you isolate a small number of data objects for analysis, you can further expand the depth of analysis done for each of them. Each game calls for unique analyses, so take the time you need to think about what comparisons would be most useful to run on various pairs.

Canaries

There is a common problem when analyzing new or very foreign data: You don't always know when you have made a mistake. When you don't know the expected results of the analysis, you can't know for sure if the generated answer is right or wrong. This is a difficult problem to completely eliminate, but there is a tool you can use to help safeguard against it: a data

canary. If the canary finds that something is wrong with your data, then you know there might be a bigger problem.

Note

The idea of the data canary is based on the canaries coal miners used to determine if the air in parts of a mine was safe to breathe.

To see how a data canary works, consider the list of percentages in [Figure 15.11](#). You know intuitively that the sum of all percentages is 100% in most cases. So, is the list in [Figure 15.11](#) correct? It's very difficult to tell just by looking at the list. This type of problem can appear anywhere that there is data analysis: It may look like all the results are correct, but it's hard to tell for sure if they are.

	A	B
1	Things	Percent
2	Thing 1	11.00%
3	Thing 2	2.00%
4	Thing 3	15.80%
5	Thing 4	3.00%
6	Thing 5	4.00%
7	Thing 6	2.50%
8	Thing 7	2.00%
9	Thing 8	4.50%
10	Thing 9	3.10%
11	Thing 10	1.00%
12	Thing 11	0.04%
13	Thing 12	6.00%
14	Thing 13	3.00%
15	Thing 14	12.00%
16	Thing 15	4.00%
17	Thing 16	6.00%
18	Thing 17	7.00%
19	Thing 18	3.50%
20	Thing 19	1.00%
21	Thing 20	8.20%

Figure 15.11 Do these percentages total 100%?

The canary is a calculation for which you already know the result, either because it is intuitive—such as 100% being the sum of percentages—or because you can double-check it by calculating the same result in two different ways (for example, using the function AVERAGE and then using SUM and COUNT to get the average). Redundancy and checks like this can reduce mistakes to a great extent (although mistakes may still happen). In the list of percentages in [Figure 15.12](#), you can see that there is a mistake of some sort because the total is less than 100%.

	A	B
1	Things	Percent
2	Thing 1	11.00%
3	Thing 2	2.00%
4	Thing 3	15.80%
5	Thing 4	3.00%
6	Thing 5	4.00%
7	Thing 6	2.50%
8	Thing 7	2.00%
9	Thing 8	4.50%
10	Thing 9	3.10%
11	Thing 10	1.00%
12	Thing 11	0.04%
13	Thing 12	6.00%
14	Thing 13	3.00%
15	Thing 14	12.00%
16	Thing 15	4.00%
17	Thing 16	6.00%

18	Thing 17	7.00%
19	Thing 18	3.50%
20	Thing 19	1.00%
21	Thing 20	8.20%
22		
23		
24	Sum	99.64%

Figure 15.12 Mistake found

To examine a slightly more sophisticated example, let's revisit the RPG characters. This time, say that you want to double-check the distribution of attribute values. You know that the count of attributes is 52, and you can do a quick sum of attribute counts for a single column of distributions to make sure it comes up as 52 as well (see [Figure 15.13](#)).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Attribute	STR	DEX	HP	AV	DG	MV		STR	STR#	DEX	DEX#	HP	HP#	AV	AV#	DG	DG#	MV	MV#
2	Count	52	52	52	52	52	52		2	1	2	1	3	1	0	1	0	1	1	2
3	Average	7.42	5.77	15.81	5.50	3.90	4.90		3	6	3	3	5	2	1	6	1	5	2	3
4	Median	7	6	15	6	3.5	5		4	3	4	10	6	2	2	5	2	7	3	1
5	Mode	6	5	15	6	3	6		5	2	5	11	7	3	3	6	3	13	4	13
6	CountMode	10	11	12	9	13	15		6	10	6	7	8	4	4	3	4	4	5	14
7	Mode %	19.2%	21.2%	23.1%	17.3%	25.0%	28.8%		7	7	7	9	9	1	5	4	5	9	6	15
8	Max	15	9	50	20	8	10		8	5	8	10	10	4	6	9	6	9	7	2
9	Max Count	1	1	1	1	2	1		9	3	9	1	12	2	7	5	7	2	8	1
10	Min	2	2	3	0	0	1		10	6		13	2	8	3	8	2	10		1
11	Min Count	1	1	1	1	1	2		11	4		15	12	9	5					
12									12	3		17	1	10	3					
13									13	1		18	1	12	1					
14									15	1		20	10	20	1					
15												25	2							
16									canary	52		30	3							
17												40	1							
18												50	1							

Figure 15.13 Using a canary to check the accuracy of the results

By consistently using canaries in this way, you can have a much greater degree of confidence that most or all of your data is doing what you expect it to do.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the concepts of data analysis in both a high-level overview and in more detailed review.

- Create a spreadsheet and either import or copy and paste in data from a favorite game. Take some time to clean up the data and ensure that it is arranged in a way that is easy to evaluate (for example, ensuring that each attribute has its own column and that each data object has its own row).
- When you have data imported and tidied up, use the methods presented in this chapter to do an analysis of the data. After you use all the basic methods described here, look to see if the game you have chosen lends itself to any other form of analysis. Which type of analysis works best with your game—a broad overview of all data or a more detailed comparison of individual objects?
- Start to draw conclusions about how your chosen game plays, given the data. After doing this exercise on a game, you should understand much more deeply why the system designers made the game the way they did. Also while doing this, keep an eye out for mistakes or loopholes. It's not common for them to slip through in modern games, but it does occasionally happen.

Chapter 16

Macrosystems and Player Engagement

You can use several different styles of difficulty adjustment to make a game harder or easier as a whole or to adjust it to a player's particular needs. You will often find, especially in larger games, that you can use multiple methods at the same time to get a better balance. This chapter discusses balancing the game at a very high level, across all the integrated systems.

Macrosystem Difficulty Adjustment

Difficulty adjustment is quite easy to implement in games, and it is often used for encouraging high levels of competition. The following sections discuss these broad categories of difficulty adjustment:

- Flat balancing
- Positive feedback loop
- Negative feedback loop
- Dynamic difficulty adjustment
- Layered difficulty adjustment
- Cross-feeding

Flat Balancing

A flat balanced game is inherently fair and never adjusted for either side.

Note

Flat balancing is the oldest form of balancing, and it is used for most sports games. It is also used for many old games like chess and

backgammon. Here's what it's good and bad for:

- **Good for:** Serious competition
- **Bad for:** Fun, accessible games

With flat balancing, it does not matter what either player does, what position the players are in, or how well or poorly they're doing: The game runs without adjustment.

For example, if two players play chess and Player 1 wins, you can assume that Player 1 is better. You know that the game has not done anything to make Player 1 have a better chance or worse chance of winning. The downside of this is that if Player 1 is considerably better than Player 2, then Player 2 will never win. Player 1 will always win, and the game will soon be boring for both sides. It's not particularly fun to play a game when you know you're always going to lose or always going to win. Part of the fun of playing games is the mystery of the outcome, and if the outcome is not mysterious in any way, then it's not interesting.

Flat balancing is really good for competitions where you want to find out who is the best, such as sports. In sports at the professional level, you don't want to see a close game each time; you want to see which side is better at the game. On the other hand, if you want a game to be competitive and fun, which is generally the goal with consumer-facing games, you don't want to use flat balancing.

Positive Feedback Loops

In addition to flat balancing, another common type of balancing is a *positive feedback loop*; it is also sometimes called a rich *get richer, poor get poorer* mechanic or a *divergent curve*. [Figure 16.1](#) provides an illustration of the concept of a positive feedback loop.

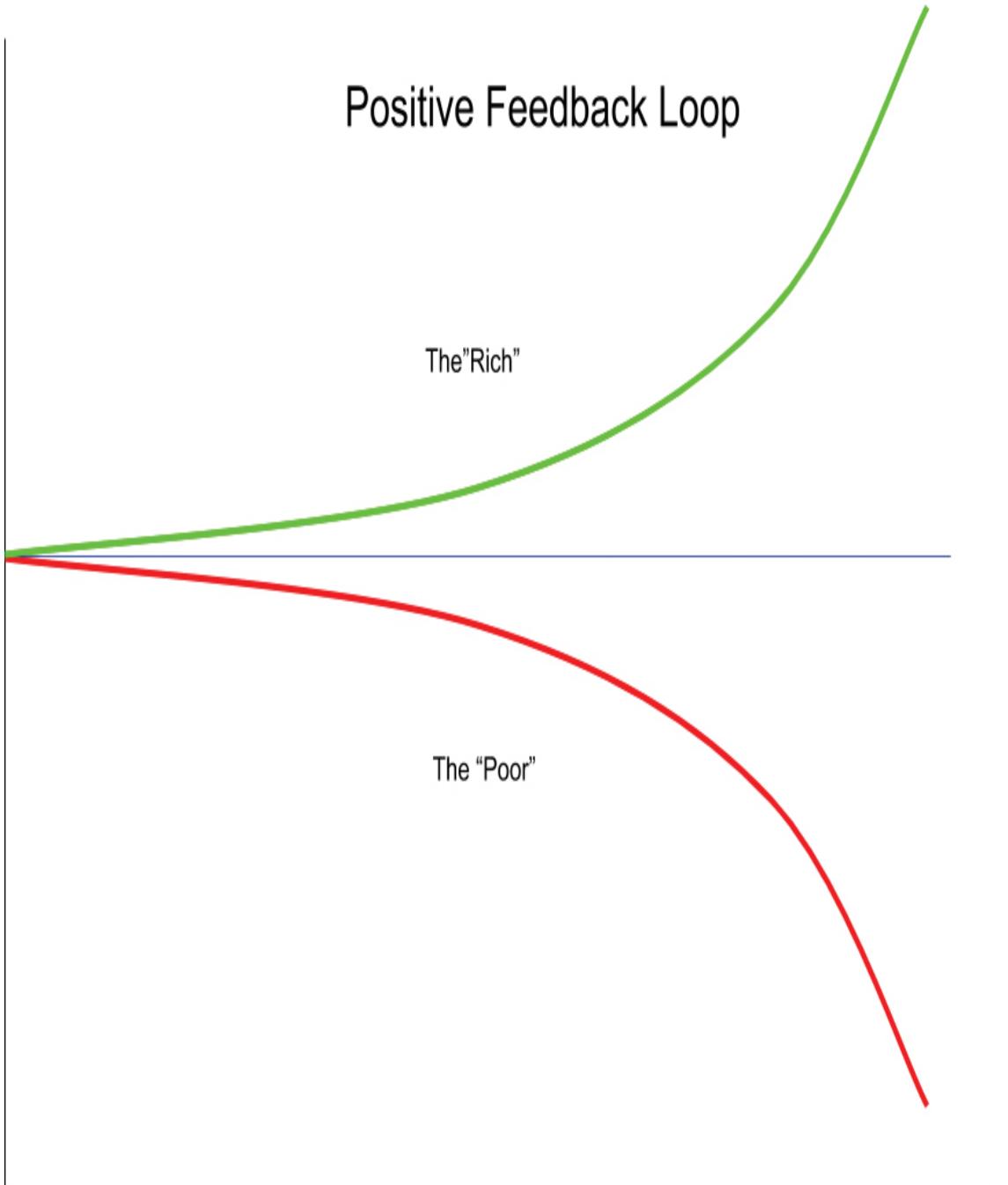


Figure 16.1 Positive feedback loop

Note

Positive feedback loops are used in poker and bowling. Here's what they're good and bad for:

- **Good for:** Breaking a stalemate, where scores are consistently too close
- **Bad for:** Allowing players to catch up and fun, not-so-serious serious games

A positive feedback loop works by diverging players to give the better players an advantage and put the worse players at a disadvantage. It is easy to implement this type of balancing in a game: All you have to do is reward players who are doing better with a reward that makes the game easier. If players do better and get better rewards, they are probably going to continue to do even better. In the same vein, you can also use this form of balancing to punish players for doing poorly. Both methods can cause a positive feedback loop.

For example, say that in an FPS, every time you get a successful shot, you are given a better gun that does more damage and has better accuracy. Conversely, every time you miss a shot, your gun gets worse. This would very quickly lead to the better players becoming completely unstoppable and the worse players having no ability to compete. In general, this may seem to be a bad thing because you don't want players to be completely divergent from each other. If you find that someone is continually getting better and you're continually getting worse, the game is going to cease to be fun.

However, you can use this mechanic for a few good effects. One of the reasons you would use a positive feedback loop is when a game tends to get stuck in a stalemate. If games are consistently very close or go back and forth, you can break the stalemate with a positive feedback loop. By rewarding a player who is doing slightly better, you will enable the player to pull away and bring the game to a conclusion.

Bowling is a really good example of a game where a positive feedback loop can be used well. In bowling, the most pins you can knock down with a single ball is 10. You get 10 chances to knock down all the pins, so a total score of 100 would seem to be the natural maximum. Once players start to develop skill at bowling, however they are able to get 10 pins each attempt

much more often. If the maximum score were 100, game scores would be very close and often tied. At the professional level, players would score around 96 to 98 points each and every game. This would leave little mystery in the progression of the game and would make the game less fun to play and watch. To keep things interesting in bowling, each time a player knocks down all 10 pins on the first attempt, it is considered a strike, and the frame is scored 10 plus the total of the player's next two frames. This allows a good player to get up to 30 points for a single frame and allows the top score to be up to 300 points. Because a single missed attempt can lead to a massive reduction in points, the scores diverge much further than it seems they would over a few missed pins—and the game becomes less predictable.

Another example is the classic game Monopoly. In Monopoly, the better you roll, the more properties you are able to own and the more money you get, which allows you to buy more properties, so you can get more money, more properties, and more money. This cycle continues until one player inevitably wins. If it were more difficult for players to get more money and property, the game would go on forever. Given that the game already takes a long time, a further extension of gameplay would likely cause severe boredom in players.

Negative Feedback Loop

Negative feedback loops are sometimes called *rubber banding* because a player who does well gets more penalties and fewer rewards. Conversely, a player who is doing poorly gets more rewards and fewer penalties. This acts like a metaphorical rubber band that holds back the leading player and pulls forward the trailing player. [Figure 16.2](#) illustrates this concept.

Negative Feedback Loop

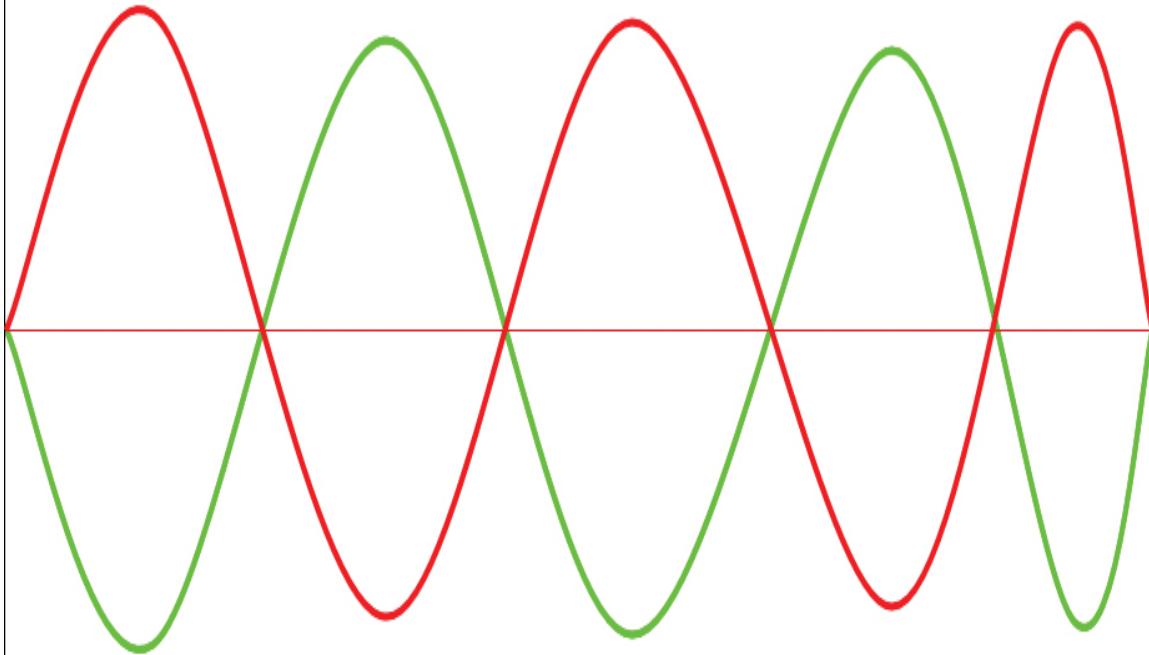


Figure 16.2 Negative feedback loop

The idea of a negative feedback loop is fairly unintuitive, because you would think you would want to reward good play and punish bad play, but it really depends on the kind of game you're creating. If you're creating a very competitive game, a negative feedback loop would be a bad thing as it tends to make the outcomes between better and worse players closer than

they otherwise would have been. But often you want to make games more inclusive of more players. This could be a style choice to make a family-friendly game or implemented to grow the game’s audience. Leveling the playing field allows better players to compete with lesser players and still have some mystery in terms of the outcome of the game.

Note

Cart-racing and party games typically use negative feedback loops. Here’s what they’re good and bad for:

- **Good for:** Light-hearted, fun games
- **Bad for:** Highly competitive games

One of the most obvious places this form of balancing is used is in mass-market cart-racing games—that is, games where players are racing around and can get little power-ups that can help a little bit or a whole lot.

Classically, in these games, the further behind you are, the better the power-ups you get, which allows you to catch up. Conversely, the further ahead you are, the worse the power-ups you get, which makes you fall back. This type of balancing tends to pull everyone toward the middle. This can be frustrating for very competitive players because it seems like the game rewards bad behavior and punishes good behavior—and this is exactly the case. Negative feedback loops are used in light-hearted games to make sure players of different skill levels all feel like they have a chance to win.

Negative feedback loops are commonly used for *handicapping*—that is, giving extra points to players with lower skill levels. Handicapping is used in the real world in bowling, golf, and horse racing. Let’s look at an example of a handicapped bowling video game. In bowling, recall from earlier in the chapter that the highest score a player can get is 300. Say that two players have the following averages:

Player 1: 100 points average

Player 2: 200 points average

Each player receives a handicap score that is the maximum score possible minus their average score:

Player 1: 300 max score – 100 points average = 200-point handicap

Player 2: 300 max score – 200 points average = 100-point handicap

The next time the players bowl a game, they each get to add their handicap score to the score they bowled on the most recent game. The less skilled player now has a handicap of 200 points, and the better player now has a handicap of 100 points. If these players were to play against each other with no handicap, and they were consistent, the first player would always lose and the second player would always win. With flat balancing, neither of them would go into the contest with any mystery about the outcome. In a high-level competition, this would be good, but for a friendly game, you want some mystery.

Now let's say these players face off against each other, and each of them scores 150 points in this game. To update the handicap for each of the players, you add up each player's most recent score and handicap. To Player 1's score of 150 you add the handicap of 200, so Player 1 now has 350 points—even though this is a higher total than is possible in bowling without a handicap. (When using a handicap system, you often see scores that are higher than naturally possible.) For Player 2, you also start with 150 and then add this player's handicap, which is 100, so Player 2 now has 250 points. Or, to put it more simply:

Player 1 handicap 200 + score 150 = 350 points for this game

Player 2 handicap 100 + score 150 = 250 points for this game

In this game, Player 1 wins even though both players bowled the same score. This may seem unfair, but handicapping levels the playing field by comparing a player against their average play instead of directly against their opponent. The scoring system is metaphorically saying “on this day, Player 1, who is a worse player in general, was having a good day. Player 2, who is a generally better player, was having a bad day. Therefore, today, Player 1 wins because they had a relatively better day.” In this way, you can

compare each player on the day, which creates some mystery in the outcome of the game.

Dynamic Difficulty Adjustment

Dynamic difficulty adjustment (DDA) is an interactive method of adjustment that relies on either human or AI decisions to adjust difficulty within the game in real time, depending on conditions.

Note

Examples of DDA include freemium mobile games and modern AAA action games. Here's what they're good and bad for:

- **Good for:** Ensuring a consistent experience among large numbers of players
- **Bad for:** Giving players unique or individual and controlling scope

One way DDA was commonly used before the advent of computers was by arbiters—and arbiters still use this balancing method in games today. A dungeon master or a game master can dynamically change the scenario based on intuition. The game master monitors the players and guides them through a scenario. A game master who finds that a scenario is too difficult for the players can, on the fly, make some changes and adjust the difficulty so that it is more fun for the group as a whole. A game master who finds that the players outsmart a scenario or are too good at it can also adjust the difficulty. Throughout a game, the game master may find that some sections are too difficult for the group and others are too easy, and they can adjust the difficulty in both directions. This is an analog form of dynamic difficulty adjustment.

Computers that run video games are the intelligence behind the flow of the games. We've gotten good enough at making games that we can program artificial intelligence to make the same kinds of adjustments that human game masters make. Computers, without human intervention, can adjust the difficulty of a game in real time to provide dynamic difficulty adjustment.

For example, if there is a boss in a game that a player tries to defeat multiple times without success and then, on the next attempt, the boss feels much easier, it is likely that some numbers have been changed behind the scenes to ensure that the player does not get frustrated and quit. The game changes in a quantified way, like reducing the hit points of the boss or reducing the damage output.

It's appropriate to use DDA when you want to provide a consistent experience for all of your players throughout a game. One of the big problems in creating mass-market games is that players' abilities vary widely. Some players do not have very good hand-eye coordination, and others have amazing hand-eye coordination. Some players are not very good puzzle solvers, and others are incredibly good. How do you get all these players through the same game and give them the same consistent experience? You can do dynamic difficulty adjustment behind the scenes to monitor the progress of players and adjust the difficulty so their experience feels similar. For the boss battle example, let's say that a player is not very good at the game and doesn't have very good hand-eye coordination. A second player is a professional-level video game player. As a designer, you want to have both players experience the boss battle, lose once or twice, and then get through the battle on the third or fourth try. You would encode into the game the desired metrics, such as the rate at which the boss is losing hit points compared to the player, the number of tries the player has attempted to beat the boss, and the time the player has spent fighting the boss. Then you set up trigger thresholds. For example, on the first attempt, if the boss is losing HP three times as fast as the player, this condition can trigger an increase in difficulty. Conversely, if this happens on the player's fourth attempt, it can trigger a decrease in difficulty.

It is sometimes not advisable to use DDA. For example, you typically don't want to use this form of balancing when you want each player to have a unique experience based on their ability and choices. If you think players of different skill levels should have different experiences in a game, then you absolutely would not want to use DDA. Big open-world RPGs are good examples of where you would not want to have much DDA because players are meant to be able to choose their own experience. Similarly, in sandbox games, you let the player choose their own experience and you let their ability take them where it does. You would also avoid using DDA when you

need to limit a game's scope. Defining each of the metrics and coding the triggers for DDA can be very difficult and can take nearly as long to do as making the game in the first place. You should count on spending at least half again as much time adding even light DDA to a project as you spent creating it.

Note

Different types of gamers have very different reactions to DDA. Some gamers who are out to prove their skill or create a unique experience will reject a game wholesale if they suspect any DDA in the game. On the flipside, many gamers who are more interested in the experience and smooth flow of going through a game to completion are attracted to DDA.

Layered Difficulty Adjustment

If you've ever played a game that has easy, medium, and hard modes, you've experienced layered difficulty. Layered difficulty adjustment simply means allowing players agency in adjusting difficulty on their own.

Note

Here's what layered difficulty adjustment is good and bad for:

- **Good for:** Letting players make their own choices
- **Bad for:** Giving players a consistent experience and controlling scope

Very often in games you will see that there's one path you can progress along that will give more rewards but involves more dangers. Then, there's another path you can take that is simpler to complete but presents fewer rewards. However, rewards don't even necessarily have to be tied to layered difficulty. A particular path or mode might just be more difficult for the

sake of being more difficult. In fighting games, when you can choose an opponent, and you choose more and more difficult opponents, that is also layered difficulty.

The benefits of layered difficulty are that it gives players a lot of choice, and it gives players agency over their choice. They feel in control and empowered.

In the games industry, a particularly difficult game might be called “90s hard.” In the 1990s, games were made almost exclusively for experienced gamers. There was an inflation of difficulty going on, where gamers kept getting better, so games kept getting harder—and the cycle continued. In those days, if you started a game and you couldn’t progress, that was the end of the game. Today, not allowing a player to go any further in a game would be seen as mostly unacceptable in mass-market games. Game makers want to get as many players through as much of the experience of the game as possible.

If a player is playing a game without layered difficulty and gets stuck, this often causes frustration. Very often, the player’s frustration is targeted at the game, the design team, or even the individual designer of the scenario. We hear comments that a game is “unfair” or “cheating.” However, if a game has layered difficulty and a player chooses to play on hard mode, the response is different. It is much easier for the player to accept that the difficulty may be their own fault. They know they have agency to adjust the difficulty down.

As another example, in an open-world game, say that you are encountering a creature in a lair. It’s a very difficult scenario, but you can easily walk around the area. If you choose to attempt the scenario and it’s very difficult, you can leave and come back later if you wish. You might feel frustrated at the difficulty of the scenario, but you may stop feeling frustrated pretty quickly, when you find another scenario that is a better match for you. If there were no option to stop a particular scenario and move on to another one, your frustration would likely continue to grow.

You should avoid layered difficulty when a game’s scope is an issue. Adding layered difficulty is not simply a matter of adding a multiplier to difficulty, as you’d do with damage outputs or health. In fact, it is

significantly more difficult than simply changing a few universal numbers. Not only do you have to rebalance the entire game to make a different difficulty level, it's usually not a linear change. Similarly, if there are areas of a game that a player can opt out of, then those areas of content—which the team dedicated time and resources to making—go unused.

Cross-Feeding

Cross-feeding is a method of balancing that allows the players to use different skills to complete the same task. Cross-feeding is often used in stealth combat games and RPGs with multiple classes.

Note

Here's what cross-feeding is good and bad for:

- **Good for:** Offering player choice and unique stories
- **Bad for:** Balancing difficulty and providing a consistent player experience

A classic example of cross-feeding is being able to defeat an enemy boss in various ways. For example, one way might be to take extra time to get very powerful gear that allows you to defeat the boss even if your skill at combat is lacking. Another way to defeat the boss might be to attack the boss directly with poor gear and rely on your ability at combat. In one scenario, the player is good at finding gear but is not particularly good at combat. In the other scenario, the player is good at combat but does not like searching for more powerful gear. With cross-feeding, you allow the player to tackle the task in either way.

Cross-feeding is most applicable when you have a large game with lots of interrelated mechanics. Giving the player a variety of ways to complete challenges makes the game feel very personal and allows the player a great deal of agency. It can lead to conversations between players where they discuss and debate the strengths of their own preferred method. For us as

designers, this is one of the highest compliments we can get from players in regard to the way we created our systems.

One downside of cross-feeding is that it is not possible in very simple games. Another downside is that it can be difficult to balance individual scenarios when the designer does not know how the player is going to tackle the presented challenge. Another problem that can arise with cross-feeding is that when there are several ways to tackle a scenario, one of them might be clearly the worst. In some games you might find that there are mechanics you can use but that don't seem to be very useful. It's quite likely that they were designed to be a cross-feeding balance mechanics, but the mechanics were not fully balanced and ended up being useless.

Poorly implemented cross-feeding can lead to wasted game-creation effort and player dissatisfaction. Consider, for example, a shooter who has lots of different varieties of guns. If the designer created 50 different styles of guns but 1 of them is clearly better than the rest, players will all gravitate to the overpowered gun and ignore the other 49 guns. This means that the designers and the art team who made those guns and the QA team who tested those guns wasted their time on those guns. Even worse, players will say the game is boring and lacks variety even though it has a large variety built into it.

Balancing Combinations

As games get more sophisticated, they are likely to use multiple methods of balancing in combination. They may implement negative feedback loops in the player-versus-player competitive mode and dynamic difficulty adjustment in single-player mode, for example. They may use flat balancing in certain areas in order to present the game fairly to all players, and they may use layered difficulty adjustment in other areas. The important takeaway is that the balancing methods described in this chapter can be used in combination with each other.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further

explore the various methods of balancing discussed in this chapter.

- Pick several games that you enjoy from a variety of genres (for example, video games, tabletop or board games, sports). Go through the rules of the games and identify what specifically in the rules causes each of the types of balancing listed in this chapter to happen in that game. Some games use only a single method of balancing, and others use multiple balancing methods together.
- After identifying the balancing methods in the games you choose, try to switch them in the rules and think about what would happen. Here are a few hypotheticals to get you started:
 - Basketball with a positive feedback loop
 - Monopoly with a negative feedback loop
 - Mario Kart with cross-feeding

Chapter 17

Fine-Tuning Balance, Testing, and Problem Solving

Most of a system designer's time is actually not spent designing. Most of a system designer's time is actually spent tuning, testing, and problem solving. This chapter discusses some of the specific methods that make this very large task more manageable.

Balance

Balance, in terms of game design, refers to massaging game data so that players feel that they have agency. Throughout a game, you want a player to constantly consider questions such as these:

- Does my RPG hero use a sword or an axe?
- Does my racing car use front-wheel or rear-wheel drive?
- Does my farmer plant corn or wheat?

In a good game, the answers to questions such as these are not always obvious. In addition, different players should have different opinions, and different situations in the game should have different solutions. You can consider a game's data to be balanced when there is a time and place or playstyle for every system and data object, and thought is required to make the right choices.

It is not enough to feel as though a game is balanced. You need evidence, based on clear and quantifiable guidelines. Some of this evidence will need to be observational in testing, but other aspects can have actual numbers applied that can be analyzed either with live testers or with telemetry.

Why Balance Matters

Players play games to have agency. *Agency* in a game means that the player has a real influence on how the game progresses. If you only offer players false options in a game, and they have no real agency, they will no longer have a reason to spend time on the game.

For example, say that you are playing a game and gearing up to fight the toughest battle you have yet dealt with. You are at the armory, choosing a weapon, and you have two choices:

- **The Sword of Destruction:** It hits 100% of the time and does 100 points of damage.
- **A ham sandwich:** It hits 2% of the time and does 1 point of damage.

This is not a legitimate choice. Not many players would choose the sandwich. Such “choices” are really chores. The game makes you select the armory and makes you click on a weapon, but it is clear which one you should pick, so you are simply going through the motions. You’re not likely to feel a connection to the game or have an investment in your choice.

A more balanced choice that actually gives the player some agency might look like this:

- **The Sword of Destruction:** It hits 100% of the time and does 25 points of damage.
- **The Sword of Kapow:** It hits 5% of the time and does 500 points of damage.

If you work out these choices mathematically, you find that they both do, on average, 25 points of damage per hit in the long run. (Refer to [Chapter 9](#), “[Attributes: Creating and Quantifying Life](#),” for information on attributes.) But practically, they will feel very different. With the Sword of Destruction, you know you can take down the opponent in regular chopping shots that do consistent and predictable damage. The Sword of Kapow, on the other hand, is a big gamble. You might be able to take out the opponent with a single shot and be done with it, or you might be swinging and missing for ages. Is this a gamble you want to take? There is not a clear choice. The

choice really depends on the player's personality. Individual players may be clearly drawn to one choice or the other, and there will be disagreement between players. Even an individual player may need to ponder this decision for a while. This is exactly what you want in your game.

Regardless of which choice the player makes, they feel as though they have had agency in the game and decided what is personally best for them. This strengthens the player's investment in further choices and the game as a whole. It can also strengthen the development of the player community. A "Team Destruction" and a "Team Kapow" could defend and argue for their weapon preferences. Getting a game in balance matters deeply to game designers because it gives players agency and encourages them to invest in the game.

Let's look at one final weapons choice example to illuminate another balance issue:

- **A ham sandwich:** It hits 2% of the time and does 1 point of damage.
- **A broken stick:** It hits 1% of the time and does 2 points of damage.

Mathematically, this is a balanced decision, but the problem is obvious: These are both terrible choices. Giving players false agency while also setting them up to fail is one of the best ways to frustrate them. In the end, you want players to have multiple balanced and viable options to choose from.

Another practical reason balance matters comes down to allocation of team resources while building a game. If you spend the time and resources to build and test 20 cars for a racing game, but players only ever choose 2 of them, then you have wasted time building and testing the other 18. This is clearly not what anyone wants. By making all 20 choices viable, you give more agency to the player and allow for a broader spectrum of players to feel investment in the game. You also fully allocate the time of the developers. There are many stories in the industry of developers reviewing telemetry to find out portions of the game they worked very hard on were viewed by only a tiny portion of the audience. This is a terrible feeling for any developer.

General Game Balance

While there is not an exact formula or set of criteria that can definitively prove that a game is balanced, there are several indicators that can be measured objectively and tracked. Theoretical numeric balancing can help you start balancing a game, but you don't really know whether you have achieved balance until the testing stage . When doing tests, you can look for the following indications that the game is getting close to or has achieved good balance:

- **Head scratching:** When a player understands the game and faces a choice, notice how quickly they make the choice. The choice may be picking between a sword or an axe before a battle. It could be the choice of using a sports car or a muscle car when racing on a new track. It could also be choosing between planting corn or wheat in a farming game. Any time there is a choice, you should be especially observant of the time testers take to make the choice and their commitment to the choice. If all the testers choose the axe within a second, then that is an indication of imbalance. On the other hand, if many of the testers pause, go back and forth between the options, ponder, and possibly ask questions before making a decision, this is a good indicator that the decision is balanced. It is important to note, however, that this pausing and asking questions can also indicate player confusion. It is important to make sure your testers are doing the head scratching because they are trying to come up with the best solution and not because they are confused or frustrated.
- **Disagreements among players:** It is important to let testers who made different decisions discuss why they made them. If there is disagreement—and, ideally, passionate defense of the different decisions—you are doing great balancing the game. Even if it is a group of only two testers, each with their own bias, this is a good sign. With this kind of feedback, you know you have done something to make both options feel viable to at least one person. On the other hand, if all testers quickly come to the same consensus, there is possibly an imbalance. This would be a great opportunity to ask the testers about the reasoning behind their decisions and look for clues about how to solve the imbalance.

- **Even distribution of choices selected:** When you work with a large pool of testers, you can begin to see patterns in the distribution of their choices. These patterns can emerge during the playtest stage, and they really start to show with telemetry. If you present players with five options, and each option is chosen roughly 20% of the time, you could not ask for anything better! Such an outcome is not likely, though, even with the most balanced games, as there are always some external factors involved that can influence players. It's not critical that all variations are chosen in exactly equal proportions, but it is important that they are all chosen at least some of the time.
- **Changing game conditions:** If players change objects or preferences throughout the game based on conditions, that is another excellent indicator of balance. This one is fairly easy to enforce artificially with concepts like damage types (for example, in the land of fire, only ice weapons do damage). Beyond the artificial forced changing of items and tactics, there are more natural, organic, forms of changing. Players' preferences change based on the conditions they see and the choices available. As long as players have multiple legitimate choices for each decision, and players don't all agree on an outcome, having players also change due to conditions is a great expansion of balance.
- **Rock paper scissors combinations:** Another good sign of balance is when any given option has a clear use and an Achilles heel, much as in the game rock paper scissors. Such combinations are often purposely built in to game mechanics but can also occur naturally as a game develops.
- **Change in lead:** Change in lead—one player or team taking the lead from the other side—can be a sign that a game is balanced. It is not always a perfect indicator but can offer some clues to balance in multiplayer games in particular. If a game has measured competition, watch for frequency of change in lead. This also applies to real-world games. Watch any given sporting event and pay attention to change in lead. In games where there is a high number of changes in lead, observers will more often consider the game to be exciting.

Breaking Your Data

When you have gone through the full gauntlet of tests and are starting to see the kinds of responses you want, it is tempting to stop—to “cash out” with a minor win. But don’t give in to this temptation. Instead, keep making changes. The more time you have, the more experimental your changes should be. You want to purposely break the bounds of the systems to see what happens. You should do this type of experimentation in isolation, safely away from the shippable game. During this important phase, game data can go from technically acceptable to truly innovative.

Problems with Balancing Judged Contests

If you have ever screamed at a referee or judge, you already know there are problems with judged contests. In many cases, people disagree with judgment calls and feel that contests are not fair. A game that is seen as unfair may be greatly diminished in the eyes of its players and audience. While older games may be able to get away with seeming unfair because they are embedded in cultural tradition and have been accepted for a long time, new games are at great risk of failure if they are seen as unfair.

There are two primary ways to deal with problems with judged contests:

- **Playing to the judge:** This is an elegant solution because it sidesteps the problem altogether. If the goal of a cooking contest is to make the judge happy, then that judge can fairly determine which dish makes them the happiest. Whether the judge’s pick is objectively the best dish can’t really be argued because the judge is not expected to be impartial. Instead, bias is built in to the contest as a mechanic. The biggest negative to this system is that it only determines the winner in the eyes of the judge; the judge doesn’t determine who is objectively better at something. For example, the judge might pick the winning dish because it is loaded with an ingredient he likes. The dish might not be the best designed or executed dish, but it contains something that plays well to that specific judge. The result of the contest really only shows the preference of the judge and not the ability of the contestants.

Note

Playing to the judge comes up all the time in the game industry in pitch sessions. If you are fortunate enough to be able to pitch a game idea to an executive or investor, you do not want to spend that time trying to prove that your game project is the best; instead, you want to show that your game is the best for them. An investor is fully allowed to be biased and may choose what you feel is an inferior project because they like it better. There is nothing wrong with this. Investors are free to invest where they want in what they want.

- **Quantification:** The other method of handling issues with judged contests is by clearly quantifying criteria that the judges use to minimize inconsistency. In video games, you have a massive advantage in doing this. Because the entire game universe is programmed by your team and fully under your control, you can set up exact and perfectly quantifiable criteria for literally every situation that needs it. For example, in a fighting game, unlike in MMA or boxing in the real world, you can 100% guarantee that there will be no illegal shots in the game. All you need to do is program the game so that it does not allow them. In the real world, there are always going to be gray areas.

If you are designing an analog game or sport, there are further measures that you can take to make sure all situations are quantified. Officials in games such as tennis used to have a difficult time judging exactly where a ball landed and whether it was in or out. Today we have high-speed cameras with computer controllers that watch over a game and can clearly show the state of the ball at any moment—and arguments over line judgment have been drastically reduced. Similar cameras are used in races to impartially prove, almost without any doubt, exactly who won.

The recommendation for an analog game would be to minimize—or ideally eliminate—any situation that could be interpreted differently by different people. A good way to determine whether you have properly quantified judged actions is to ask several qualified judges to observe several judged situations and then give their opinions. If all judges consistently come to the same conclusion, then you can be sure that the

mechanic has been properly quantified and is ready. If judges disagree, on the other hand, you know you have a problem with the mechanic.

How to Start Balancing Data

To balance a game, you need to do a lot of testing. To fully test video games, you can't just jump right into testing. First, you need a code base, an engine, data structures, systems, placeholder art, and much more. But you can start testing for balance before any of that.

Prototyping

One way to approach balancing is with prototyping. *Prototyping* involves making a small piece of a game with the specific intent to gain knowledge from it. Prototypes are intended to be as cheap and fast to make as possible. They are also designed specifically to be thrown away after they have served their purpose.

Note

Do not make the mistake of slapping together a prototype and then falling in love with it and wanting to keep it. Make it with the clear intention of discarding it after use.

A concept related to a prototype is a *vertical slice*—an early version of a partially completed game. A vertical slice has many or even all the mechanics in place, much of data, and much of the art.

A vertical slice should include as much of the game as possible so you can see how the game as a whole is trending. With prototyping, on the other hand, you purposely exclude as much as possible so you can focus on a single aspect in detail.

The following sections provide a few examples of where you can use prototypes.

Crafting Prototype

Say that you have created a crafting system for an exploration video game. Instead of building it in a full game engine, you are going to build some of the data in a spreadsheet and then make rules and actions into physical notecards. You plan to make a temporary analog set of rules that mimic how the mechanics work, and then you will test the system. You will leave out as much as possible, including harvesting mechanics, leveling mechanics, graphics, and UI elements. You want to focus on just one factor with the prototype: how the materials stack up and get exchanged for completed items.

As you play this prototype, you should be looking for items that are used all the time or none of the time. Are there certain combinations that turn out to be the obvious choice in most situations? Are there items or combinations that never get used? Finally, if you find a weakness, you need to figure out whether it is a symptom of the game data or whether it is more likely a symptom of the prototype. Keep in mind that you can't port all data into a prototype easily, so you can't have 100% certainty that everything in a prototype will be analogous in the finished game.

By focusing on just these aspects, you can get started prototyping before a single line of code is written or a game engine is chosen. Often, it takes only a few hours to go from a prototype idea to a completed test. If you are lucky, you might find that this prototype is actually fun. It may even spawn a spinoff idea. This has happened several times in the industry. But don't worry if it does not. You are not trying to make a completed game; you just need to focus on a single factor.

Leveling Up Prototype

Say that you have an RPG that allows characters to choose new skills from a skill tree after earning XP. Again, you can go fully analog and test this system for a specific purpose—in this case, to find the best pace of XP gain to fuel the leveling system. Again, you can use notecards to represent the skills. You can draw out skill trees with stats on a white board or paper. You can even give out XP as poker chips, coins, or using some other proxy. You won't be able to tell a whole lot about the game as a whole from this

prototype, but you will be able to drill down on one single aspect and work out many of the details before the game is built. While it is likely that there will need to be some tweaks to the system once it is integrated with the rest of the game, the prototype will at least give you a starting point to work from. It is possible, with some practice, to build all the systems exterior to the game and fairly accurately predict how they will operate once integrated.

Jumping Prototype

Now say that you are going to make a classic platformer. Instead of creating levels for the game, you can create an empty level in an off-the-shelf engine (such as Unity or Unreal) with just a floor, no enemies, no power-ups, and no goals. This is called a *proxy test* because you won't be using your actual game engine but rather a stand-in or proxy engine. It could be any engine or system you can modify enough to find the information you are looking for. Many game engines can be easily modified and used as proxy engines for prototyping. When you are planning to discard a prototype test, you don't need to worry about licensing, hardware constraints, and other baggage that come with using an engine for a full game.

Once your proxy game is set up, you can make your main character and put in numbers for their jump—velocity, direction, controls, and so on—but just the smallest attribute set possible to get the character jumping around. Then you can write down an outcome—something like “Find the jump height that feels best.” Next, you need to playtest the game...a lot. Jumping around in an empty room would be a very boring game, but with the prototype, you are not looking for fun. Instead, you need to focus on the jump mechanic in isolation, considering the size of the character onscreen and the portion of the screen the character should cover in a single jump.

By the time the prototype test is over, you should have some good numbers to start with. This test will likely lead you directly into a new prototype test as well. Once you get the jump feeling the way you want it, for example, you might be ready to give the player character a box to jump over. Then you can test again. The cycle repeats, the tests get slightly more complex, and your focus becomes more clear. By the time the real game engine,

character models, and animations are ready for use, you will have excellent starting guesses for your data attribute values.

Macrosystems Prototype

Say that you plan to build a large-scale war game in which players have battles in small locations that have ramifications on a larger battle map. You want to determine how to deal with the information from the small battles on the large map. In this case, you can use a combination of analog and proxy prototyping. For the small battles, you might choose a preexisting game that is in roughly the same ballpark as the game you want to build (even if it is not a perfect fit). You can then play out battles in that game and take the results to a fully analog board game map you create on paper. So part of the testing is done in another game, and part of the testing is done on a large sheet of paper.

Note

Using any combination of proxy, analog, and any other methods to move on to testing as quickly as possible will get you test results faster and allow for more iteration. Some teams have even gone as far as using action figure toys, rubber bands, and miniature models to quickly lay out scenarios and test game mechanics. The prototype phase is the best time in the life cycle of game development to get creative and crafty. It also gives the system and data designers something to do while the core engine is being built and prevents them from being idle while waiting out the lag time.

Remember that your first iteration will be the worst, so you should just get it out of the way and move on to the next one quickly.

Performing Playtests

Playtesting is much more testing than playing. Depending on the phase of game creation and the desired outcome, different kinds of playtests can be used. Playtesting tends to start with minimum viability testing and then

moves to more types of testing as the game progresses. As testing expands, the early types of testing remain constant, and the scope of testing increases.

When choosing testers, keep in mind that you only get fresh eyes once. So if you have a limited pool of testers, you shouldn't use them all for the first tests. If possible, you should trickle in new testers with every new round of tests, always holding back a few of them for later tests. Someone who has never seen the game before is going to have a very different perspective than someone who has already had some exposure to the game. Further, testers who see early versions of the game are going to have a unique bias. They will be able to mentally compare what they saw in the first version they experienced with what they see now. Your full audience will not have this bias, so it makes those testers, in some ways, less valuable for measuring how the game will be received once it ships. For example, if a tester at an early phase complains about a specific level and then that level is changed, the tester is going to be much more likely to artificially praise the change, regardless of whether the game is now better, because they feel a personal connection in asking for the change and seeing it made. This is not to say that testers are useful only once. Sometimes, having a tester who is already ramped up and familiar with the game is a great asset. It saves time with instructions, and you know the tester is at least enjoying the experience enough to do another test voluntarily. It is mainly important to keep in mind that experienced testers will behave differently from fresh testers and to adjust your expectations accordingly.

The following sections discuss the primary divisions of playtesting:

- Minimum viability testing
- Balance testing
- Bug testing
- User testing
- Beta/postlaunch telemetry testing

Minimum Viability Testing

Minimum viability testing is usually done directly by the game creator, and it should be done constantly. As you make changes to a game, you must test them. During the minimum viability testing phase, you want to ensure that any change you have made works as intended. It's much better to err on the side of too much testing rather than not enough. For a video game, this means (possibly) compiling the game in the engine and then playing quickly to see if the change worked. For changes in mechanics or code, this type of testing should be done every time a single change is completed.

If you are planning to add a lot of changes, it might not be feasible to test each and every data unit. For example, if you have just built 200 different weapons for a game in a spreadsheet and are ready to port them over to a game engine, it would not be practical to check each one as it is imported. Instead, at this point, it is advisable to do a test of the first one, without any others. To do a thorough check, you need to consider questions like the following:

- Can the hero equip the weapon?
- Does the weapon do the amount of damage intended?
- Does the weapon hit as often as possible?
- Does the weapon work with existing animations?

Once the first data object—usually the fulcrum—is implemented and tested, you can move up to a small batch, such as five data objects. Are you able to successfully implement five in a batch and then have one or two of them pass the same tests? If so, you can do a bigger batch, each time spot testing to see if the data is importing properly. You should never assume that if you set up one thing and it works, it will work properly every time.

Once minimum viability tests are successful and you have enough data that you can do more exhaustive scrutiny, it is time to move on to balance testing.

Balance Testing

After data has been entered into a game, it's time to start balance testing. If the data has been compared against a fulcrum, then balance testing should

go quickly. As mentioned in [Chapter 13, “Range Balancing, Data Fulcrums, and Hierarchical Design,”](#) you need to start by testing the fulcrum against itself. Then you can pick the data objects that are furthest from the fulcrum and test them against each other, as this is the most likely place to find imbalances.

For example, let’s say you are working on a fantasy RPG, and you made the knight your fulcrum. Orks, goblins, kobolds, barbarians, Vikings, and soldiers are all fairly similar characters, but when you built archers, you gave them ranged weapons instead of melee weapons, and when you built giant spiders, you gave them poison for attacking. So the archers and giant spiders are quite different from your fulcrum character and from each other, and even though you already balanced them against the fulcrum, you would want to plan to spot check this combination individually. The more special and unique a data object is, the more important it is to test it thoroughly.

The next step is to have someone else play with the new data. As the creators of the data, your team will have insights no one else would, and you will likely create the data in such a way that you know how to use it to its fullest potential. Other players won’t know any of this, and it’s important to get fresh eyes on a test. When doing a balance test with others, it’s okay to coach them on expectations. At this early point, you can explain controls, context for the test they might not know, and other information that is important for them to do the test the way you want it done.

While testing for balance, encourage your testers to speak a lot—and find talkative testers, if possible. A nonstop stream of consciousness from a tester is a great thing, especially at the balance stage. Ask testers to talk about their feelings a lot. Do they feel surprised? Annoyed? Puzzled? Do these feelings align with what is desired? When you hear statements such as the following from testers, you know you need to pay close attention:

- I don’t know what to do.
- This one feels way better than the other one.
- I know what to do, but I’d rather do something else.
- I am debating which of these options is best.

Statements like these during a balance test are gold. Be sure to take note of important statements like these and what prompted them.

Finally, you should be sure to balance test others' work. You will get better at running your own test when you know what it is like to be a tester under the microscope. When you are a tester, you should aim to be a good tester. As you will learn when you run balance tests, you really don't want to hear comments like "It's okay" and other vague, noncommittal statements that provide very little actionable information. When you are testing for others, be as specific as possible and use language that is as direct as possible.

Bug Testing

After a game is in production, it is common to bring on an independent quality assurance (QA) team. While the members of this team can certainly help with balance testing, their job is to find bugs. Their ultimate goal is to save the company money. If a large game goes to a publisher with game-breaking bugs, it will be rejected. This might push back release dates, make a publisher miss an advertising window, and in other ways increase the cost of the project. To avoid such problems, a QA team needs to do a very deep dive into a game to ensure that it is in good enough shape to go to the publisher.

As a game system designer, you need to be in communication with the QA team about your game, your systems, and your data. System and data designers in particular are often able to make content faster than they can test it themselves. You should start writing a plan for a QA test as soon as systems are built. QA team members can also be great balance testers. They are, after all, highly trained game players who play games all day every day. If possible, hang out with the QA team while they are testing portions of your work. They also tend to be very open about criticism (unlike typical game players). Be ready for harsh and direct feedback and embrace it when you get it. It does not feel good to hear all the ways you did something wrong, but it is by far the best way to get better.

User Testing

User testing comes into play as a game gets closer to completion. The goal of user testing is generally to find out how the users are interacting with the game, how the balance is working, and whether users are having fun.

Finding good test subjects at this stage is much more important than in previous rounds of testing. Ideally, the testers should closely match the target audience for the first few rounds. When asking for testers, specify the kind of tester you are looking for. You do not have to develop a profile as thorough as the target audience profile, but you should write a good enough profile to eliminate some people from your test pool. Not every player is going to be right for the game, and it is important to have the right testers. The following are some examples of calls for playtesters that specify the types of testers:

- Testers needed. Must enjoy fast-paced FPS games.
- Testers needed. Must enjoy free-to-play mobile games and be willing to play for an hour.
- Testers needed. Fans of anime and JRPGs are ideal, but any RPG players are welcome.

Note the jargon and acronyms in these calls. For user testing, this type of language is fine because players who don't understand it probably won't be the tester you want. Make your call for testers short, simple, and directly to the point. Don't worry if it will reduce the tester pool for now; getting the right kinds of testers at this stage is more important than getting a large number of testers.

For each user test you should outline a goal ahead of time. However, in most cases, you should not communicate this goal to the testers. You want them to have as little external information going into the test as possible. Here are a few examples of user test goals:

- How long does it take players to learn the controls?
- How well are players able to navigate the main menus and go through the tutorial?
- Do players tend to pick one character class over the others?

- What points in the game are most likely to confuse new players?

Feel free to collect as much information as possible, even if it is outside the framework of the stated goal. Pay particular attention to the testers as they go through the experience to get better insight into the stated goal.

For a user test, you may want to create a script for the proctor to read to a tester before the test begins. If the test is for the beginning of the game, the script might be something like this:

Please start the game and comment on anything you notice. The test will run for 30 minutes. Keep in mind we will not be answering any questions until after the test, but please ask them anyway. We will take down notes and make sure you understand everything clearly after the test.

Not only does a script like this help you stay organized, but it can be handed off to someone else. If you are using the same testers now and also later in the game cycle—which is a very common practice—you might need a little bit more in the script. You would want to describe where the tester has been placed in the game and what has come before, but that should be about it.

Note

To eliminate bias, some teams purposely use a proctor who has no personal attachment to the game. The proctor can then assure the testers at the start of the test that they are not involved with the game, and nothing the testers say will hurt their feelings.

The most important thing a user test proctor can do is observe. If possible, the proctor should be absolutely silent once the test starts and make observations only. As the game is nearing completion, it is critical for the game to be able to speak for itself. If at any point in the test the proctor must intervene, you know there is a rather large issue that needs to be addressed before more user testing can proceed. If the error is bad enough, it might be time to shut down the test for now, thank the tester for their

time, and go about fixing the issue. If a tester gets too far off in the weeds with errors, the team can't get good information from the test.

While testing, it is common to make video recordings of the testers while playing the game and the game as it is being played. When a tester really gets into the game, the notes can come faster than they can be written. A tester's face may also give great feedback that may be missed in an "over the shoulder" observation. When possible, you should have a camera pointed at the tester's face and also record the game at the same time. On larger projects, game developers can even record and play user input through the controller. Being able to go back and review all of these aspects of the testing in detail can provide feedback that would likely be missed otherwise.

After a user test, it is a good idea to conduct a conclusion interview with the tester and use prepared questions, but you can also dig more deeply into any comments the tester makes. A tester who is doing the testing in person and has volunteered their time is much more likely to give positive feedback than is a common audience, and it is important to remember this. It certainly feels good to hear a glowing review, but honest constructive criticism is more useful.

Note

Playtesters should not be professional QA folks (unless that is the target market of the game). Amateur playtesters do not have the same training or ability to communicate as professionals. This is not said to discredit them but as an important factor that the proctor must consider.

When formulating final interview questions, a proctor should avoid open-ended, ambiguous, and leading questions like "How did you like the game?" This particular question is both leading—in that it assumes the player did like the game—and too easily leads to the most dreaded answer a proctor can receive: "It was pretty good." This kind of feedback provides developers with almost no actionable information. It also shuts down the

conversation. Instead, the proctor should phrase questions in a way that encourages testers to offer up insightful information. Let's look at a couple of sample questions that ask basically the same thing but are phrased in subtly different ways.

Original question: “Did you find any of the levels you played confusing?”

This question often leads to answers like “No” or “Not really” that give you nothing to work with. If the tester says “Yes” or “Kind of,” that’s better, but the proctor will still need to do much more digging to get some good information.

Revised question: “Of the levels you played, which was most confusing?” and, after the first question is answered, “Why was that?”

See how this question gently seeks out more information? Instead of prompting for a binary yes/no question, this wording prompts the user to mentally rank the levels in order of confusion and pick one. It can also open up great follow-up questions such as “Which was easiest to understand?” or “Besides the most confusing, what other levels also confused you?” You can see that this is a fruitful cycle of communication with the tester.

Let's look at another interview question that needs revision.

Original question: “What improvements could we make in the game?”

This is a very unfair question to ask a playtester. A tester is probably not a game designer and may have no idea what would improve a game. It's also very open-ended and begs for the response “Nothing.” If the tester does offer up improvements, either in response to such a question or impromptu, you need to listen! Keep in mind that testers have no clue what is going on in the game internally, and they may say things that are totally unrealistic, but that's fine. You want to hear what they are thinking. When listening to this kind of feedback, it's important for the team not to think about whether it is possible but instead to think about whether it would help the game and

whether there is something else that could address the essence of the improvement.

Revised question: “We are thinking of these three improvements to the portion of the game you played. Which do you prefer?”

This question frames the answers you get the way you want them. You already know what is possible, and you know the kinds of answers you are looking for at this phase. This wording encourages the tester to go beyond thinking in a binary yes/no way and to think of simple options that are much easier to digest because they are limited.

One good way to handle questioning is to make contrasts (for example, “What was the best aspect of this character class, and what was the worst?”). Even great games have best and worst aspects. A question framed as a contrast invites the playtester to give more honest feedback.

Consider these final notes on proctoring playtests:

- Thank the testers and be nice. They are doing you a service, not the other way around.
- Never disagree with a tester unless they are factually wrong. Any feedback they give is good feedback. If a tester says something wild and completely unrealistic, a simple response like “Thank you for that feedback. I’ll write it down” helps you move past it and get to more useful information.
- Call back good testers. Most games take more than a single session to complete, and having experienced testers is often not only wanted but really needed.
- Don’t call back bad testers. If a tester gives little or no feedback or clearly has an agenda of changing the game into something else, then there is little more to learn from that tester. While such a person is playtesting, be polite but note that they will not be needed for further tests. You may even need to go as far as to throw out the data generated from a bad tester. If you have a tester who is purposely playing the game in a way that was not intended or is clearly uninterested in learning the game, it’s not likely that they are going to be a good

representative of real players. Almost all data from testers is good data, but in extreme cases it's not.

- Get a variety of testers from within or near your target audience. Find a variety of ages as well as gaming and social backgrounds. It's also a good idea to get a few testers who are well outside your target market, just to get a fresh perspective. Characteristics of testers should be noted so their feedback can be judged accordingly. If a parent brings in a young tester, that parent becomes a great possible outsider tester.
- Don't expect gold from any single tester. Everyone has different opinions, desires, and play styles in a game. It's more likely that as the game matures and tests are conducted, you will receive less actionable feedback and more repetition of the same themes.
- Look for repetition. If one tester has a piece of feedback, that is valuable. If many or most have the same feedback, then that is critical feedback that should not be ignored.

After you run user testing and review of all the notes, it's important to bring your team into the process and let them know how it went. In particular, the design team needs to understand what happened during tests and why changes are being made as a result of testing. If possible, show designers raw footage of testers playing the game. It is humbling but very informative to see a new player struggling through something you have spent a lot of time and effort crafting.

Beta/Postlaunch Telemetry Testing

Telemetry testing is, as the word implies, done at a distance. While the full scope of telemetry testing is massive and beyond what this book can cover, it is worth noting that most video games use some form of it today. Even smaller games made by individual developers can do some light forms of telemetry data collection. Every mobile platform, console, and PC game has the capability to do at least a little of it.

Data Hooks

Doing telemetry testing involves inserting data hooks into a game and then sending information to the team from everywhere the game is being played.

A data hook can be as simple as a single line of code that records a variable to be sent. For example, each time the player character dies, you can add that to a variable “total PC deaths” and send it off. The team can then capture this data and tally it in a database. Some studios have entire teams whose only function is to evaluate these hooks and use them to determine how to make the game better. On smaller teams, this task often falls to the system designers, who are, after all, the spreadsheet people. On any team of any size, system designers are heavily involved in the design of the hooks and the analysis of the data.

The following are some examples of a nearly endless possible set of data hooks you might want to implement in a game:

- Player time in game
- Player session time
- Time in main menu
- Time per level
- Amount of in-game currency earned per level
- Number of times the PC died/failed
- Usage of a character class, specific vehicle, or other item
- Game completed
- Amount of XP or currency earned using different methods
- Number of players in a multiplayer game
- Best and worst scores in a competitive match
- User screen size
- Operating system
- Geographic location
- Default language
- How much money the player has spent in game

For each of these hooks, there needs to be a snippet of code that collects the data as described. For each hook, the data needs to be fully quantified. To record player session time, for example, you need to define the start and end of a session. To define telemetry hooks, you use the same methods you use when asking data questions, as described in [Chapter 3, “Asking Questions.”](#)

When all the hooks are encoded in the game, and the game has been sent out to testers (or even customers), the code sends back regular updates on all the hooks. As you might expect, this process can generate a very large amount of data. While it can occasionally be useful to drill down to a single user or session, it usually is better to try to look at the data as a whole for patterns that you can take action on to improve the game.

Telemetry Considerations

Telemetry data is not a magic bullet or cure-all. It can provide massive amounts of information and can yield some otherwise invisible insights, but real humans are required to interpret it properly. When making telemetry hooks, it’s tempting to put in a hook for everything you can think of, but it is important to avoid creating noise and not useful information. Therefore, when designing hooks, you should think of the kinds of information that will actually have an effect on the way the game is being made.

While your data sample is low, watch for outliers. Even when the sample gets large, you should keep a keen eye out for discrepancies in data. This ties back to the section on analyzing data. Don’t simply rely on averages or the minimum and maximum values you find in the data. Those types of data provide great starting points but can easily distort the results. The following two examples from the game industry illustrate why this is so important.

Telemetry Example 1: Time in Tutorial

A simple mobile puzzle game had a tutorial. The team found through playtesting that players got bored with tutorials after about 5 minutes, so they designed their tutorial to be well short of this, at 4 minutes. They put in a data hook to calculate the amount of time it took for players to complete the tutorial and launched the game into a small set of beta tests. After the game ran for a day, the team evaluated the time players were spending in

the tutorial, and the result was a shock. Their average time was over 2 hours! What could have gone so horribly wrong? If you think back to Chapter 15, “[Analyzing Game Data](#),” the answer is pretty obvious: There were some huge outliers. By digging more deeply into the game and the hooks that already existed, the team was able to discover that some players started the tutorial and then set their phone aside for the night and went to bed; they came back and finished the tutorial some time the next day. In response, the team simply discounted any time over 20 minutes and used other methods of measurement, such as median, to get a more accurate picture of what the data meant. When looking at the data this new way, the team found that the tutorial was taking most players roughly the correct amount of time to complete. The team found that many users who spent longer than this on the tutorial did not speak the language. This informed the team that they needed to take the primary language of the player into consideration as well.

Telemetry Example 2: Abandoned Levels

In another game, a large team of designers was each assigned to a set of levels. Each designer was primarily responsible for making sure their level was great. When the game shipped, it had a hook in it that tracked when players started the level, how long it took them to complete, and whether they quit the level. After the game had been out a few weeks, the levels were analyzed to determine which of them were the most often abandoned. It turned out that all of the most abandoned levels belonged to the same designer. Just looking at the surface of this data, it seemed clear that the designer was not very good at his job. However, the team knew that this was the best designer on the team. He was brought on very late in the project and was purposely given all the most broken and unbalanced levels to salvage as best he could in the very limited time that was left before the game shipped. Again, it was very important to look past the raw data and get deeper into the meaning of the numbers before drawing a conclusion.

Note

Telemetry is an amazing and useful tool for evaluating games with large numbers of testers in an unbiased way, but it is a tool that is

only as good as the person using it. Therefore, it is a good idea to plan for and use telemetry in every game possible but not to put too much faith into every result that comes from the data.

Solving Problems

Making games involves solving a lot of problems. Coming up with initial ideas can happen quickly. Implementing ideas also becomes fairly fast with experience. However, but the process of solving problems with ideas and implementation is a slow one—and it always will be. [Figure 17.1](#) provides a rough breakdown of the time spent on making a game.

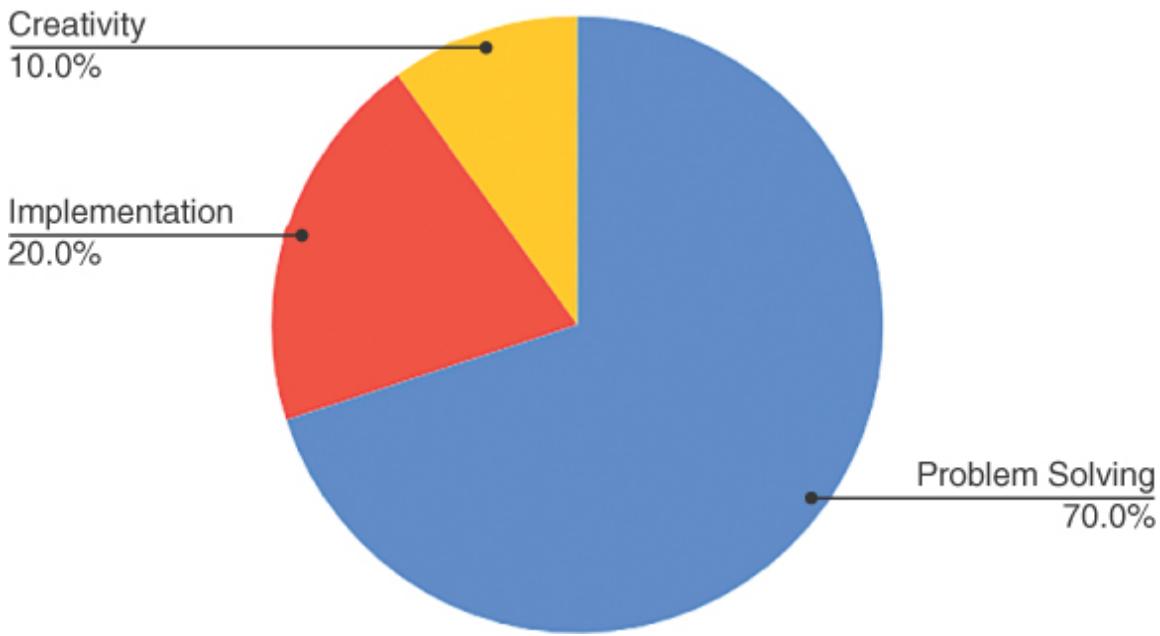


Figure 17.1 Designer time spent making a game

To solve problems in a highly technical system, such as a game, you follow many of the same steps you use to create a game system in the first place but in a more condensed structure. A series of steps go into solving problems. What you do in each step depends on the problem, but the steps remain relatively the same in any case:

1. Identify the problem.
2. Eliminate variables.
3. Come up with solutions.
4. Communicate with the team.
5. Prototype and test.
6. Document the changes.

Identify the Problem

This is the first and arguably the most important step in the process of solving a problem. It's not as obvious as it sounds, though. When doing this step, there is a big temptation to identify symptoms, but what you really need to do is identify the problem. For example, the initial problem statement might be "How do we keep players beyond Level 3?" This is not actually the problem but a symptom. There is no switch to adjust to make players stay longer. When identifying a problem, it's important to identify aspects that are under your control. If, for example, players are quitting early because of low batteries in their cell phone, this is not something you can fix. You need to dig deeper to find the problem that is causing the perceived symptom. In the cell phone example, it might be that your game is taxing the hardware too much, causing the battery to drain. This is something that is in your control and can yield a good problem statement. Players are likely quitting on Level 3 because something is poorly designed in this level. A better problem statement would be "Investigate Level 3 to discover why players' phone batteries are consistently dying at this level."

Eliminate Variables

Once the problem has been identified, the next step is to eliminate variables. What variables might be causing the problem? Which ones are under your control? The best way to find a problem variable is to isolate it and test it individually.

In the cell phone game example, you would want to look at what processes are running during the game and separate them out and test them individually. If the levels are empty, do they still drain the battery? If the AI and game objects are in an empty box level, devoid of geometry, do they drain the battery? If one does and the other does not, then you have gotten closer to finding the actual problem. Continue to break down large chunks of suspect areas in the game into smaller ones until you can identify the exact location of the problem.

Come Up with Solutions

Coming up with solutions is very much like coming up with new ideas, and you can use the same methods listed in [Chapter 8, “Coming Up with Ideas,”](#) to generate solutions. The only real difference is that the problem, instead of a conceptual idea, defines your brainstorming session.

Communicate with the Team

At this later point in the game-making process, it’s important that the affected team know that a problem exists and that they may need to be part of the solution. While it is not always possible to get universal buy-in for every solution, it should be the goal to get as much team agreement as possible. At this point, the team should be relying on the game hierarchy, discussed in [Chapter 13, “Range Balancing, Data Fulcrums, and Hierarchical Design.”](#) If more than one system can be changed to solve the problem, then it is important to change the system as far down on the hierarchy as possible and account for further trickle-down effects.

Prototype and Test

Before injecting a large change into a complex project like a game, you should test the solution in isolation, if possible. If needed, make a test level, character, weapon, vehicle, or whatever is being changed and test it in full isolation from the game. This process should work in a very similar fashion to the way new features are rolled out as discussed early in this chapter.

Document the Changes

After a suitable solution is found, be sure the changes are well documented. This documentation helps the team understand what change needed to happen and can serve as reference in the future if a similar problem crops up.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further fine-tuning data, testing, and revising data.

- Develop and run some playtests on either a simple game you have made or a very simple game that is new to your testers (such as backgammon or checkers). Observe the actions of your testers and note how many of the indications listed in the section “[General Game Balance](#)” you observe. Do the testers do a lot of head scratching? Is there change in lead? Do the players disagree on optimal strategy?

Chapter 18

Systems Communication and Psychology

Unlike books, movies, songs, paintings, and other forms of artistic expression, games have communication going in multiple directions. When a movie plays on a screen, it unfolds the same way every time, regardless of what the audience thinks or does about it. This makes games different from other art forms in ways that are important to acknowledge. The player has agency to change the outcome of the game, and the game has influence on how the player reacts; together, they form a communication circle.

Games are, in some ways, similar to human communications. Interpersonal communication is a reciprocal cycle between two or more people. Many studies have been done on this subject, and some of them are directly applicable to games. This chapter highlights some of the concepts from human communications that most directly apply to game making, especially game systems.

Games as Conversations

You can think of a single-player game played between a human and a machine as a conversation. You can similarly think of a multiplayer game as a conversation between multiple people. In both these analogies, the game is the language. For example, it is possible for two people who do not speak the same language to play a game of chess. They can play a complete game, get to know each other's gaming personality, and conclude the game all without needing to know a common language—beyond the rules of the game.

In single-player games, the rules to start the game open the dialog with the player. The designers of a game make a set of rules they want the player to understand and use to communicate their thoughts to the game or to another player (see [Figure 18.1](#)).

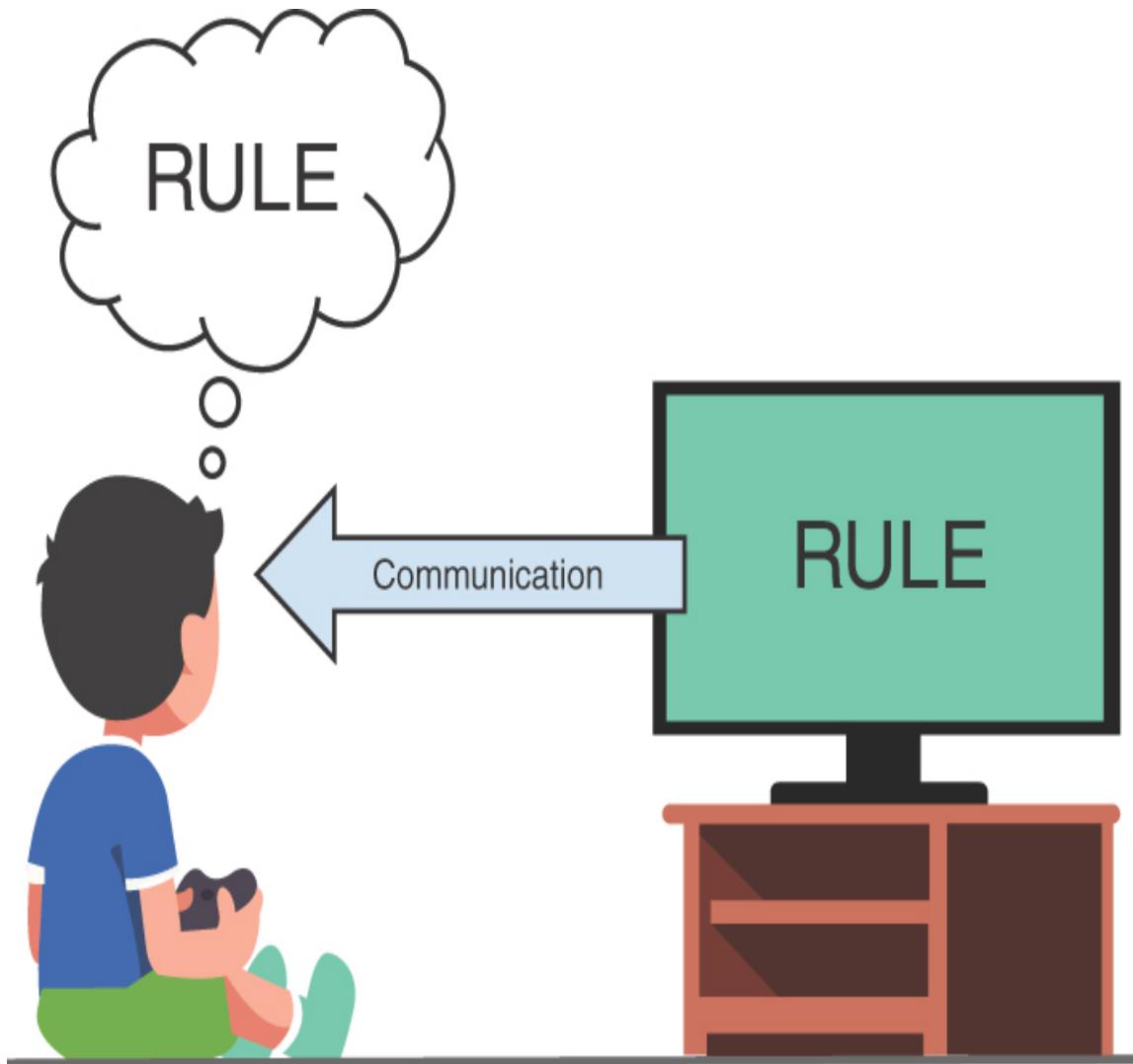


Figure 18.1 Player learning rules

[Figure 18.2](#) illustrates a fairly standard video game communication loop. The game includes images and sounds that the player interprets. The player then communicates back to the game what they want to do, based on their interpretation of what they experience in the game. The game interprets the information from the player according to the game rules and sends the player feedback through audio and visuals on the screen. The loop repeats constantly throughout the game, much like a conversation between two people.

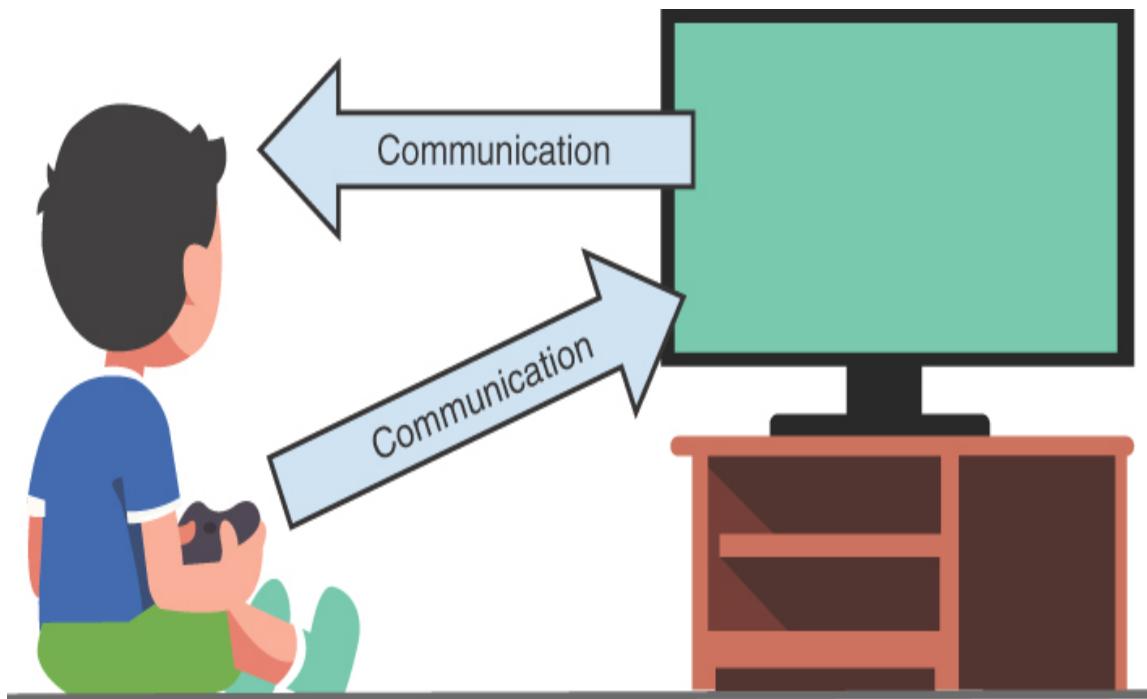


Figure 18.2 Example of a player and a game communication loop

Word Meanings

Language is inherently imperfect, and it is important to keep this in mind when writing rules for a game. The high-level goal of language is to convey an idea from the mind of one person to the mind of another, but language simply can't be precise enough to always do this effectively. For example, consider the following sentence:

The dog likes eating meat.

It seems like a straightforward sentence, right? Do you imagine a dog eating meat in your head? Does what you imagine look like any of the images shown in [Figure 18.3](#)?

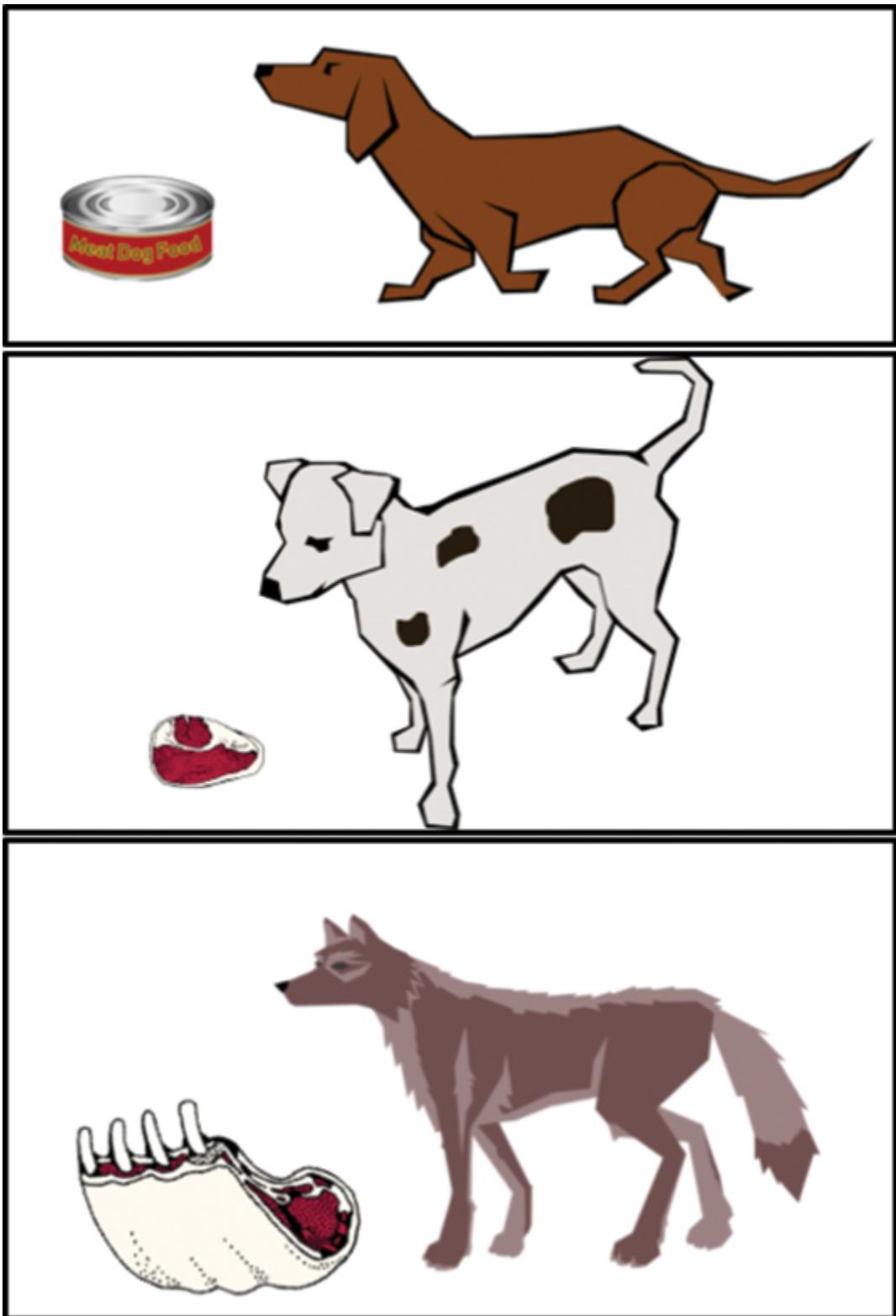


Figure 18.3 The dog likes eating meat

All of these images fit the description, but they are completely different from each other. Which one of them is most like what the author of the sentence intended? We can't know.

To complicate the imprecise language problem, pronouns—which serve as nice shortcuts in casual conversation—can cause a great deal of misunderstanding in written works, especially in rules. Consider the following rule:

When the player character hits the enemy with the Staff of Purity, he becomes immune to poison.

Who becomes immune: the player character or the enemy? It is impossible to tell because of the way this rule is written: We don't know who the pronoun *he* refers to.

Dealing with the meanings of words is a difficult challenge that all humans face, and game designers need to be particularly careful with their wording because of the importance of having clear rules in a game. There is also, sadly, no quick trick to solve this problem. There are methods you can practice to get better at more clearly getting the ideas from your head into the head of the audience. You may have heard the saying “Show, don’t tell.” This is a tactic you can use in games—and you also have a number of other tools at your disposal. The following are some of these tools, listed in order from most effective way to communicate concepts to least:

- **Interact:** Show the player what to do. Have them do it and correct them if they do it wrong. Test to make sure the player has a good grasp of what is and is not allowed before letting the player move on. In video games, this function is often called a *gatekeeper*. For example, if a core mechanic of a game is for players to shoot a bow and arrow, you can present a player with a locked door that the player must open by hitting an easy target with an arrow in a low-stress situation before she can move on. Once you understand the concept of gatekeepers, you will start to see this mechanic pop up in video games everywhere.
- **Show:** If it is not possible to explain rules interactively, show the player the rules by using video and audio. You might, for instance,

have a helpful NPC doing the action you want the player to do or use a cut scene that takes control away from the player to show how the player character can do the desired action. Or, as illustrated earlier in this chapter with the meat-loving dog, you can show a simple picture—such as a map or a diagram—that illuminates what you want the user to do.

- **Tell:** If it is not possible to use any type of video means to instruct the player, you can use clear audio instructions with good audio feedback.
- **Write:** Players are generally less likely to absorb directions that are written than directions presented using the above-mentioned methods, but if it is not possible to interact with, show, or tell the player how a particular rule works, you can provide written instructions. It may be beneficial to add written instructions along with using the methods listed above. A user might, for example, want to refer to written information to quickly get a refresher on the rules.
- **Hint:** If it is not possible to give the player written instructions, you can provide hints such as iconography to help players understand what is expected. You could use cracks in a wall, for example, to provide an iconographic hint to players that they should attack those walls to find that it crumbles, revealing a secret door. In some situations, you might want to use more blatant hints, such as an arrow painted on a wall, telling the player where to turn.

Note

For many game mechanics, hinting is actually the preferred method of communication because part of the fun for a player is figuring out what to do. The design team should purposely think through where to use hints. What parts of the game do you want to require the player to figure out, and what parts of the game do you want the player to intuitively understand? Old video games made players figure out basically everything on their own. This method is now considered primitive and frustrating for modern audiences, and game designers pick and choose which mechanics belong at various levels of communication.

- **Guess:** The most frustrating—and least advisable—way to have players figure out the rules is to have them guess and punish them if they guess incorrectly. In general, you should avoid this method. If you find that testers are guessing about rules, consider using one of the methods listed above to make sure they don't need to guess.

Noise

From the perspective of communication theory, noise is not just background sounds. Noise is anything that can impede communication or lead to misunderstandings (see [Figure 18.4](#)). In games, this can be many things, including game controllers, understanding of the rules, the physical setup of the game, literal audio noise, visual noise on a computer screen, or anything else that can cause communication to be interrupted. Noise is inevitable any time there is communication, and a designer needs to acknowledge that and minimize it as much as possible.

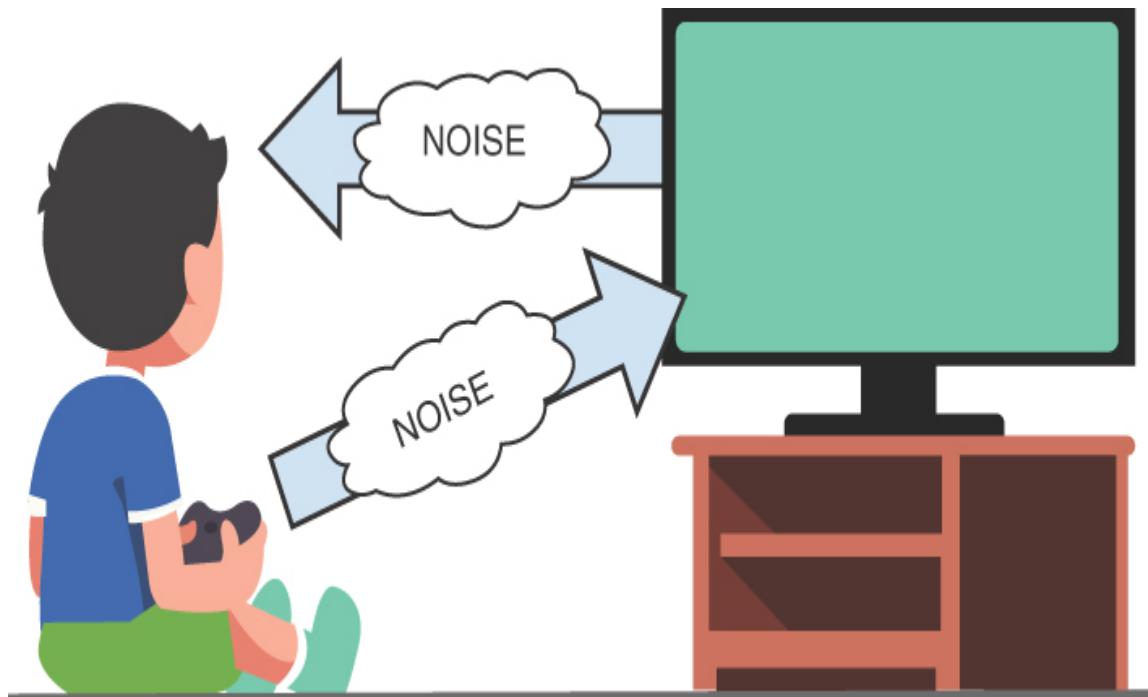


Figure 18.4 Noise in the communication loop

When you write or code rules for a game, you know what you want the player to experience and how you want them to communicate, but the players don't have that knowledge, so the conversation starts out quite lopsided. Whether you teach rules in a book, in a video, or in an interactive tutorial, you must battle noise to get your idea from your head, into the game, communicated to the player, and into the brain of the player.

Even if you have the most well-written set of game rules ever made, if the player experiences noise while learning the rules, they will have a diminished experience. Noise can be injected into the system at many points on the side of the game or on the side of the players. Some of the most common points of noise are as follows:

- **Unclear rules:** If the rules are poorly written or communicated, they are bad, regardless of how well they were intended. You have likely experienced a moment playing a game where you found out about a rule of the game that you had previously not known. Once you took in that rule, you probably thought about how that rule would have drastically changed decisions you had made up to that point. Such misunderstandings cause a large amount of frustration in players. Testing is the best way to find unclear or ambiguous rules. Ask your testers what the rules of the game are. Challenge them (nicely) with fringe case rulings and see if they know how the game should play. Ask your testers about what circumstances in the game would break the rules and see if their answers are right. The design team is responsible for ensuring that all of the rules they come up with are clearly communicated.
- **Technology limitations:** Even with today's powerful computers, there are limitations to what you can show onscreen. You might have a play area that is too big to fit on a screen. You might not be able to do what you want to do with a controller. You might not have the ability to produce enough distinct sounds to properly signal to players. This is another place where designers need to take responsibility. No system will ever be perfect, and it is a designer's job to acknowledge the limitations of hardware and build communication with the player out of what is available.

- **Visual limitations:** Even if the screen on which you are displaying a game is capable of displaying every aspect of the game you want, it might not be digestible by the players. Think of very complex MMOs that have more than 50% of the screen filled with various UI elements. To an audience that is fluent in the language of the MMO, this is acceptable, but to new players, it looks like chaos. This can be a difficult problem to solve, and we will talk more about it in the next section, “[Reciprocity](#).”
- **Controller limitations:** Humans have arms, legs, fingers, eyes, and a mouth to communicate in person. We have all spent our entire lives learning to use these tools as subtle or blatant forms of communication. However, when playing a video game, players can’t use any of those tools. They must, instead, use a game controller. Modern controllers have a multitude of buttons and joysticks with various degrees of pressure and motion sensitivity. A controller provides a special form of communication pathway for the player to use with the computer. This can be very intimidating to new players because they have not spent their entire life learning to use a controller the way a game designer wants them to. Over the past 30 or so years, designers have come up with many conventions for using controllers for games. For the most part, this is a good thing as it offers a shortcut to reduce noise related to controllers. This is also one place where creativity may not always be the best idea. If you are finding that players are frustrated with the controllers as they play your game, you should look at what other games are doing with controllers. Do you think you need to have a unique interaction? Would it be better to go with convention for the sake of noise reduction? While it is certainly possible that your game has a mechanic that really does require the player to use the controller in an unorthodox way, keep in mind that this novelty will inevitably create noise when players are learning. Take extra steps to ensure that any controller usage has been thoroughly researched and tested against what your target audience expects.

For example, if you were making an FPS that required a keyboard and mouse, you could research the traditions of other FPS games that also use a keyboard and mouse. If you don’t currently use WASD keys for

player movement, why not? There may be a good reason, but is it worth fighting expectations and introducing noise?

While these are some of the most common forms of noise and miscommunication in games, they certainly are not the only ones. The key to reducing noise is to identify it through testing and questioning of testers. Once a source of noise is found, the design team needs to figure out how to minimize it.

Reciprocity

There is a communication theory called *reciprocity* that is part of the greater idea *social penetration*. The very high-level explanation of the theory of reciprocity is that people start more reserved when they meet someone new, revealing very little about themselves at first. As people get to know and trust each other, slowly, layer by layer, they expose more of themselves to each other. Only with complete trust in each other do they finally disclose their innermost thoughts and feelings. What does this have to do with games? Games and players are building a sort of relationship with each other. As discussed earlier in this chapter, there is a great amount of communication between them. In the relationship between humans and games, the game asks the player to perform tasks of ever increasing difficulty and in return gives the player rewards both virtually in the game and in entertainment value derived from the game. Players ask the game for instructions on how to perform the tasks and, in return, performs the needed tasks. This is reciprocity. The following sections provide a few examples of problems with reciprocity.

Overstepping Bounds

A player who is first beginning to play a game tends to be reserved in their willingness to commit to the game. The player wants to get some positive feedback from the game before being willing to commit more time and effort to the relationship. If the game asks too much of the player too early, the player is likely to see the game as overstepping its bounds; this is a form of rudeness toward the player that is off-putting.

Say that two people meet for the first time. The first one says, “Nice weather.” The second says, “I was in a house fire as a child. I still wake up crying about it every night.” Whoa. You may have had a conversation in your life like this and know how awkward it feels. The second person’s feelings and experiences are not invalid, but they have overstepped the bounds of societal norms by exposing so much of their inner feelings upon first meeting. This is too much to ask of someone you do not know.

Now say that a player picks up a new video game and boots it up on their computer for the first time. The game has large text onscreen that says “Read and memorize the 63-page user manual before beginning the game.” Whoa again! The player does not know if they are going to like this game and if they are willing to invest time memorizing 63 pages of a user manual to find out. The game has overstepped the bounds of what it can ask from a player upon first meeting. It is completely possible that over the course of playing the game, the player might happily read 63 or even more pages of instructions—if they know that their time investment is worth it and that the game is willing to reciprocate the player’s effort with rewards in entertainment.

Shallow Relationship

If a game fails to gradually ask more and then give more to the player in return, the player may feel as though the relationship is shallow.

For example, say that two old men have known each other for 50 years. The wife of the first old man asks him what the second old man’s favorite dinner is. The first old man says, “I don’t know. We never really talk about anything except the weather.” You would think that people who know each other for such a long time would pick up a bit of information about each other. When a relationship has been going for a long time, it is expected to advance at least somewhat. If it does not, it is considered a very shallow, unfulfilling relationship.

A player has been playing an MMO for years and has invested many hours playing and studying how the game works. The player logs in for the newly launched quest line and goes through the needed steps to launch the quest. At this point, the quest giver says to the player, “Hail adventurer! We are

overrun by Level 1 goblins. Please go kill two Level 1 goblins to complete the quest.” What a letdown! All that hype, all that time invested, all that work and planning...and the game asks the player to do something trivial. While the player has held up their end of the bargain, the game has shown itself to be shallow in the relationship.

Right Balance

Now let’s look at a couple examples that strike the right balance in terms of reciprocity. First, say that two strangers meet at a party and spend the evening chatting about movies. They find that they like many of the same movies, so they go see new movies together. As time goes by, they start to have deeper conversations about relationships, family, work, and social life. After many happy events together, they slowly come to trust each other enough to reveal inner secrets, wants, and fears that they would not trust anyone else to know.

Now say that a player picks up a game, and the game immediately welcomes her. The game then lays out a very simple task, such as “Pick up that sword on the ground by pressing A.” When the player does this, she is rewarded with a nice sound and the message “You now have a sword!” As the player progresses, the game places slightly more difficult challenges in front of her. Because the first reward felt good, she is now willing to invest a little more time and effort into completing this more difficult challenge, trusting that the game will reward her fairly for the effort. After many hours of play, with ever-increasing challenges, the player is faced with a “boss level” that is significantly difficult. She will fail often and may have to go away to better equip and come back. She is ready and willing to take on this challenge because the game has established a deep level of trust with her that the reward for completing the challenge will be worth it.

As you can see from these examples, by knowing about reciprocity and planning it into game systems as you build a game, you can better develop the kind of relationship you want your games to have with players. By purposely taking steps to establish trust with a player in the beginning and by handing out rewards in a methodical way, you can create more willingness in players to commit themselves to your grand vision of the game, regardless of how deep or sophisticated you make it.

Reward Expectations

Players' reactions to rewards can differ dramatically, depending on the expectations they have for the rewards. As discussed in the preceding section, if players keep giving more of themselves and the game does not respond, the players may become disillusioned with the game. There are also other ways to disappoint players. One is setting misleading expectations of reward. In general, the rule should be to underpromise to players and overdeliver. Let's look at an example.

Say that a game tells a player that if they complete the next quest, they will receive a special prize that is worth up to 100 gold pieces. The player completes the quest and claims the prize, which turns out to be 50 gold pieces. The player is now disappointed because when they heard "Blah blah blah...100 gold pieces reward," they tuned out everything except "100 gold pieces" and immediately set their expectations for that. Anything less is a disappointment, so even though the player has been rewarded, they are dissatisfied with the result and possibly also dissatisfied with the output of effort required to get the reward.

Now let's look at the same circumstances but rearrange the example just a bit to see a very different effect. In this revised version, the game tells the player that if they complete the quest, they will receive a prize worth at least 40 gold pieces. They complete the quest and claim the prize, which ends up being 50 gold pieces. Even though this is exactly the same reward as in the first example, in this case the player is very happy instead of disappointed. She was already happy enough with the reward that she was willing to take the quest, and she would have been happy with a reward of 40 gold pieces. When you give players an extra reward, they are happily surprised on top of already being happy with the original deal. They are likely to take on even more tasks in the game, thinking there could be more potential rewards hidden throughout.

If you think back through your own game-playing experience, I bet you could think of more than one time that a game overpromised and underdelivered rewards. This might happen, for example, with a hyped named prize object like The Amazing, Flaming Sword of the Ancient Gods!

that turns out to be a sword that is basically like every other sword, but it's orange.

To avoid overpromising and underdelivering rewards, purposely make a plan to slightly underpromise and overdeliver whenever possible. Also keep a close eye out for this during the testing phase. The look on the face of a tester who has just been rewarded can be revealing. Players are often at their most emotive when they are disappointed.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the concepts of player communication theory.

- For a game that you are working on, write a list of possible noise barriers that the player might face. Then list mitigation steps you are planning to take to ensure that these factors do not disrupt the player's experience.
- Make a purposeful plan for reciprocity in a game. Determine how deep each mechanic is and what you expect from the player before you expose that mechanic to them.
- Find an example of a game that you feel oversteps bounds and an example of a game that provides a shallow relationship. How could each of these games be improved to make a well-balanced relationship with the player?

Chapter 19

Probability

Not everything is predictable in the world or in games. However, it is possible to understand unpredictability. By examining basic probability, you will understand how existing games use random chance to manipulate players. Further, you can learn techniques that allow you as a system designer to allow random, unpredictable events while maintaining control over the flow of the game. But what is random?

Random is more difficult to define than you might imagine. There are entire fields of study dedicated to randomness and probability. This book covers many of the most basic concepts of probability, but it stops well short of providing full explanations; you can find books and other resources dedicated to those subjects.

For the purposes of this chapter, I define *random event* as follows:

Any event for which the player can't accurately predict the outcome with the information they have at the time.

Randomness is not binary. Events don't have to be either all random or not random. For example, think about golf. If a player is about to drive the golf ball toward the green, it is impossible to know the exact position where the ball will come to rest. However, this does not make the event totally random. You could estimate an approximate location where the ball will likely land. Based on what you know of the player, the shot, and the conditions in the environment, you might be able to make a somewhat accurate prediction, but you can never be completely accurate.

Basic Probability

This section covers the absolute basics of probability, including how it is written and how it is calculated.

Probability Notation

Probability notation is surprisingly easy, and you are likely already familiar with it even if you don't know that you are. A probability calculation has only two variables: the total number of outcomes and the number of outcomes considered success. To break this down with a simple example, we can look at flipping a coin. Say that you want to know the probability that a coin flip will be heads. There are two sides of the coin: one heads and one tails. Of the two sides, only one is heads. This means the coin has heads on one out of two possible sides; therefore the chance, or probability, that the coin will land on heads is 1 out of 2, or 1/2. Based on the fact that the probability of an event equals the number of successful options out of number of total options, we can create the following formula:

$$\text{Probability} = \text{Successes} / \text{Total options}$$

$$P = S/T$$

In the case of a coin toss landing on heads, we can use this notation:

1 chance of heads out of 2 possible outcomes

1/2 = chance of heads

We use this final notation throughout this chapter to further explain probability.

Real and virtual dice are often used to determine outcomes in games. Special notation is used for the probability of dice rolls. With dice, we use the notation XDN to express the count of the number of options on a die (N) and the number of dice that are being rolled (X). For example, the notation for rolling a single six-sided die is 1D6. The notation for rolling two six-sided dice and adding them together is 2D6. The notation for rolling four two-sided dice is 4D2. You could also use this notation to represent coins, which are, in effect, two-sided dice.

Calculating One-Dimensional Even-Distribution Probability

Spreadsheets are fantastic tools for doing probability calculations, but they need to be set up correctly. In this section, we begin to look at calculating probability, and we again use the coin example—but this time we bring it into a spreadsheet. This section shows how to start building a framework for doing complex calculations in a spreadsheet.

Note

As you read this chapter, I strongly recommend that you follow along in a spreadsheet and build this framework yourself.

To get started, you need to enter a few pieces of data. You know the type of die you are using, 1D2, and you know that this die has a heads side and a tails side. [Figure 19.1](#) shows how you would lay this out in a spreadsheet.

	A	B	C
1	1D2	Event	Count
2		Heads	1
3		Tails	1

Figure 19.1 1D2 probability

As you can see, you want to include in the spreadsheet the label 1D2, an Event column for outcomes you want to calculate, and a Count column that lists the possible successful outcomes. For this example, you are going to calculate the probability of each event in a simple coin toss—1 head and 1 tail—so we need two more pieces of information immediately. First, to get

the number of successes out of the total possible, you need a total count of all events. Then you need to divide each event count by that total. Your spreadsheet needs to look as shown in [Figure 19.2](#).

	A	B	C	D
1	1D2	Event	Count	Chance
2		Heads	1	50.0%
3		Tails	1	50.0%
4		Total	2	100.0%

Figure 19.2 Labeled probability table

[Figure 19.3](#) shows the same spreadsheet, with all the needed formulas and functions to do the proper calculations.

	A	B	C	D
1	1D2	Event	Count	Chance
2		Heads	1	=C2/\$C\$4
3		Tails	1	=C3/\$C\$4
4		Total	=sum(C2:C3)	=sum(D2:D3)

Figure 19.3 Probability formulas

There are a couple things of note about the spreadsheet in [Figure 19.3](#). First, notice that some of the references are absolute and others are relative. You need both absolute and relative references to make the formula ready to formfill down. When you use relative references pointing toward the count and an absolute reference pointing to the total, as the formula is copied down, it updates properly for all the cells. Second, notice that in cell D4, the percentage chances of all the events have been summed to give a total of 100%. This seems unnecessary, but it is a very good practice to total your percentages with complex calculations, so you can ensure that they indeed add up to 100%. You know intuitively that percentages should add up to 100%, and when you see in a spreadsheet that the percentages do, in fact, add up to 100%, you have confirmation that you have correctly done all the calculations in all the other cells that feed this final check. This is an example of a data canary, which helps you know when you have a problem in your data (refer to [Chapter 15](#), “[Analyzing Game Data](#)”).

Now that you have a very simple probability spreadsheet laid out, it’s time to increase the complexity. Let’s jump up to using a standard six-sided die, or 1D6. For the most part, this is going to be very similar to the 1D2 spreadsheet but longer and with different chances. To get started, you should set up your formulas so that they behave basically the same as those for 1D2 but with a longer Event list. In formula view, your spreadsheet should look as shown in [Figure 19.4](#).

	A	B	C	D
1	1D6	Event	Count	Chance
2		1	1	=C2/\$C\$8
3		2	1	
4		3	1	
5		4	1	
6		5	1	
7		6	1	
8		Total	=sum(C2:C7)	=sum(D2:D7)

Figure 19.4 1D6 table

Cells D3 through D7 have not been filled out in [Figure 19.4](#), but cell D2 has been written in such a way that you can use formfill to drag it down and complete all the rest of the cells without writing a new formula. Once that is done, you should have a spreadsheet that looks as shown in [Figure 19.5](#).

	A	B	C	D
1	1D6 Event	Count	Chance	
2		1	1	16.7%
3		2	1	16.7%
4		3	1	16.7%
5		4	1	16.7%
6		5	1	16.7%
7		6	1	16.7%
8	Total		6	100.0%

Figure 19.5 1D6 table completed

The spreadsheet now shows the basic probability events for a single six-sided die. This information is useful for making a classic board game that relies on getting specific roles with a six-sided die. If you were to graph this probability distribution, it would look as shown in [Figure 19.6](#).

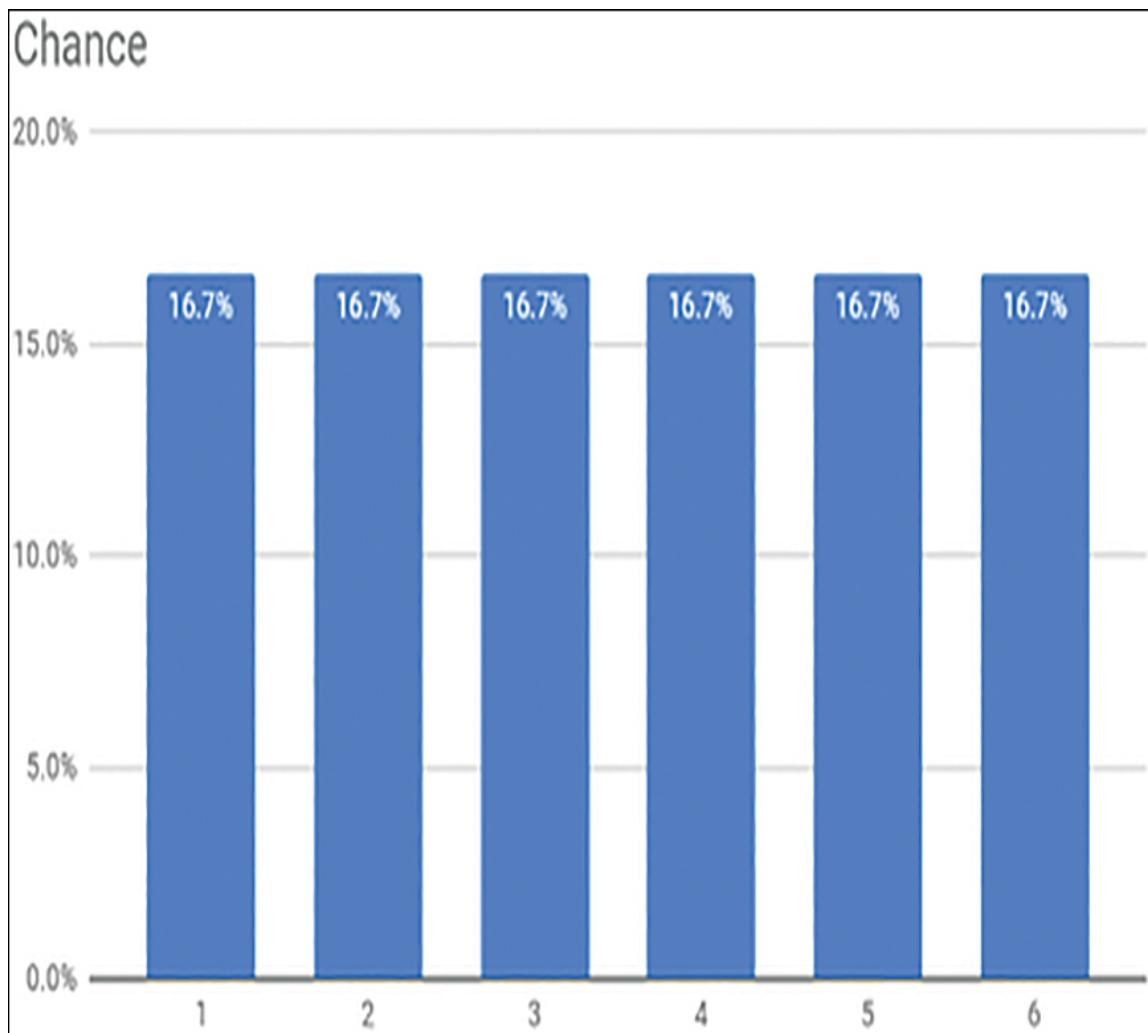


Figure 19.6 Chart of 1D6

This is not a particularly interesting chart, but keep it in mind for comparison with future charts that you make later on. Notice, in particular, that the heights of all the bars are the same, forming a level line across the chart. This is the shape of an even distribution in a chart.

In many games, specific rolls are not always required for success. Often in analog and video games, a player can be successful with a range of rolls. For example, a player may need to roll a 3 or higher to score a point. In such a case, you can't simply look at a single-series chart to find the probability of rolling 3 or higher. To get this probability, you need to add another calculation to the chart to get a cumulative probability, which is

also called a *running total*. To set it up, you need to think about what exactly is going on with the probability. What is the probability of rolling a 6 or better on a six-sided die? This seems like a trick question because there is no number higher than a six. When you start to look at cumulative probability for rolling a number or higher, you can safely calculate the highest number in the list: It is the same probability as rolling that number independently. In this case, the chance of rolling a six or better is equal to the chance of rolling a six. You need to add the new column and calculation shown in [Figure 19.7](#) to include this information.

	A	B	C	D	E
1	1D6	Event	Count	Chance	Or Higher
2			1	=C2/\$C\$8	
3			2	=C3/\$C\$8	
4			3	=C4/\$C\$8	
5			4	=C5/\$C\$8	
6			5	=C6/\$C\$8	
7			6	=C7/\$C\$8	=D7
8		Total	=sum(=sum(D2:D7)	

Figure 19.7 formulas for calculating each event of a 1D6 probability roll

This first cumulative probability calculation is the easy one, and it holds regardless of how many events you are calculating. Next, it's time to start doing the cumulative probability. Fortunately, this is not nearly as difficult as the name makes it seem; you need only a single formula to do it. However, before we can use that formula, we need to take a small detour

and look at *cumulative lists*, also known as *running totals*. Examine [Table 19.1](#), which shows numbers and payments made each week.

Table 19.1 Pay schedule

Week	Payment
1	\$10
2	\$10
3	\$10
4	\$10

Without even using a calculator, you can quickly tell that by week 3, \$30 has been paid. By week 4, the total paid is \$40. What process do you use in your head to do these calculations? You look at the first payment on week 1 and know that it stands alone. For week 2, you know that you already got the week 1 total, and now you need to add the week 2 payment to it, for a total of \$20. On week 3, you add another \$10 to the previous total, for a new total of \$30. In this way, you figure out the running total—which is exactly the same thing you do with the probability. The only differences for our probability example is that you are adding up the percentage chance, rather than dollar amounts, and you are going from the bottom to the top. Now that you know all this, it's time to write the formula for calculating the entire list of cumulative probability.

In cell E6, you need to add your running total, which starts in E7, to your new addition, which in this case is the probability that is calculated in D6. The formula and spreadsheet should look as shown in [Figure 19.8](#)

	A	B	C	D	E
1	1D6	Event	Count	Chance	Or Higher
2			1	=C2/\$C\$8	
3			2	=C3/\$C\$8	
4			3	=C4/\$C\$8	
5			4	=C5/\$C\$8	
6			5	=C6/\$C\$8	=D6+E7
7			6	=C7/\$C\$8	=D7
8		Total	=sum(=sum(D2:D7)	

Figure 19.8 Beginning to calculate 1D6 cumulative probability

By adding the chance in the cell below to the chance of the current event, you can calculate the running total of probability. After calculating the first running total as shown in [Figure 19.8](#), you can formfill the contents of E6 up through the entire range of events to complete the running total. Your spreadsheet should now look as shown in [Figure 19.9](#).

	A	B	C	D	E
1	1D6	Event	Count	Chance	Or Higher
2		1	1	16.7%	100.0%
3		2	1	16.7%	83.3%
4		3	1	16.7%	66.7%
5		4	1	16.7%	50.0%
6		5	1	16.7%	33.3%
7		6	1	16.7%	16.7%
8	Total		6	100.0%	

Figure 19.9 Completed “or higher” cumulative probability spreadsheet

Now you can take a look at the probabilities in the cumulative odds columns (column E) to gain more insights into how the game is going to work. Recall that we began this journey trying to figure out the probability of rolling a 3 or better. Looking at the spreadsheet, you can see that the answer is 29966.7% (which is technically 66.6666% repeating). This makes sense if you look at it from a slightly different perspective as well: You know that to calculate probability, you are really looking for the chances of success compared to the total chances possible. In this case, success means rolling a 3, 4, 5, or 6—which is four possible total options out of 6 total options. This means that the probability calculation here is $4/6$. You can simplify this fraction to $2/3$ —which is the same as 66.7%—so all the numbers check out.

Note

With a single six-sided die roll, any individual number is not a rare occurrence. That is, 16.7% chance of something happening or 82.3% chance of that something not happening is not a rare occurrence.

Often, players get wrapped up in the psychology of there being only a single 1 on a die and think they roll a 1 more than they should. Part of this is due to selective perception, as rolling a 1 (especially if it is a bad result) stands out more than rolling the other numbers. However, the main factor is that rolling a 1 is just not that rare.

Calculating One-Dimensional Uneven-Distribution Probability

This section builds on the principles covered in the previous sections and looks at calculating probability in an uneven distribution. Take a look at the spinner shown in [Figure 19.10](#). It has eight spaces with seven different colors. We can assume that a spin has the same chance of landing on any of the eight spaces.

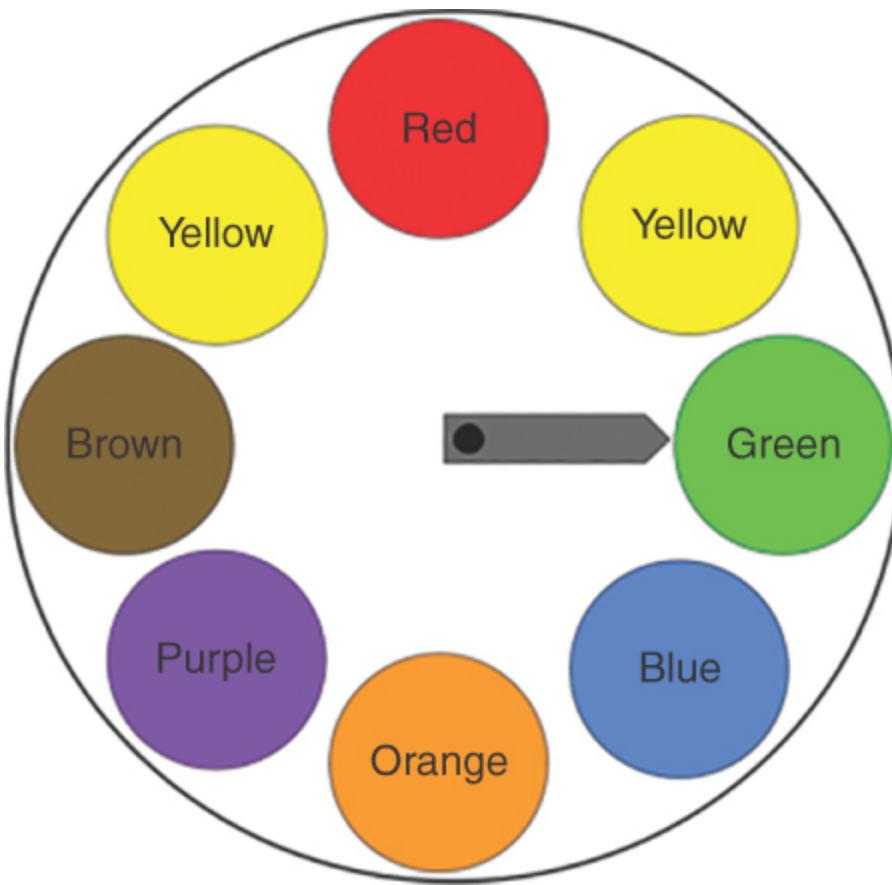


Figure 19.10 Spinner

It might seem like the calculations in this case will be complex, but they are exactly the same as for the 1D6 example. The only differences between this spinner example and the six-sided die example is that there are more possible outcomes with the spinner, one event has higher probability than the others, and it is not possible to do cumulative calculations. The fastest way to do the new calculation in a spreadsheet is to duplicate the 1D6 sheet and make the minor modifications needed. The spreadsheet should look as shown in [Figure 19.11](#).

	A	B	C	D
1	Spin Event	Count	Chance	
2	Red	1	12.5%	
3	Orange	1	12.5%	
4	Brown	1	12.5%	
5	Yellow	2	25.0%	
6	Blue	1	12.5%	
7	Green	1	12.5%	
8	Purple	1	12.5%	
9	Total	8	100.0%	

Figure 19.11 Unequal-distribution spreadsheet

By using this method, you can calculate probability for any given list, no matter how long or short.

You can use the data from [Figure 19.11](#) to create a chart that provides a visualization of the distribution, as shown in [Figure 19.12](#). As you can see in this figure, most of the probability is evenly distributed, but yellow gets a double share. Therefore, the bars in the chart are level except for the yellow bar.

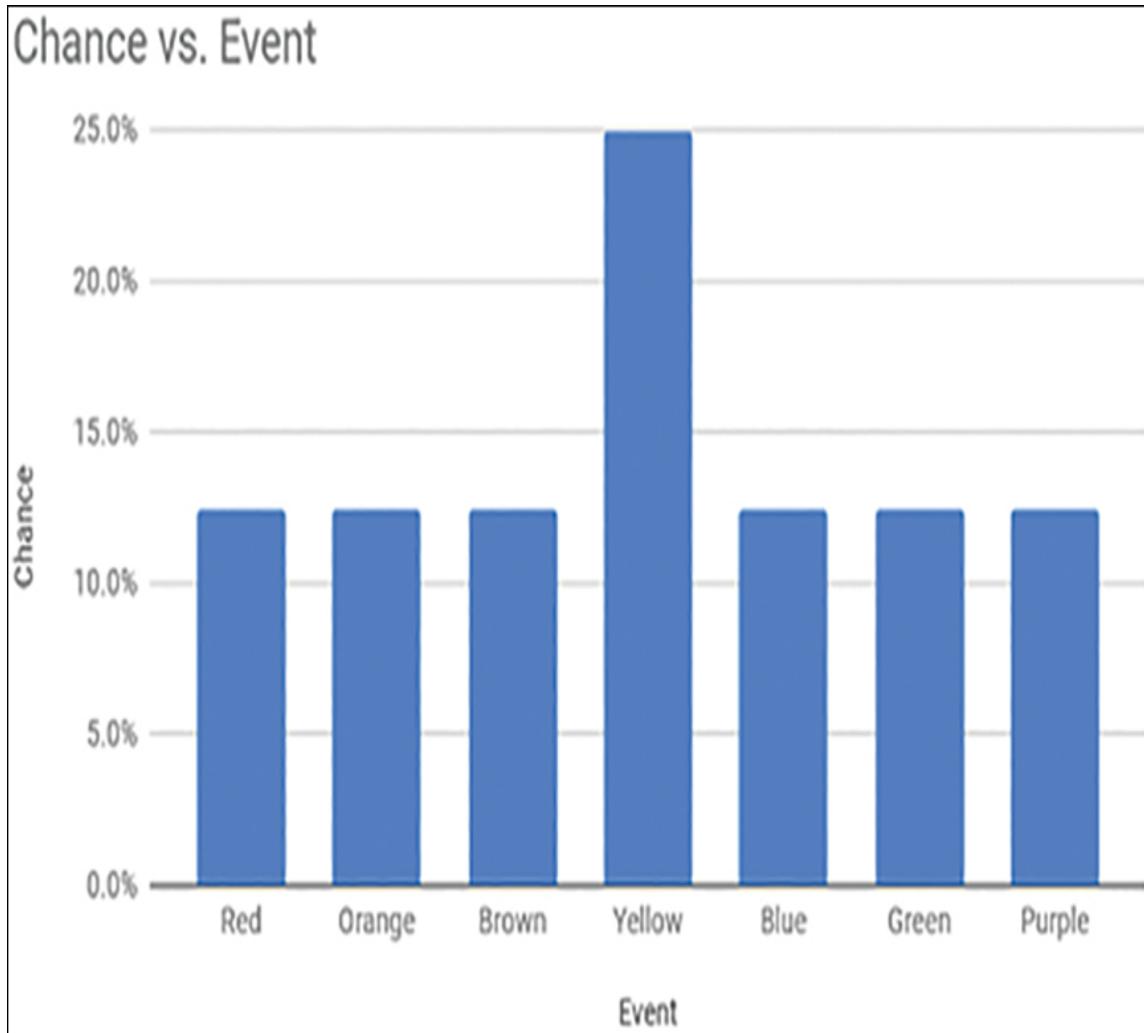


Figure 19.12 Unequal distribution chart

Calculating Compound Probability

One-dimensional probability tables can calculate a great variety of probabilities, but they are limited to a single dimension and a single event. In other words, they are missing some key probability modifiers, such as “and” and “or.” One of the simplest and most common places you encounter compound probability is when you add together the rolls of two dice. When rolling 2D6, there is no longer a single event that determines the outcome. Instead, there are two events that are then added together, each having independent probability. To account for these multiple events, you need a more sophisticated method of determining probability. When

calculating one-dimensional probability, you lay out your events in a line, which is a one-dimensional object. To illustrate this, [Figure 19.13](#) shows a line overlaid on top of the 1D6 events.

	A	B	C	D
1	1D6 Event	Count	Chance	
2		1	1	16.7%
3		2	1	16.7%
4		3	1	16.7%
5		4	1	16.7%
6		5	1	16.7%
7		6	1	16.7%
8	Total	6	100.0%	

Figure 19.13 Linear events

With compound probability, you add another dimension. So instead of having a single line going in only one direction, you need to have two lines running perpendicular to each other to represent all of the possible events.

Note

This is very similar in concept to the difference between linear feet and square feet. When measuring a straight line, such as a rope, you

do the measurement in linear feet. When you measure an area in two dimensions, such as a field or the floor of a room, you use square feet.

When you add the new dimension to the data, it is going to run exactly perpendicular to the first set, starting at the same origin point. The lines the events make will end up looking as shown in [Figure 19.14](#).

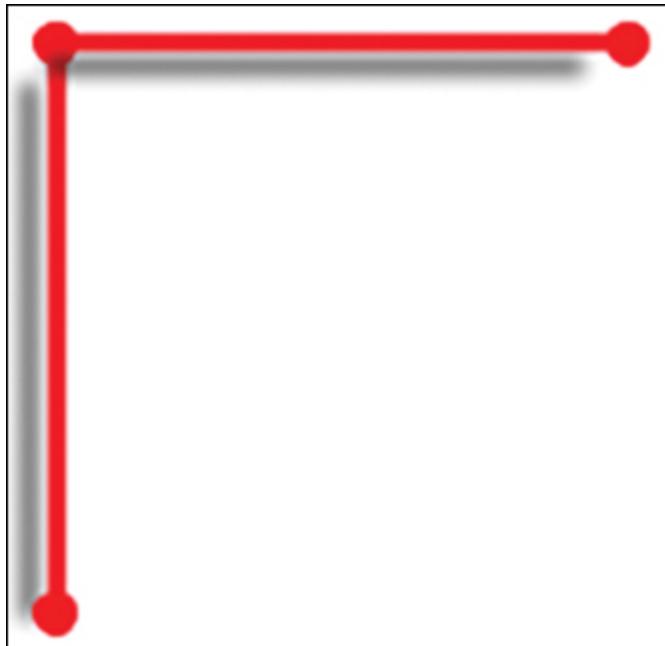


Figure 19.14 Two-dimensional events

To actually start adding data to your spreadsheet, first you can tackle two two-sided dice—which are really coins—where for each coin you can roll either a 1 or a 2. To begin, you can enter all the events of one coin in a single column, as shown in [Figure 19.15](#).

	A
1	2D2
2	1
3	2

Figure 19.15 2D2 table of possible events

To add the second dimension, you need to enter the same numbers from column A into row 1 (see [Figure 19.16](#)). With this addition, you now have a two-dimensional spreadsheet.

	A	B	C
1	2D2	1	2
2	1		
3	2		

Figure 19.16 2D2 two-dimensional spreadsheet

The next step is to fill in the grid. You should first do this manually in your head. Since it is a very small spreadsheet, it should be fairly easy. In B2, you add the 1 from A2 and the 1 from B1 to get 2. You follow a similar procedure for each of the other squares in the grid. The result will look as shown in [Figure 19.17](#).

	A	B	C
1	2D2	1	2
2	1	2	3
3	2	3	4

Figure 19.17 2D2 two-dimensional spreadsheet with probabilities calculated manually

It is easy to see that this simple example has a single 2, two 3s, and a single 4. The next step is to write a scalable formula that will calculate all the cells in the grid, regardless of how many there are. To do this, you need to fully understand mixed reference formulas. In each two-dimensional probability grid, there is a header row and a header column. In this case, the header row is all of row 1, and the header column is all of column A. Regardless of where the formula is copied, it always needs to refer to the header row and column. When locking a reference as absolute, you need to ensure that there is a \$ in front of the portion of the reference that is to be absolute. In this case, you know that row 1 is absolute, so when writing the formula, you need to place a \$ in front of the reference to row 1. The same goes for column A. On the other hand, you need the reference to move up and down row 1 and side to side on column A because the even values change in their respective axes. This means that portions of the reference must not be absolute and can't have a \$ in front. When you use these rules to create the

master formula that will populate the entire grid, you get a formula that looks as shown in [Figure 19.18](#).

	A	B	C
1	2D2	1	2
2	1	=\\$A\$2+B\$1	
3	2		

Figure 19.18 Two-dimensional formula

You can now use this formula to formfill the entire grid. You could further use it to formfill any two-dimensional probability grid of any size. As a result, you get the same numbers you calculated in your head earlier (see [Figure 19.19](#), which looks just like [Figure 19.17](#)).

	A	B	C
1	2D2	1	2
2	1	2	3
3	2	3	4

Figure 19.19 2D2 table completed with a formfilled formula

When the grid is complete, you have a list of all the possible options that can happen in this event, and you can move on to calculating the probability. To do this, you use exactly the same method you used earlier to

determine the linear probability calculations. When you do, you get the result shown in [Figure 19.20](#).

	E	F	G	H
1	2D2	Event	Count	Chance
2			2	
3			3	
4			4	
5		Total		

Figure 19.20 2D2 complete table

Notice that this is the first spreadsheet in this chapter where the data does not start on A1. Remember that you need to keep the 2D2 grid around so that it can be referenced in this new calculation. Also note that 1 is not listed in the Event column—because it is not possible to get a 1 when combining two dice this way. (This is the case any time two dice are added together.)

After you lay out this spreadsheet, you need to populate the Count column. To get the counts manually, you can look at the 2D2 grid and count how many of each number appear in the grid. Although it's possible to do this manual counting with this simple example, it would quickly become very difficult to do the counting this way. Instead, you should set up the spreadsheet to do this counting automatically. When you know how to do this in a simple case like this example, you will be able to use it easily later in more sophisticated computations.

Think for a moment what formula or function you would use to do the counting in this example. Which of the functions that described spreadsheeting in [Chapter 6, “Spreadsheet Functions,”](#) would be most

applicable? When learning how to use spreadsheets, one of the best ways to determine what function to use is to think about what you do manually to get the answer. In this case, you look at the grid in B2:C3, and each time you see a 2, you count up by 1, for a total of 1. Then, you look for 3, count up by 1, and get a total of 2. Finally, for 4, you get the result 1. So what you want a function to do is look in a specific group of cells and count each time it finds what you are looking for. The function that would do this for you is COUNTIF. To set it up to do the counting calculation, you start by entering the function in cell G2, reference the 2D2 grid, and have the function look for the event listed in F2. The function should end up looking as shown in [Figure 19.21](#).

	E	F	G	H
1	2D2	Event	Count	Chance
2		2	=countif(\$B\$2:\$C\$3,F2)	
3		3		
4		4		
5		Total		

Figure 19.21 COUNTIF formula made to look in a probability table

When this formula is formfilled to the rest of the cells, the accurate counts of the events are automatically calculated, as shown in [Figure 19.22](#).

	A	B	C	D	E	F	G	H
1	2D2	1	2	2D2	Event	Count	Chance	
2	1	2	3			2	1	
3	2	3	4			3	2	
4						4	1	
5					Total			

Figure 19.22 2D2 count of events, completed

To fill out the rest of the cells, you use exactly the same methods used to calculate 1D2 to get total and a chance for each event. After all that is done, the sheet should look as shown in [Figure 19.23](#).

	A	B	C	D	E	F	G	H
1	2D2	1	2	2D2	Event	Count	Chance	
2	1	2	3			2	1	25.0%
3	2	3	4			3	2	50.0%
4						4	1	25.0%
5					Total	4	100.0%	

Figure 19.23 2D2 chance complete

The biggest difference between the probability of 2D2 and 1D2 is that not all events have the same chance. Even though both dice have the same number of sides and have the same values on each of those sides, the count for 3 comes out higher than the count for either 2 or 4. This is the natural effect of adding separate random numbers together. The distribution is uneven. To further explore this concept, we can move on to six-sided dice.

You use exactly the same method to calculate 2D6 as you use to calculate 2D2. Again, you should make a copy of your 2D2 sheet and expand it out to accommodate the larger 2D6 grid. Because you already set up all the formulas and functions to be automatic, they automatically expand and formfill not only 2D6 but any two numerical combinations that make a grid. [Figure 19.24](#) shows a completed 2D6 event grid.

	A	B	C	D	E	F	G
1	2D6	1	2	3	4	5	6
2	1	2	3	4	5	6	7
3	2	3	4	5	6	7	8
4	3	4	5	6	7	8	9
5	4	5	6	7	8	9	10
6	5	6	7	8	9	10	11
7	6	7	8	9	10	11	12

Figure 19.24 2D6 event grid

At first, this may seem like a large field of arbitrary numbers, but take a few moments to look for patterns. You can see at the top left that there is a single 2, and then, on the diagonal, there are two 3s, then three 4s, four 5s, five 6s, and six 7s. Then the pattern shrinks back down as the numbers get larger, ending with a single 12. To illustrate this further, the spreadsheet in [Figure 19.25](#) shows the same numbers but has conditional formatting applied so that the largest and smallest numbers are highlighted with deeper colors.

	A	B	C	D	E	F	G
1	2D6	1	2	3	4	5	6
2		1	2	3	4	5	6
3		2	3	4	5	6	7
4		3	4	5	6	7	8
5		4	5	6	7	8	9
6		5	6	7	8	9	10
7		6	7	8	9	10	11
		7	8	9	10	11	12

Figure 19.25 2D6 conditionally formatted event grid

The conditional formatting graphically highlights the patterns in the grid. To go from a nice visualization to mathematical clarity, you can use exactly the same method shown for 2D2, which is also the same method used for 1D6. List all the different events, use COUNTIF to find the exact count in the grid, and use those counts to calculate the chance of each number coming up. The spreadsheet should end up looking as shown in [Figure 19.26](#).

	► I	J	K	L
1	2D6	Event	Count	Chance
2		2	1	2.78%
3		3	2	5.56%
4		4	3	8.33%
5		5	4	11.11%
6		6	5	13.89%
7		7	6	16.67%
8		8	5	13.89%
9		9	4	11.11%
10		10	3	8.33%
11		11	2	5.56%
12		12	1	2.78%
13		sum	36	100.00%

Figure 19.26 2D6 probability calculation

Figure 19.27 shows what it looks like when all this spreadsheet's functions and formulas are exposed.

	A	B	C	D	E	F	G	H	I	J	K	L
1	2D6	1	2	3	4	5	6	2D6	Event	Count	Chance	
2		=A2+B\$1	=A2+C\$1	=A2+D\$1	=A2+E\$1	=A2+F\$1	=A2+G\$1		2	=countif(\$B\$2:\$G\$7,J2)	=K2/\$K\$13	
3		=A3+B\$1	=A3+C\$1	=A3+D\$1	=A3+E\$1	=A3+F\$1	=A3+G\$1		3	=countif(\$B\$2:\$G\$7,J3)	=K3/\$K\$13	
4		=A4+B\$1	=A4+C\$1	=A4+D\$1	=A4+E\$1	=A4+F\$1	=A4+G\$1		4	=countif(\$B\$2:\$G\$7,J4)	=K4/\$K\$13	
5		=A5+B\$1	=A5+C\$1	=A5+D\$1	=A5+E\$1	=A5+F\$1	=A5+G\$1		5	=countif(\$B\$2:\$G\$7,J5)	=K5/\$K\$13	
6		=A6+B\$1	=A6+C\$1	=A6+D\$1	=A6+E\$1	=A6+F\$1	=A6+G\$1		6	=countif(\$B\$2:\$G\$7,J6)	=K6/\$K\$13	
7		=A7+B\$1	=A7+C\$1	=A7+D\$1	=A7+E\$1	=A7+F\$1	=A7+G\$1		7	=countif(\$B\$2:\$G\$7,J7)	=K7/\$K\$13	
8									8	=countif(\$B\$2:\$G\$7,J8)	=K8/\$K\$13	
9									9	=countif(\$B\$2:\$G\$7,J9)	=K9/\$K\$13	
10									10	=countif(\$B\$2:\$G\$7,J10)	=K10/\$K\$13	
11									11	=countif(\$B\$2:\$G\$7,J11)	=K11/\$K\$13	
12									12	=countif(\$B\$2:\$G\$7,J12)	=K12/\$K\$13	
13								sum		=sum(K2:K12)	=sum(L2:L12)	

Figure 19.27 2D6 formulas exposed

Note

In [Chapter 10, “Organizing Data in Spreadsheets,”](#) I recommended not using blank columns or rows to separate data in a table. It might seem like I have broken that rule in [Figure 19.27](#) because column H is blank. However, there is a distinction. On this sheet, there are two different tables: a grid of results on the left and the analysis of those results on the right. So column H is between the tables, not within a table. To create the cleanest spreadsheets possible, it would be ideal to separate those two tables into two separate sheets. However, I have shown them on a single sheet here to more clearly show all the information together in one image. There are times when having multiple small tables on a single sheet is an advantage. When in doubt, though, separate data into individual sheets.

Of course, it’s not enough to create a big grid of numbers without taking the time to evaluate them and make observations that have practical effects on games. By looking at the chance of each event in the 2D6 grid, you can see

the same patterns emerge as in the 2D2 grid and in the conditionally formatted 2D6 event grid. The numbers near the center of the list are more likely to occur than are those at the corners. To illustrate this further, you can load the data into a chart (see [Figure 19.28](#)).

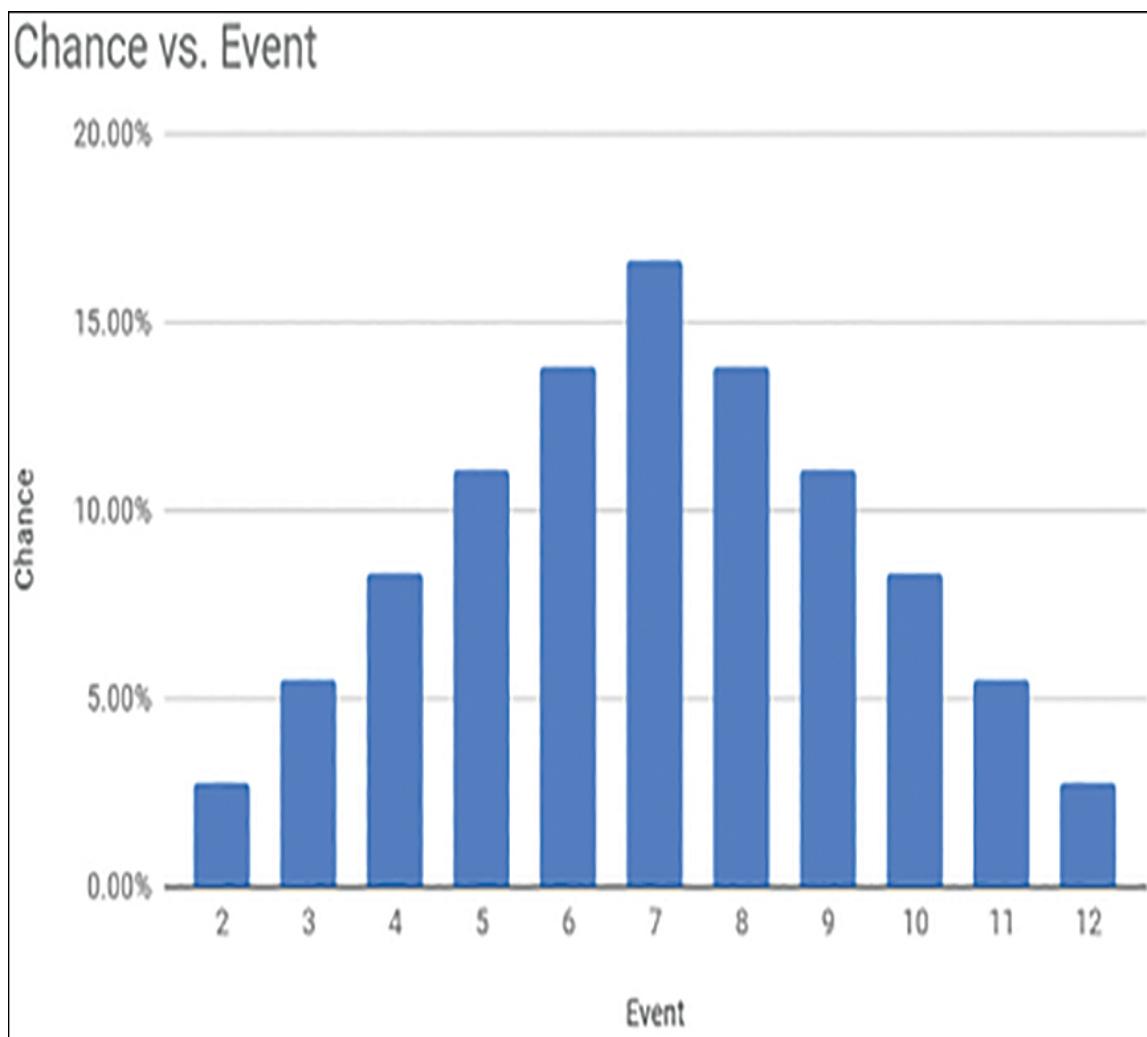


Figure 19.28 2D6 distribution chart

The layout of the bars in this chart is very different from the layout of the bars in any of the charts created for linear probability. This is called *bell curve* probability because it makes a smooth curve from each end up to the middle, similar to the shape of a bell. The distribution in a bell curve is unique in a number of ways:

- The high and low ends are quite rare. In our 2D6 example, rolling either a 2 or a 12 has less than a 3% probability.
- The central point has a very high relative probability. In the case of 2D6, it is 16.67%. It is five times more likely that you will roll a 7 than that you will roll either a 2 or a 12.
- The probability of rolling the central number, in this case 7, is the same probability of rolling any given number on a 1D6. This holds for any bell curve distribution, so if you were rolling 2D8, the probability of rolling a 9 would be the same as rolling any individual number on 1D8.
- The central point is one higher than the count of a single die. If you have 2D6, the center is $6 + 1$, or 7. The central, or most common, number for 2D13 would be 14, for 2D14 would be 15, and so on.
- The distance between any two adjacent chances (called the *step*) is the same as the probability of rolling a number at one of the ends of the curve. For example, there is a 2.78% chance of rolling a 2 on 2D6. The probability of rolling a 7 is also 2.78% higher than that of rolling a 6. This holds true for any other bell curve distribution.

Calculating 2D6 “Or Higher” Cumulative Probability

Once the distribution of 2D6 is in a spreadsheet, you can use exactly the same method used for 1D6 to find the distribution of rolling any given number or higher on 2D6. Again, you start at the highest number and write a single formula that adds up each new probability and provides a running total. The spreadsheet should look as shown in [Figure 19.29](#).

2D6	Event	Count	Chance	orHigher
	2	1	2.78%	100.00%
	3	2	5.56%	97.22%
	4	3	8.33%	91.67%
	5	4	11.11%	83.33%
	6	5	13.89%	72.22%
	7	6	16.67%	58.33%
	8	5	13.89%	41.67%
	9	4	11.11%	27.78%
	10	3	8.33%	16.67%
	11	2	5.56%	8.33%
	12	1	2.78%	2.78%
sum		36	100.00%	

Figure 19.29 2D6 “or higher” cumulative probability spreadsheet

When graphed out, the distribution looks as shown in [Figure 19.30](#).

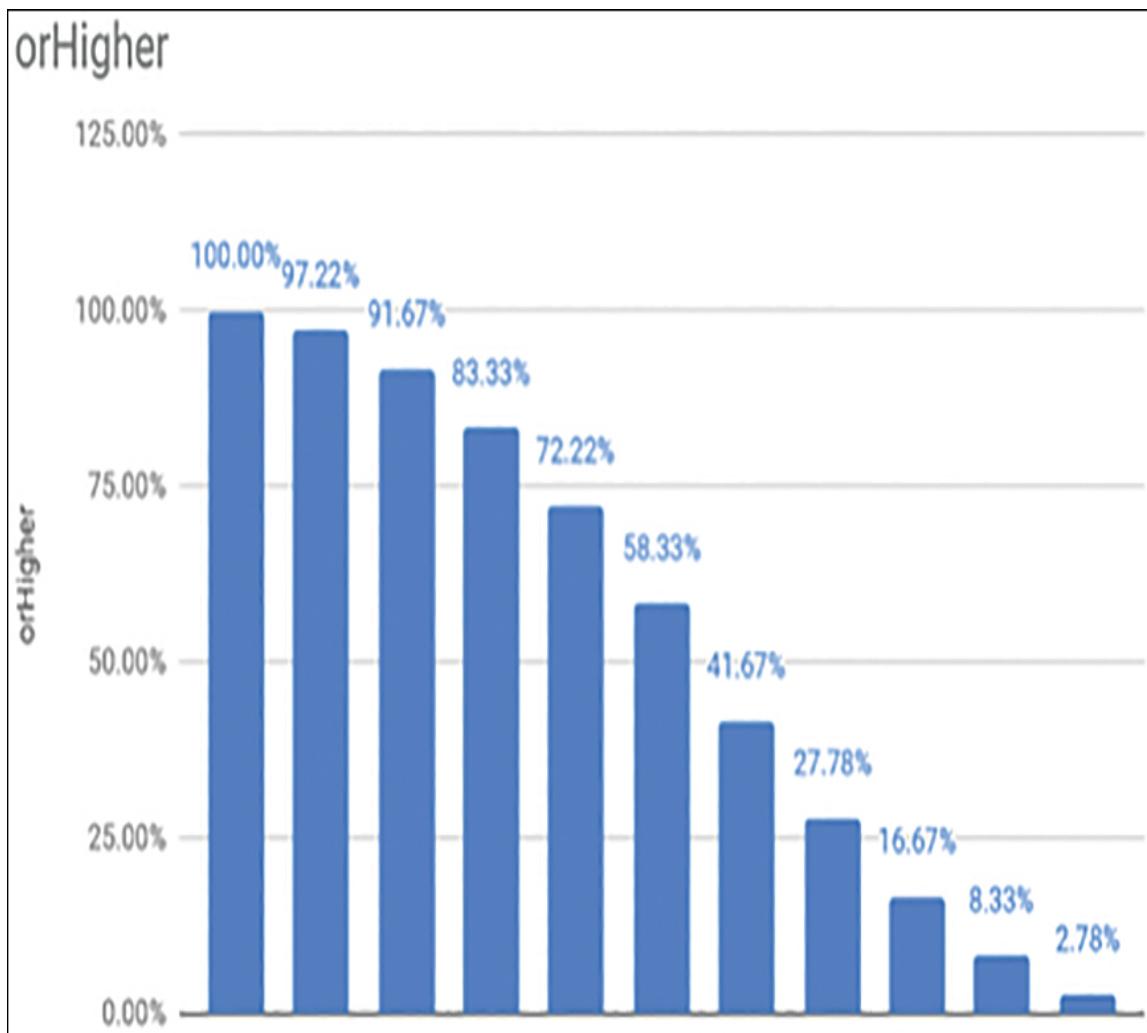


Figure 19.30 2D6 “or higher” cumulative probability chart

Consider the following important observations about the cumulative probability distribution of 2D6 that work with any two-dimensional probability table:

- The steps in between numbers at the top and bottom of the chart are small, but the steps between numbers in the center of the chart are very large. For example, a player who needs to roll an 11 or better has a 5.56% greater chance of succeeding than if they need to roll a 12 or better. In contrast, a player who needs to roll a 7 or better has a 16.67% greater chance of succeeding than if they need to roll an 8 or better; the

increase in a single step is three times greater in the middle of the chart than in the ends.

- Exactly the same probability applies to “or worse” charts as to “or higher” charts—but in the reverse order.

Calculating the Probability of Doubles

In many games, rolling doubles—that is, the same number on two dice—holds significance. To help you understand how to calculate the probability of rolling any given doubles, [Figure 19.31](#) shows the same 2D6 spreadsheet as before but now with shading applied to the doubles.

2D6	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figure 19.31 Doubles table

You can see in [Figure 19.31](#) that the pattern of doubles runs diagonally through the chart in the opposite direction from the 7s, and the count is exactly the same: There are 6 ways to roll doubles, and there are 6 ways to roll a 7. Because there are 6 ways to roll doubles out of 36 possible outcomes for 2D6, there is (yet again) a 16.67% chance of rolling doubles.

Note

You may have noticed that 16.67% keeps popping up when dealing with 1D6 and 2D6 probabilities. This is not a coincidence. The chance of an occurrence of any single result will be embedded throughout the probability of the roll: You can find the probability of doubles on any two dice roll by finding the probability of rolling a given number on one of those dice.

Calculating a Series of Single Events

With the calculations described so far in this chapter, you can figure out the probability of a vast array of different dice rolls, but those are all individual events. What do you do with a series of events? You know the chance of rolling a 1 on a one-sided die, and you have calculated the chance of rolling a 1 on each of two six-sided dice. This is the start of a series. Now, let's take this series further with a few examples.

What is the probability of rolling a 1 on 1D6 three, four, five, or more time in a row? To create a series of probability events, you must multiply the probability of the current event by itself for each new step. This may seem unintuitive, but a graphic example can help explain. Let's say you have a whole pizza. If you were to eat half of it, how much would be left (see [Figure 19.32](#))?

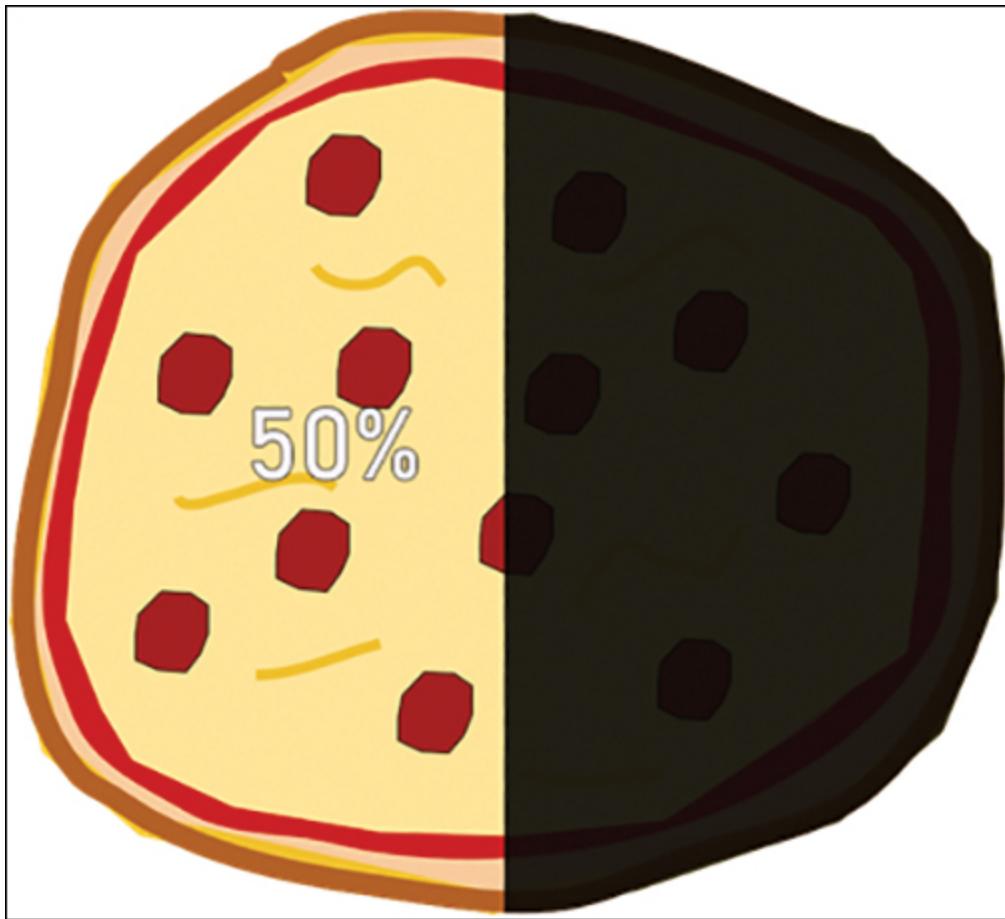


Figure 19.32 Half a pizza

You would have one-half (or $1/2$ or 50%). This is easy enough to calculate. What if you again eat half of the remaining pizza? How much is left now? You don't subtract half of the whole, or you would be left with 0%. Instead, you would subtract half of the half, so you end up with 25% (see [Figure 19.33](#)). You can apply the same method to any series of probability events.

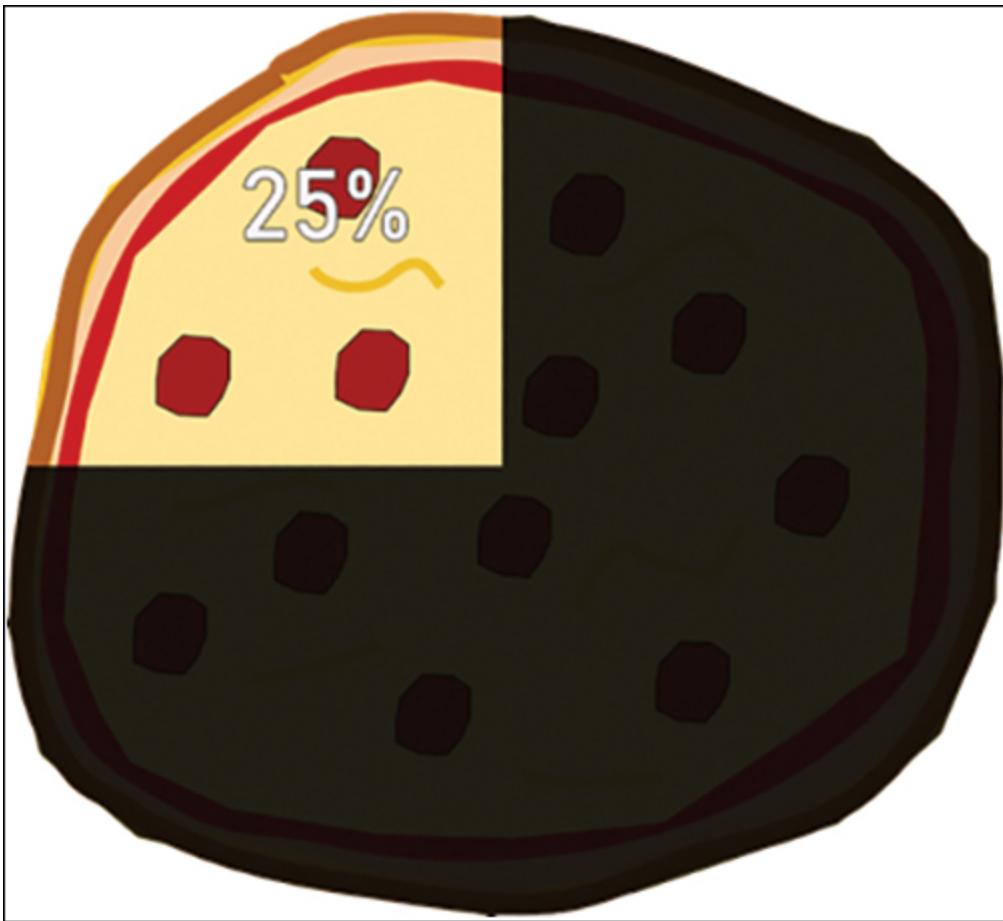


Figure 19.33 A quarter of a pizza

Now let's think about rolling a 1 on 1D6 again. In a spreadsheet, you can set up a formula starting in the first iteration with the odds 1/6 (see [Figure 19.34](#)). For all iterations down the sheet, you multiply the previous iteration by 1/6 again.

	A	B
1	In a row	Odds
2	1	=1/6
3	2	=B2*(1/6)
4	3	

Figure 19.34 Probability series formula

After you write the formula in B2, you can formfill it down to produce the odds of a 1 being rolled any number of times. As you can see in [Figure 19.35](#), the chance of this happening is incredibly small, but it will never reach zero.

	A	B
1	In a row	Odds
2	1	16.666667%
3	2	2.777778%
4	3	0.462963%
5	4	0.077160%
6	5	0.012860%
7	6	0.002143%
8	7	0.000357%
9	8	0.000060%
10	9	0.000010%
11	10	0.000002%

Figure 19.35 Probability series

This kind of spreadsheet works well for a one-time calculation of a series. You can also go a bit further and make a dynamic spreadsheet that will allow you to input any probability numbers you want, and the spreadsheet will automatically figure out the probability of the series. The variables you need, just as before, are number of events and number of successes. With those two variables, the spreadsheet does the rest. [Figure 19.36](#) shows all the related formulas exposed.

	A	B	C	D	E
1	In a row	Odds		Possible	6
2	1	=E3		Successes	1
3	2	=B2*\$E\$3		Probability	=E2/E1
4	3	=B3*\$E\$3			
5	4	=B4*\$E\$3			
6	5	=B5*\$E\$3			
7	6	=B6*\$E\$3			
8	7	=B7*\$E\$3			
9	8	=B8*\$E\$3			
10	9	=B9*\$E\$3			
11	10	=B10*\$E\$3			

Figure 19.36 Variable-driven series calculation

Once this spreadsheet is set up, you can change the “possible” and “successes” numbers and cause the probability of each item in the series to be recalculated.

Now let’s look at an atypical example. The spreadsheet in [Figure 19.37](#) shows 7 possible successes out of 11 possibilities. (You are very unlikely to find these probabilities on dice, but you might find them in cards or with a video game loot table.)

	A	B	C	D	E
1	In a row	Odds	Possible		11
2	1	63.6364%	Successes		7
3	2	40.4959%	Probability		63.6364%
4	3	25.7701%			
5	4	16.3992%			
6	5	10.4358%			
7	6	6.6410%			
8	7	4.2261%			
9	8	2.6893%			
10	9	1.7114%			
11	10	1.0891%			

Figure 19.37 Series output

This kind of formula can be applied to any set of probability to find the likelihood of the event happening. The same formula can be used in inverse as well. Let's say you have a game where you roll 2D6, and if you get a 12, you automatically lose; otherwise, you can continue rolling. In this case, you are not trying to figure out the odds of rolling a 12 multiple times in a row; instead, you are trying to figure out the odds of *not* rolling a 12. While this may seem more complex, it is exactly the same calculation but with one additional step. You know that the odds of rolling a 12 on 2D6 are 1/36, or 2.78%. This means that the odds of not rolling 12 are 100% minus the odds of rolling 12: $100\% - 2.78\% = 97.22\%$. You can then substitute these numbers into the original formula. To adapt the spreadsheet to account for this, as shown in [Figure 19.38](#), you can add one more calculation below Probability, for Not, and you can add one more column, Inverse, which is the probability of the event not happening a number of times in a row.

	A	B	C	D	E	F
1	In a row	Odds	Inverse		Possible Successes	36
2	1	=F3	=F4		Probability	1
3	2	=B2*\$F\$3	=C2*\$F\$4		Not	=F2/F1
4	3	=B3*\$F\$3	=C3*\$F\$4			=1-F3
5	4	=B4*\$F\$3	=C4*\$F\$4			
6	5	=B5*\$F\$3	=C5*\$F\$4			
7	6	=B6*\$F\$3	=C6*\$F\$4			
8	7	=B7*\$F\$3	=C7*\$F\$4			
9	8	=B8*\$F\$3	=C8*\$F\$4			
10	9	=B9*\$F\$3	=C9*\$F\$4			
11	10	=B10*\$F\$3	=C10*\$F\$			

Figure 19.38 Underlying formulas for series and inverse

Note that with both formula series, there are mixed absolute and relative references. In the series, the references need to be relative so that they will update with each iteration of the series. When the formulas are pointing to the root variables, they need to be fixed because those variables don't change as the formula is formfilled down. With these formulas in place, the final odds should look as shown in [Figure 19.39](#).

	A	B	C	D	E	F
1	In a row	Odds	Inverse		Possible	36
2	1	2.778%	97.222%		Successes	1
3	2	0.077%	94.522%		Probability	2.78%
4	3	0.002%	91.896%		Not	97.22%
5	4	0.000%	89.343%			
6	5	0.000%	86.862%			
7	6	0.000%	84.449%			
8	7	0.000%	82.103%			
9	8	0.000%	79.822%			
10	9	0.000%	77.605%			
11	10	0.000%	75.449%			

Figure 19.39 Inverse series of events output

You can now see that the chance of rolling 2D6 10 times in a row without ever getting a 12 is slightly over 75%. Those are pretty good odds!

Let's look at a final example of calculating the inverse. This time, say that you want to find the likelihood that a player will not roll a 6 on 1D6. Based on basic intuition, many people assume that if you roll a 1D6 six times, it should be nearly inevitable to roll a 6 at least once. But [Figure 19.40](#) shows what the actual probability is.

	A	B	C	D	E	F
1	In a row	Odds	Inverse	Possible	6	
2	1	16.667%	83.333%	Successes	1	
3	2	2.778%	69.444%	Probability	16.67%	
4	3	0.463%	57.870%	Not	83.33%	
5	4	0.077%	48.225%			
6	5	0.013%	40.188%			
7	6	0.002%	33.490%			
8	7	0.000%	27.908%			
9	8	0.000%	23.257%			
10	9	0.000%	19.381%			
11	10	0.000%	16.151%			

Figure 19.40 Odds of NOT rolling a 6 on a six-sided die in a series

As it turns out, there is more than a 33% chance that you *won't* roll a 6 on a six-sided die after six rolls. It's actually quite possible that you might go 10 rolls in a row and still not see a 6. The reason for this is that each event is independent.

Calculating More Than Two Dimensions

Up to this point, we have only been discussing one- and two-event probability. A spreadsheet has two dimensions, so it is a natural fit for one or two dimensions. When you move up to calculating three dice or three dimensions, you must get a bit more creative. With this many dimensions, you need to list the pairings of two of the dice in the vertical dimension and use the horizontal dimension for the third die. That is, you use the first two columns of the spreadsheet to list every possibility for the first two dice, by repeating the first die in increments in the first column and adding each combination with the second die in the second column. [Figure 19.41](#) shows how to do this manually for 3D4.

	A	B	C	D	E	F	G	H	I	J	K	L
1	First	Second	Third	1	2	3	4	3D4	Event	Count	Chance	orHigher
2	1		1	3	4	5	6		3	1	1.56%	100.00%
3	1		2	4	5	6	7		4	3	4.69%	98.44%
4	1		3	5	6	7	8		5	6	9.38%	93.75%
5	1		4	6	7	8	9		6	10	15.63%	84.38%
6	2		1	4	5	6	7		7	12	18.75%	68.75%
7	2		2	5	6	7	8		8	12	18.75%	50.00%
8	2		3	6	7	8	9		9	10	15.63%	31.25%
9	2		4	7	8	9	10		10	6	9.38%	15.63%
10	3		1	5	6	7	8		11	3	4.69%	6.25%
11	3		2	6	7	8	9		12	1	1.56%	1.56%
12	3		3	7	8	9	10			sum	64	100.00%
13	3		4	8	9	10	11					
14	4		1	6	7	8	9					
15	4		2	7	8	9	10					
16	4		3	8	9	10	11					
17	4		4	9	10	11	12					

Figure 19.41 3D4 spreadsheet of 3 dimensional probability

In this example, you can see in column A that there are four 1s followed by four 2s and so on until all possible numbers on the first die are complete and copied as many times as there are possible outcomes on the second die (four). Column B then cycles through each of the options on the second die to show the unique combinations between each of the first two dice. Finally, the rest of row 1 is used to list all the possible options on the third die. Each cell in the range C2:F17 adds up the first, second, and third dice. After that, the Event, Count, Chance, and higher values are calculated in exactly the same way that 1D and 2D combinations are calculated. This same method can be applied with more columns for more dice, although it does get rather complex. Once all the 3D4 probabilities are calculated, you can create a

chart with the data; you should see that the chart is much higher in the middle and also more concentrated toward the middle (see [Figure 19.42](#)).

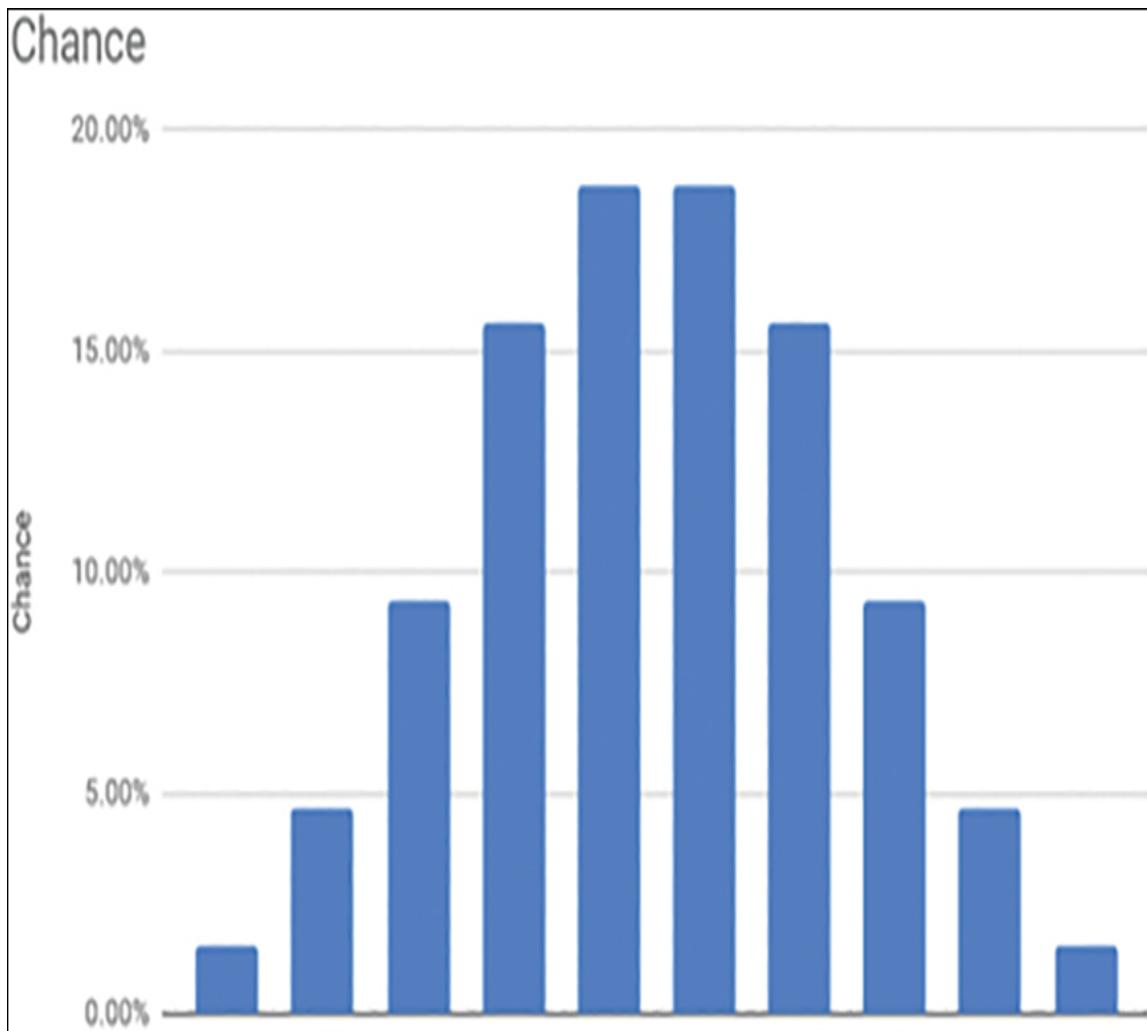


Figure 19.42 3D4 chart showing a steep bell curve in the center of probability

For each die added as a new dimension, this trend continues to grow stronger. The more dice in a pool, the more rare the chance of one of the ends occurring and the more likely it is that one of the most central possibilities will occur.

Calculating Dependent Event Probability

In every calculation of probability discussed so far, any given result has had no effect on any other given result. Rolling a single die has no effect whatsoever on the next die roll, and getting any color on a spinner has no effect on the next spin. These are the easiest kinds of probability events to track and calculate, but there are plenty of probability events that do change based on previous events. The most common is the draw of multiple cards from a deck of cards. Another event with dependence is the draw of a colored tile from a pool of colored tiles. A game might also have a loot pool of items that are removed from the pool as they are collected by players. The common feature that all of these events have is that the occurrence of the first event affects the probability of the next. Instead of using an axis to calculate these kinds of events, you use branching because each event creates a new set of probability.

As a very simple example, let's say you have 2 circles and 1 square. What is the chance that, in a completely random draw, you would get a circle? In exactly the same fashion as we calculated for dice earlier in this chapter, you would note 3 possible outcomes and 2 possible successes, or a $2/3$ chance of success, which equates to 66.66% probability. Now let's say that you have drawn one of the shapes, and you draw again from the remaining two shapes. There are no longer 3 circles but 2, so the odds have changed. If you select 1 of these remaining 2 circles, you have 1 shape remaining. It is possible to graph the possibilities with each of these selections as shown in [Figure 19.43](#).

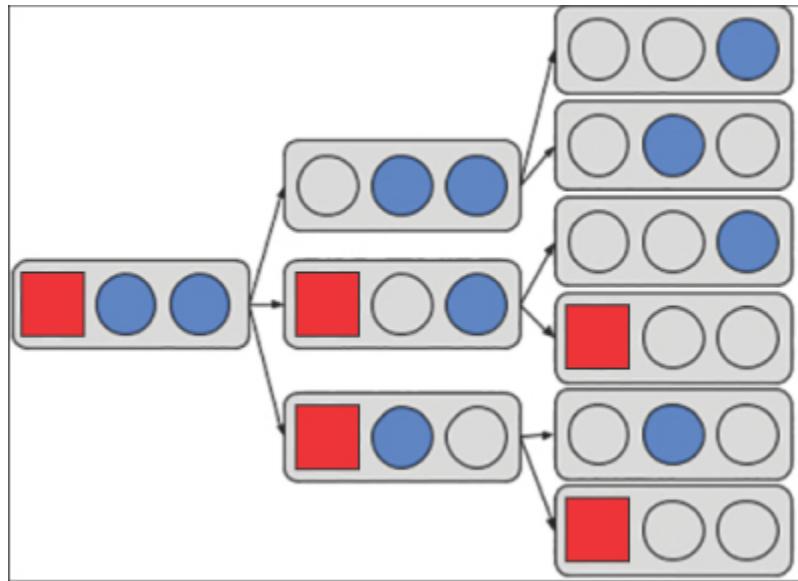


Figure 19.43 Probability tree with all possibilities still open

On the far left in [Figure 19.43](#), you can see that the first event has 3 possible outcomes, and each of those 3 possibilities has 2 possible outcomes. In this scenario, with just 3 shapes, there are 6 possible outcomes:

Square then circle

Square then circle

Circle then square

Circle then circle

Circle the square

Circle then circle

Now that you know how many events there are and how many possibilities are generated, you can calculate the probability of lots of different possibilities. For example, if you wanted to find the chance of a square being the last shape drawn, you would look at the third column and count the total possibilities as 6 and the successes as 2 (where squares remain) and get $2/6$ (which simplifies to $1/3$), or 33.3% chance of a square being the last

remaining shape. Much as with a series of dice, you could also calculate the chance of each event along the way.

For example, if a circle is drawn first, what is the chance of a square being drawn second? In this case, you focus on just the branches where a circle is drawn first, as illustrated in [Figure 19.44](#).

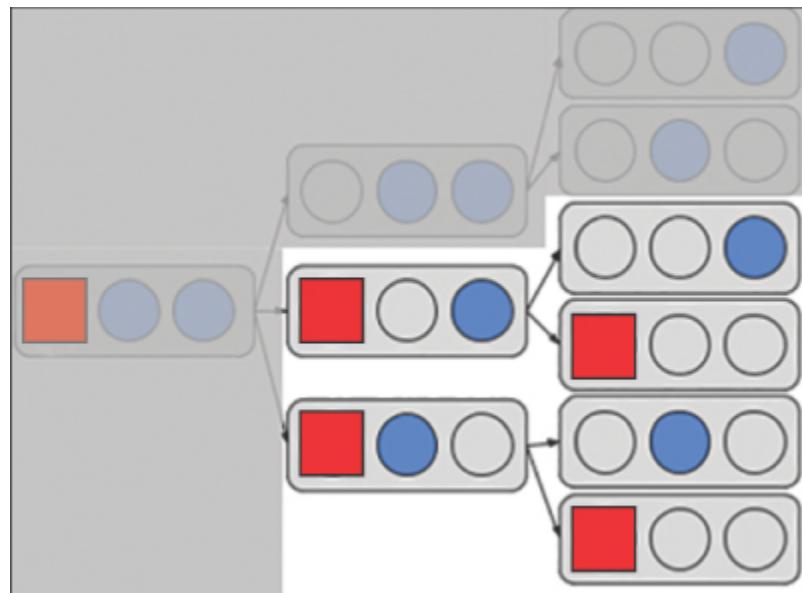


Figure 19.44 Probabilities of drawing a circle first

When you look at the remaining events in the right column, you can see that there are 4 possible outcomes, and in 2 of them, squares are drawn, so the probability is $2/4$, or $1/2$, or 50%.

Calculating dependent probability is certainly more complex than calculating independent probability, but it is well worth exploring because games use it often.

As another example, consider a deck of 52 cards that is divided into 4 equal suits of 13 cards: hearts, spades, clubs, and diamonds. What is the probability of drawing a heart? What is the probability of drawing 2 hearts in 2 draws? To find this answer, you would use a similar method as with the checkers, but this time you can focus on a single branch of the tree.

For the first draw, 13 of 52 cards would be successful draws—that is, $13/52$, or $1/4$, or 25%. In the second draw, however, there are only 12 hearts remaining, and there are 13 each of the other suits. The probability of a successful draw would now be $12/51$ (remember that the total is also lower now), or 23.53%—which is slightly lower than it was on the first draw. You use a spreadsheet to calculate this single branch of the tree by iterating the remaining hearts and the remaining total cards (see [Figure 19.45](#)).

C2	A	B	C
1	Hearts	Total	Chance to draw
2	13	52	25.00%
3	12	51	23.53%
4	11	50	22.00%
5	10	49	20.41%
6	9	48	18.75%
7	8	47	17.02%
8	7	46	15.22%
9	6	45	13.33%
10	5	44	11.36%
11	4	43	9.30%
12	3	42	7.14%
13	2	41	4.88%
14	1	40	2.50%

Figure 19.45 Series of probability of card draws

Note

Note that this example follows only one branch of the immense number of possible branches available when drawing cards from a deck. When calculating dependent probability, it's often not possible to graph out the entire tree of possibilities, as it would be too large for any practical use. Instead, you can calculate a branch of it, as was shown here, to get a sample of what is going on in the data.

Calculating Mutually Exclusive Event Probability

In the previous section, we looked at the odds of drawing a certain suit from a deck of cards in a series. You saw that drawing two hearts in a row involves dependent events because the draw of the first card has an influence on the probability of the second draw. Another type of probability you might have to calculate at some point involves mutually exclusive events. *Mutually exclusive events* are events that cannot occur together: When one event happens, the other can't happen.

For example, say that you are calculating the probability of drawing the ace of spades from a card deck. On the first try, it would be $1/52$, or 1.92%. If the ace of spades is not drawn on the first attempt, then on the second attempt, the odds improve slightly because the pool of cards has been reduced from 52 to 51. The odds of drawing the ace of spades on the second attempt would be $3241/51$, or 1.96%. But what if the ace of spades is drawn on the first attempt? In this case, the odds of drawing the ace of spades on the second attempt would be 0% because there is no ace of spades left in the deck. Similarly, if you were to draw two cards at once, what would be the odds of drawing the ace of spades twice? Again, 0%. This is what it means to be a mutually exclusive probability event. If one of the events happens, the other event can't happen.

Calculating Enumerated Probability with an Even Distribution

Not all random events happen with fixed-quantity variables such as dice and standard card decks. Many modern games have customizable, variable-sized card decks, loot tables, and other methods of randomizing among an

indeterminate list of items. For such pools of items, you must calculate enumerated probability. Calculating enumerated probability works very much like the methods already discussed. For example, let's look at a small loot table for an RPG (see [Figure 19.46](#)).

	A	B
1	Name	Sell Value
2	Small Coin Purse	10
3	Coin purse	15
4	Small Chest	30
5	Treasure Chest	50
6	Weak Health Potion	5
7	Minor Health Potion	15
8	Health Potion	30
9	Strong Health Potion	50
10	Cloth	1
11	Healing Potion	60
12	Mesmo Scroll	10
13	Book of Knowledge	10
14	Cloth Armor	5
15	Troll Ring	10
16	Treasure Map	0
17	Rabbit Foot	15

Figure 19.46 Loot table enumerated probability with an even distribution

This list includes several different items, each having a value. If a player were to get an item each time they opened a chest, it would be easy enough to calculate the probability of each item coming up by using the same method used earlier in this chapter for dice: You would add up the total—in this case 16 items—and specify 1 as the chance of each item coming up. From a mathematical standpoint, this is exactly the same as 1D16 and should be treated the same. So each item would have 1/16 chance, or 6.25% probability of being drawn. Calculating enumerated probability with an even distribution works the same way regardless of how long the list of items and regardless of whether it is done with a spinner, a deck of cards, or a computer.

Calculating Enumerated Probability with an Uneven Distribution

While calculating the probability of a list of items is useful for simpler games, in even mildly complex games, there is often an uneven chance of events from a list occurring. With the RPG loot list, for example, you might want the small coin purse to be the item most commonly drawn and for some of the better items to be much more rare. There are a few ways to handle, and two of the most common are by using percentages and by using probability weights.

To use percentages, you list the percentage chance of each event in the list occurring. This is really all there is to it. The upside of using percentages is that the chance of each event occurring is clearly shown. The downside of using percentages is that you as the data designer must ensure that the list of percentages always adds up to 100%, or the probability will not be accurate. This means any change made to the chance of any item on the list occurring can cause a cascade effect of percentages changing, requiring constant rebalancing.

Using probability weights, on the other hand, does not initially seem possible, and there is no fixed scale of probability to start with. You, as the

designer, must decide what the smallest weight is and what the largest is, and then you distribute weights to all the items on the list. For the RPG loot example, [Figure 19.47](#) shows each of the items on the list now been given a probability weight.

	A	B	C
1	Name	Sell Value	Weight
2	Small Coin Purse	10	60
3	Coin purse	15	20
4	Small Chest	30	20
5	Treasure Chest	50	10
6	Weak Health Potion	5	30
7	Minor Health Potion	15	50
8	Health Potion	30	20
9	Strong Health Potion	50	10
10	Cloth	1	40
11	Healing Potion	60	10
12	Mesmo Scroll	10	10
13	Book of Knowledge	10	10
14	Cloth Armor	5	10
15	Troll Ring	10	10
16	Treasure Map	0	10
17	Rabbit Foot	15	10

Figure 19.47 Loot table enumerated probability with an uneven distribution

In the spreadsheet in [Figure 19.47](#), why is the smallest weight 10 and the largest 60? Whoever designed this example wanted the most common item to occur 6 times as frequently as the rarest, based on balancing goals for the game. The smallest number was decided as 10 to leave room in the

weighting scale in case the team forgot about some items that should be of even lower weight or needed to make adjustments to put an item rarity between two weights that are already listed. One of the upsides of using arbitrary probability weight values is that there is no minimum or maximum. Each team can determine what scale to use for a specific game. Another large benefit is that the list does not need to add up to 100. It can add up to anything and still work.

Once each item in the list has a weight assigned, you can calculate the probability of each event in exactly the same way you calculated probability for 2D6. First, you get a sum of the weights (in column C in [Figure 19.48](#)), and then you divide each event weight by the total.

fx | =C2/\$G\$1

	A	B	C	D	E	F	G
1	Name	Sell Value	Weight	%	Total	330	
2	Small Coin Purse	10	60	18.18%			
3	Coin purse	15	20	6.06%			
4	Small Chest	30	20	6.06%			
5	Treasure Chest	50	10	3.03%			
6	Weak Health Potion	5	30	9.09%			
7	Minor Health Potion	15	50	15.15%			
8	Health Potion	30	20	6.06%			
9	Strong Health Potion	50	10	3.03%			
10	Cloth	1	40	12.12%			
11	Healing Potion	60	10	3.03%			
12	Mesmo Scroll	10	10	3.03%			
13	Book of Knowledge	10	10	3.03%			
14	Cloth Armor	5	10	3.03%			
15	Troll Ring	10	10	3.03%			
16	Treasure Map	0	10	3.03%			
17	Rabbit Foot	15	10	3.03%			

Figure 19.48 Loot table enumerated probability with an uneven distribution, using probability weights

Once the weights are set up for calculation into a percentage chance, you can change any of the raw weight values, and the entire list will automatically update. To illustrate this, let's say that the healing potion has occurred as a result too often. To fix this, you can simply lower the individual weight for that item, and the spreadsheet will instantly

recalculate all the probability chances for the entire list, as shown in [Figure 19.49](#).

A	B	C	D	E	F	G
1	Name	Sell Value	Weight %	Total	325	
2	Small Coin Purse	10	60	18.46%		
3	Coin purse	15	20	6.15%		
4	Small Chest	30	20	6.15%		
5	Treasure Chest	50	10	3.08%		
6	Weak Health Potion	5	30	9.23%		
7	Minor Health Potion	15	50	15.38%		
8	Health Potion	30	20	6.15%		
9	Strong Health Potion	50	10	3.08%		
10	Cloth	1	40	12.31%		
11	Healing Potion	60	5	1.54%		
12	Mesmo Scroll	10	10	3.08%		
13	Book of Knowledge	10	10	3.08%		
14	Cloth Armor	5	10	3.08%		
15	Troll Ring	10	10	3.08%		
16	Treasure Map	0	10	3.08%		
17	Rabbit Foot	15	10	3.08%		

Figure 19.49 Loot table adjusted weight

With this basic setup, a list of any size with probability weights in any range can be balanced and calculated instantly. If a game engine needs a percentage chance, those numbers are available for export. Similarly, if you want to show the chance to players, you could add one more column for rounded display values.

Calculating Attributes Weights Based on Probability

In [Chapter 12](#), “System Design Foundations,” you learned applying weights to attributes based on usefulness in game. However, there is yet another layer to finding the value of attributes that are influenced by chance. In the RPG loot table example, what is the value of loot that the player receives on a given draw or on an average draw? To make this somewhat complex calculation a bit easier to digest, let’s start with a very simple example. Say that you have 10 items, each with an equal chance of happening, at 10%. Nine of those items have a value of 0, and 1 item has a value of 100 (see [Figure 19.50](#)).

	A	B	C	D
1	Items	Value	Weight	Chance
2	a	100	1	10.00%
3	b	0	1	10.00%
4	c	0	1	10.00%
5	d	0	1	10.00%
6	e	0	1	10.00%
7	f	0	1	10.00%
8	g	0	1	10.00%
9	h	0	1	10.00%
10	i	0	1	10.00%
11	j	0	1	10.00%

Figure 19.50 Even weight distribution in an enumerated list

In this scenario, what is the average payout for a draw from the loot table? In this case, there is a 10% chance of a payout of 100, so on average, the payout would be 10% of 100, or 10. You could do this calculation in the spreadsheet as shown in [Figure 19.51](#).

	A	B	C	D	E	F	G	H
1	Items	Value	Weight	% Chance	% Value			
2	a	100	1	10.00%	10.0	Total % Value	10.0	
3	b	0	1	10.00%	0.0			
4	c	0	1	10.00%	0.0			
5	d	0	1	10.00%	0.0			
6	e	0	1	10.00%	0.0			
7	f	0	1	10.00%	0.0			
8	g	0	1	10.00%	0.0			
9	h	0	1	10.00%	0.0			
10	i	0	1	10.00%	0.0			
11	j	0	1	10.00%	0.0			

Figure 19.51 Returned value of all enumerated items

For each event, the percentage chance of occurrence is multiplied by the value of the event. The average value for a draw is then calculated by adding up the percentage chance values of all the events. In this case, it is 10.

To move up in complexity, we can return to our RPG loot table example. Here, the events have different weights and values, so how do you calculate

the expected value per draw? You use the same method just shown and also use two additional evaluations:

- **Total Weights:** Determines the chance based on weight.
- **Average Sell Value:** The value that each draw would be if there were no weights applied to the events.

The formulas for the entire chart look as shown in [Figure 19.52](#).

	A	B	C	D	E	F	G	H
1	Name	Sell Value	Weight	% Chance	% * Value			
2	Small Coin Purse	10	60	=C2/\$H\$2	=D2*B2	Total Weights	=sum(C2:C17)	
3	Coin purse	15	20	=C3/\$H\$2	=D3*B3	Ave sell value	=AVERAGE(B2:B17)	
4	Small Chest	30	20	=C4/\$H\$2	=D4*B4	Event Ave Value	=sum(E2:E17)	
5	Treasure Chest	50	10	=C5/\$H\$2	=D5*B5			
6	Weak Health Potion	5	30	=C6/\$H\$2	=D6*B6			
7	Minor Health Potion	15	50	=C7/\$H\$2	=D7*B7			
8	Health Potion	30	20	=C8/\$H\$2	=D8*B8			
9	Strong Health Potion	50	10	=C9/\$H\$2	=D9*B9			
10	Cloth	1	40	=C10/\$H\$2	=D10*B10			
11	Healing Potion	60	5	=C11/\$H\$2	=D11*B11			
12	Mesmo Scroll	10	10	=C12/\$H\$2	=D12*B12			
13	Book of Knowledge	10	10	=C13/\$H\$2	=D13*B13			
14	Cloth Armor	5	10	=C14/\$H\$2	=D14*B14			
15	Troll Ring	10	10	=C15/\$H\$2	=D15*B15			
16	Treasure Map	0	10	=C16/\$H\$2	=D16*B16			
17	Rabbit Foot	15	10	=C17/\$H\$2	=D17*B17			

Figure 19.52 Returned value formulas

And the results of using these formulas look as shown in [Figure 19.53](#).

	A	B	C	D	E	F	G	H
1	Name	Sell Value	Weight	% Chance	% * Value			
2	Small Coin Purse	10	60	18.46%	1.846	Total Weights	325	
3	Coin purse	15	20	6.15%	0.923	Ave sell value	19.75	
4	Small Chest	30	20	6.15%	1.846	Event Ave Value	14.892	
5	Treasure Chest	50	10	3.08%	1.538			
6	Weak Health Potion	5	30	9.23%	0.462			
7	Minor Health Potion	15	50	15.38%	2.308			
8	Health Potion	30	20	6.15%	1.846			
9	Strong Health Potion	50	10	3.08%	1.538			
10	Cloth	1	40	12.31%	0.123			
11	Healing Potion	60	5	1.54%	0.923			
12	Mesmo Scroll	10	10	3.08%	0.308			
13	Book of Knowledge	10	10	3.08%	0.308			
14	Cloth Armor	5	10	3.08%	0.154			
15	Troll Ring	10	10	3.08%	0.308			
16	Treasure Map	0	10	3.08%	0.000			
17	Rabbit Foot	15	10	3.08%	0.462			

Figure 19.53 Table calculation

You can see that the chance of each event occurring will affect the outcome of the calculated value, as events with a smaller chance of happening will contribute less to the average than events that have a high chance of happening. You can see here that if unweighted, a draw would, on average, have a monetary value of 19.75, but with weights, it has been dropped to 14.892. This makes sense because the more valuable items are also the rarest and will be drawn far fewer times than will the cheaper, more common items. By doing calculations like this, you can better predict the player experience throughout the game. You know, for example, that if you want the player to have a minimum average draw of 45 gold coins by the end of a scenario, three draws from the loot table will, on average, not be

quite enough to hit that goal. Using this information, you could then adjust the scenario to allow a fourth draw from the loot table.

Calculating Imperfect Information Probability

Up to this point, this chapter has discussed only events that are clearly and measurably random. However, games are full of other events that are both random and not random, based on perspective. Rock, Scissors, Paper, is a classic example. Each player chooses an option, so it would seem that it would not be random. However, each player does not know what the opponent is going to do, so their choice of options is based on unknowable outcomes—and this is the definition of random. In this case, the events are both random and not random.

In games, you also face many other similar scenarios. For example, in an FPS, there is no way to know for sure where opponent players will be at any given moment, which makes their appearance seemingly random—or unpredictable—to a player who is ambushed. This is not true randomness but instead *imperfect information*.

Trying to do calculations of probability based on imperfect information is impractical at a theoretical level, but it can be measured during testing through observation and telemetry. You could, for example, run hundreds of tests to determine what is the most commonly chosen option in rock paper scissors, or you could track players in the FPS to determine most likely paths taken.

Perception of Probability

Another very important concept that all system designers must be aware of is that players do not perceive probability of events in games in a realistic way. There is a psychological phenomenon called *optimism bias* which roughly states that players think they are less likely than they really are to receive a negative event. In game systems, if a player has a 40% chance to get lucky and receive a random reward, because of the optimism bias, they will expect that reward far greater than 40% of the time. This does not make logical sense, but it's the way that humans think.

As another example, if a player has a 70% chance to hit a target, the player is likely to get frustrated at the game if they miss 3 out of 10 tries—even though the game has clearly stated the 70% chance of success, which implicitly means a 30% chance of failure. So what can you do about this? Some games blatantly lie to the players. They show the player far lower chances of success than are running in the background of the video game. Other games make the odds of everything happening high and deal with the ramifications. Other games leave everything as is and learn to deal with grumpy players who all feel they have terrible luck.

Probability Uncertainty

When learning the methods of predicting probability discussed so far in this chapter, there is a natural tendency to feel as though you can predict probability with some certainty. While this is true, there is a very important caveat you need to understand: All the methods described here represent mathematical hypotheticals and not real-world data.

For example, think about an 1D2 coin. What are the odds of getting a heads? As discussed earlier, the odds in this case are 1 chance out of 2 possible, or $1/2$, or 50%. You might assume that if you flipped a coin 4 times, it would come up heads 2 times and tails 2 times. It might, but it also might not. The 50% chance is not a cosmic certainty that forces all coin flips to be distributed perfectly. Instead, the 50% is an average that often has little effect on short-term events. It would be entirely possible to flip a coin 4 times and get 4 heads.

To take this one step further, if you were to roll 1D6 1,000 times, you would expect each number to come up 16.67% of the time, or 167 times. It is highly unlikely that events would unfold this way, but the more iterations done on the random event, the closer you are likely to get to the average expected probability outcome. So you could easily see a range from 127 to 207 rolls of any given number, and it would be highly unlikely to see 900 rolls of a given number (although it would not be impossible).

Mapping Probability

As you start to develop a game, it is a good exercise to map the probability of events in the game. As a system designer, you should have a good understanding of what kinds of randomness your players will be facing. You can use this information to purposefully create scenarios and feelings that are appropriate for the game. While it is not necessary, or even possible, to graph every random event that happens in a game, it's a good practice to create a large sample of graphs to get better insights into the function of randomness in the game. This is similar, in theory, to graphing sentences for children as they learn to read and write; it's not necessary to graph every sentence written to make graphing sample sentences useful.

Attributes of a Random Event

Each random event has several attributes that change its effect on a game. These attributes can be charted for any individual random event and will create patterns that more deeply inform how that event will affect the game.

The following sections discuss these attributes of a random event:

- Computation use
- Measure of random result
- Type of randomness
- Probability distribution
- Outcome dependency

Computation Use

Computation use measures whether and how a random result can be calculated and manipulated for mathematical functionality. This is important to use as game designers to know if we are able to use the generated random result as a starting point for mathematical manipulation. For example, being able to add dice together.

- **Numeric:** A finite, known range of numbers. Results can be used for mathematical functionality.

Examples: 1–6 on a die, 1–36 on a roulette wheel, 1–13 in a suit of cards (when treating face cards and the ace as numbers), distance from a target in inches

- **Enumerated list:** A finite, known list of results that can't be used to perform mathematical functionality.

Examples: Loot tables, Magic the Gathering cards, Settlers of Catan tiles, colors on a spinner

- **Non-enumerated result:** An infinite, unknown result that can't be used to perform mathematical functionality.

Examples: Curtain prizes on a game show, random unknown loot box, thrift store grab bag

Measure of Random Result

Random result data is produced by random events. This is used to determine what a designer can do with the result and how predictable the result will be. Discrete results are far more predictable than nonlinear, for example.

- **Discrete:** Discrete random results have finite and listable results.

Examples: A coin toss, die roll, or roulette wheel spin

- **Continuous linear:** Continuous linear random results have infinitely variable but measurable results.

Examples: Golf shot distance, prediction of the height of a child when grown, final score of a basketball game, tomorrow's exact temperature

- **Continuous nonlinear:** Continuous nonlinear random results have infinitely variable and nonquantifiable results.

Examples: Curtain prizes on a game show, random unknown loot box, grab bag

Types of Randomness

For each random event in a game, there can be one or more methods of creating randomness. Each of these types of randomness informs how the

game is played and how the designer can track the output. For example, in Pure random, the chance should not change based on players, but in the performance-based type, it is heavily dependent on the players in the game.

- **Pure chance:** Designed specifically to be unpredictable. All players have equal probability of all results.
Examples: Dice, lottery machine, cards, spinners, lava lamps, scattered coins
- **Performance variation:** Players have some amount of influence on the outcome but not total control.
Examples: Darts, pool, baseball batting, golf, bocce, parimutuel betting
- **Obscured knowledge or incomplete information:** One or more people possess knowledge that the other player does not have. This may or may not be chosen by another player.
Examples: Guess the number, rock paper scissors, poker

Probability Distribution

There are many different variations of probability distribution. For games, the following are some of the most popular probability distributions:

- **Equal distribution:** Every outcome has the same probability of occurrence.
Examples: Deck of cards, single die, roulette wheel, lottery numbers
- **Bell curve distribution:** When random results are added together, a bell curve distribution results.
Examples: 2D6 and other multidimensional probabilities
- **Exponential:** Many results are possible on one side, and increasingly few are possible on the other.
Examples: “Or greater” dice probability, pounds of weight lifted by an athlete, score on *Donkey Kong*
- **Uneven distribution:** Results vary widely in the distribution of probability among events.

Examples: Loot tables, curtain prizes, carnival game prizes

Outcome Dependency

Outcome dependency is an attribute that indicates whether the current random event is affected or limited by other events. Based on the type of outcome dependency the event has, the type of probability calculation you will need to evaluate it will change. For example, independent is linear, and dependent is a form of bell curve.

- **Independent:** An independent random event stands on its own and has an equal chance to each other event.

Examples: Dice roll, roulette wheel spin, throw of a single dart

- **Dependent:** A dependent result is affected by past events and can affect future events.

Examples: Drawing a card from an unshuffled deck, hitting a scoring space in darts

- **Mutually exclusive:** When one result occurs, it makes other results impossible.

Examples: Drawing the ace of spades in a series of card draws, choosing a path in a maze

Mapping Probability Examples

By using the attributes listed above, you can describe nearly any random event that happens in a game. By sampling some of the events that most commonly happen in a game, you can get better insights into how that game works and what the character of the game will be to the players who play it. To graph an event, you first need to pick a very specific event. The smaller the event you pick, the more specific information you will be able to observe in it.

Let's look at a backgammon example. In the scenario illustrated in [Figure 19.54](#), it is red's turn.

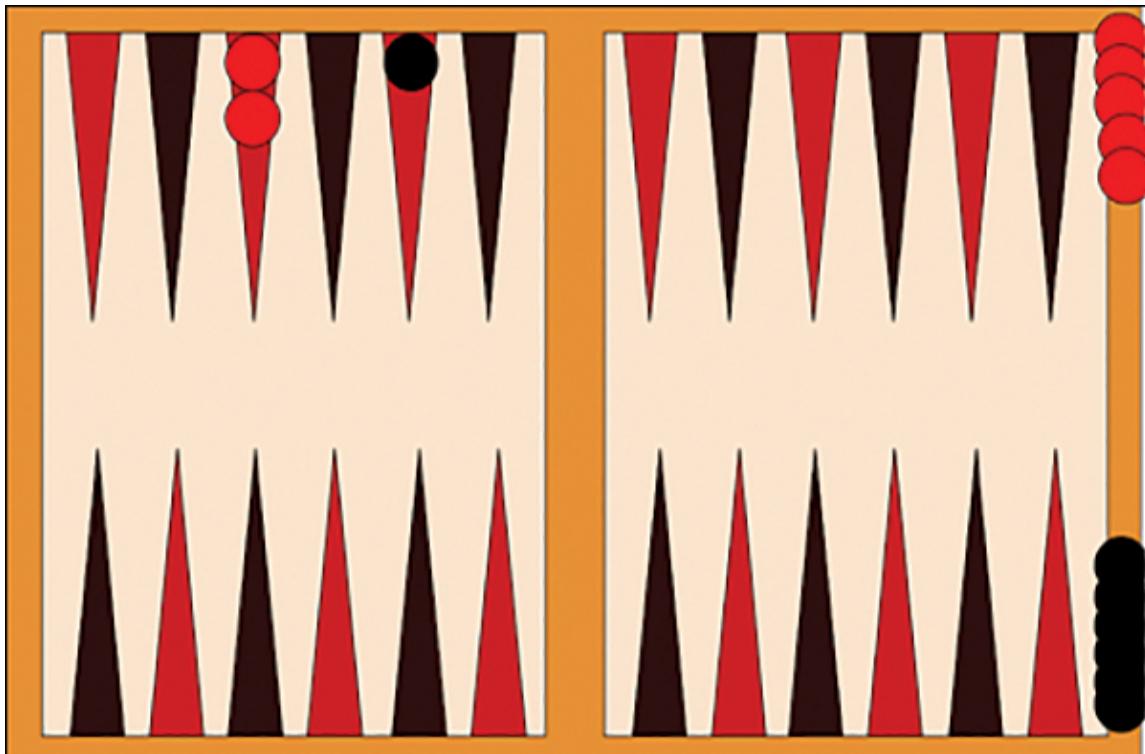


Figure 19.54 Backgammon move

On their turn, red will roll a pair of dice to determine movement points available. You could describe this event as follows:

- **Name:** Backgammon move
- **Computation use:** Numeric
- **Measure of random result:** Discrete
- **Type of randomness:** Pure chance
- **Probability distribution:** Bell curve
- **Outcome dependency:** Independent

Now let's look at a completely different type of probability event: a carnival game where the player throws a dart at a wall of balloons (see [Figure 19.55](#)). The dotted balloons are empty, but the other colored balloons contain small toys or other prizes.

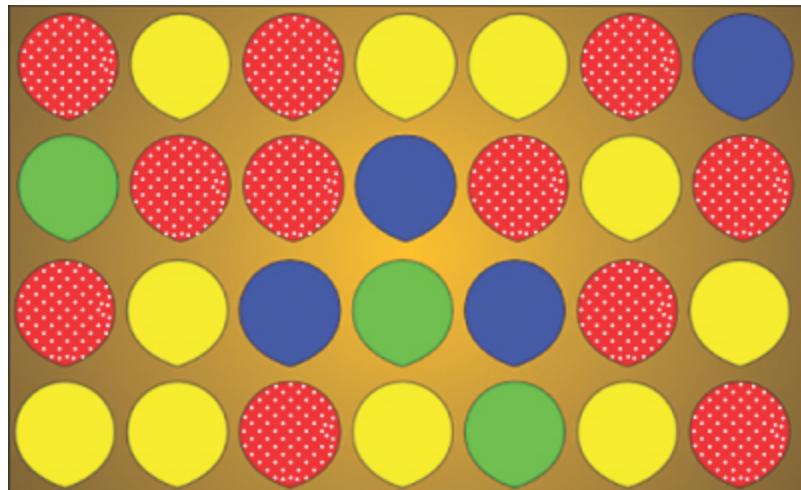


Figure 19.55 Balloon pop game

You could describe this event as follows:

- **Name:** Balloon pop, first dart
- **Computation use:** Non-enumerated result
- **Measure of random result:** Discrete
- **Type of randomness:** Performance variation
- **Probability distribution:** Uneven distribution
- **Outcome dependency:** Independent

This list of attributes is quite different from the list for backgammon, although the games do have two attributes in common. By comparing the two lists, you can see some characteristic differences between the two games. In backgammon, players have no control over the dice but a high level of control over what they do with the results of the dice. Players can also make predictions about the dice based on bell curve probability. With the balloon pop, on the other hand, the player has some agency in choosing which target to aim for but not complete control. Once a target is hit, the player has absolutely no agency in what to do with the result. These differences cause two different characteristics in the game. In backgammon, players are not as interested in what the individual dice rolls are as they are in playing the odds of what they can be and reacting appropriately to the

results. In the balloon pop game, players are most interested in trying to target a desired balloon, knowing that once the dart leaves their hand, the rest of the game is out of their control.

While working on a game, you need to think about the kinds of events and attributes you want to happen in your game. It may be a useful exercise to examine the random events and attributes in similar games or in a competitor's game and then use that information to inform the design of your game as you build it. Think about the events and attributes you see in other games and what feelings players, especially testers, have associated with them.

As a game project develops, you can come back to listing events and attributes if a problem arises. If playtesters are getting frustrated with the rewards from a random loot table, for example, you can list the attributes of the event to see if any aspects of the event might be changed to fix the problem. With some creative thinking, a designer can change individual aspects of an event to get dramatically different feelings from very similar events.

Let's look at one more example. What happens if you combine the previous two examples by mixing and matching their lists? Think about what that would mean for a game with these attributes:

- **Name:** Hybrid graph
- **Computation use:** Numeric
- **Measure of random result:** Discrete
- **Type of randomness:** Performance variation
- **Probability distribution:** Uneven distribution
- **Outcome dependency:** Independent

What might this list of attributes look like as a game mechanic? It could be manifested in several ways, which is great for designers. By simply thinking about the relationships of the attributes, you are likely to encounter new ideas. In this case, one possible outcome could be a backgammon type of game, but instead of rolling dice, the players throw two darts at a board

with squares representing dice results. It might look something like the game in Figure 19.56.

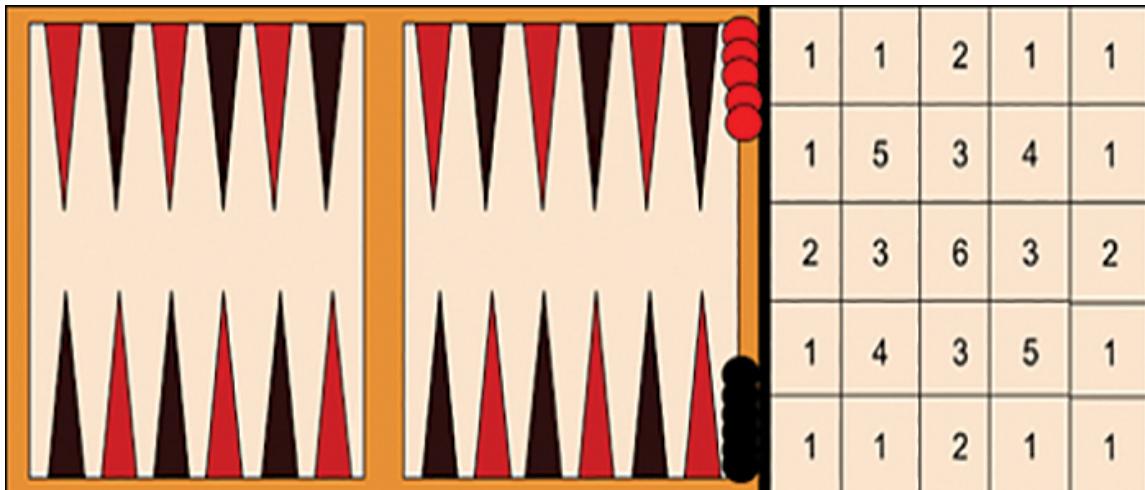


Figure 19.56 Hybrid backgammon/darts game

This random event now combines aspects of the two previous games to create something completely new and very different in feel from either of the other games.

Measuring Luck in a Game

For complex games, unfortunately there is no single algorithm you can use to determine whether a game is pure luck, luck dominant, luck influenced, or pure ability. In fact, there have been several famous lawsuits pertaining to this exact topic. Pinball was once made illegal because it was considered gambling. Poker has been argued as both a luck-based game and a skill-based game in courts of law. There are many factors involved with complex games that may make a game appear to be more or less luck based than it is, but you can look for signs that a game—or even part of a game—relies on luck.

In terms of the influence of luck, games fit into four broad categories:

- **Pure luck:** Player decisions have little or no impact on the activity. At this point, the activity does not meet the criteria for being a game.

Examples: Candy Land, war (card game), roulette, lottery

- **Luck dominant:** Player decisions impact the game, but the outcome of the game can't be predicted based on the ability of the player. Better players win slightly more often than worse players in the long run, but there is no way to accurately predict a winner of an individual session based on player ability.

Examples: Bingo, blackjack, video poker, solitaire

- **Luck influenced:** Player decisions have a major impact on the game. Players with more ability at the game win more frequently, but players who are close in ability will have less predictable results when paired against each other. This variable also takes into account uncertainty that is not random. For example, a player attempting to hit a golf ball into the hole may miss unpredictably but not randomly.

Examples: Royal Game of Ur, backgammon, poker, golf, Tetris, pool

- **Pure ability:** There is no inherent randomness built into the game. The player with the better ability at the game or the player who is having a better day will be more likely to win. In other words, the outcome of the game is not affected by any mechanism in the game that is outside of the players' full control.

Example: Chess, Go, 100-meter dash

Testing for Pure Luck

You can test for pure luck by carefully reading and understanding the rules of the game. Does the player have agency? is the only question you need to answer. If there are no mechanics that give the player agency to affect the outcome of the game, then the outcome is a matter of pure luck, and the activity ceases to be a game (refer to [Chapter 1, “Games and Players: Defined”](#)). Another way to think about it: Does there need to be a player? Candy Land, for example, is a four-player game, but one player can handle all the turns for all the players, and exactly the same outcome will result as if the four players all perform their own turns.

Testing for Luck Dominant

You can do some simple tests to determine if a game is luck dominant. In a game that is luck influenced or even luck dominant, part of the outcome of the game is determined by luck, and another part of the game is determined by player ability. The easiest test to perform to determine whether a game is mostly based on luck is to purposefully try to lose. In a single-player game, this is straightforward. The player should make what they think is the worst possible decision at every opportunity. In a multiplayer game, one of the players is designated the winner, and the others as the losers. The designated losers attempt to purposely lose at the game. In games with some degree of skill, the outcome of the game should be clear. For example, imagine playing a game of chess in which one side is trying to lose and the other side is trying to win. It would be a fast and decisive game every single time. Even games that are heavily luck influenced—like The Royal Game of Ur, backgammon, and Parcheesi—become absolutely lopsided blowouts under this test.

If players are unable to consistently cause a predetermined losing player to lose, the game can be considered luck dominant. Blackjack is a good example of a game that is luck dominant. There is skill involved in the game, and it is certainly possible to make better or worse decisions. A player who tries to lose certainly will, but for those who play using perfect strategy, the outcome is still far from predictable. In blackjack, even when playing with perfect strategy, a player can still expect to lose slightly more often than they win. This game is considered luck dominant.

Testing for Luck Influenced

Luck influenced games are by far the largest category of modern games. Even many old games of chance are still more skill than luck but heavily luck influenced. Backgammon and poker are excellent examples. In these games, large levels of disparity in player ability are obvious. For example, a poker pro will absolutely dominate children playing poker nearly all the time. A professional backgammon player will defeat a novice with a very high percentage of wins.

With luck-influenced games, an important factor to look at is how close in player ability players need to be before the outcome of the game is unpredictable. In chess, players need to be almost identical in skill level for

the outcome to be unpredictable. With backgammon, players of similar skill levels have unpredictable outcomes. In most cart-racing video games, players of a fairly large spectrum of ability will produce unpredictable outcomes. Individual instances of luck-influenced games don't provide usable results. Instead, with games like this, trends over a large number of instances start to provide a clearer picture of player ability.

Getting the measure of player quality and win percentage is not particularly easy. Because it takes many iterations of games with players of varying skill levels all playing against each other to determine long-term trends, you need a rather big system. This is where player ranking systems become useful. Systems like Elo rating, TrueSkill, Tournaments, and player ladders can be used to assess and compare players to determine ability.

Adjusting the Influence of Luck

Once you have done testing on your game to determine what kinds of randomness or luck-based mechanics influence it and to what level the game is randomly determined, you may find that you want to adjust the amount of influence luck has. Unfortunately, it is not as straightforward as adding or subtracting some dice rolls. You need to determine how much impact luck should have on the game. As a general rule, the more luck involved in a game, the more inclusive the game is. This is why games with heavy randomness are perennial family favorites. When playing spades, dominoes, and party-style video games, everyone knows going in that they have a chance to win. This means that seasoned experts can play with relative novices. In a family setting, this means grandma who has been playing the game for 70 years and knows the rules by heart can play with the grandkids who just learned the rules, and they can all feel competitive in the game.

Two major factors to think about when adjusting luck in a game are the frequency of random events and the amount of impact that individual random events have. Intuitively, it may seem like a game with a large number of random events would be more random, but this is not always the case. Let's look at a modified version of backgammon as an example.

In the standard rules of backgammon, each player rolls 2D6 before their turn and then may move pieces that number of spaces. All movement in the game is randomly determined, so it would seem that the game would be dominated by luck. We know, though, that this is not the case with backgammon. If you want to test this for yourself, look online for a backgammon simulator with advanced AI. Play against the best AI the game has and notice how incredibly good the computer is at the game. Even very seasoned backgammon players have a very difficult time beating the AI. How is this so? Based on the list of backgammon random event attributes you saw earlier in this chapter, you might guess that because there are so many random events, the results of a series of games would eventually average out to roughly 50% won against the AI. It works out like this precisely because there are so many random events, each having only a small impact on the game, and the game allows a high degree of player choice in using the random events. Because there are numerous random events, the curve of probability for the game tends to even out. Any given player will get mostly middle rolls, around 7, with just a few at the extremes, 2 and 12. This means that if the players are getting roughly the same distribution of rolls each game, player skill is actually having an impact on the game.

To see this more clearly, let's change the rules of backgammon a bit and imagine how the game would play out. In the standard rules, a player starts with 15 pieces and attempts to bring all of them home, moving based on the values of rolled dice. In our changed version, let's say that there are no more random rolls. All the movement has been predetermined and is visible to both players in a list that is equal for both players. The list of example moves would look something like this:

1. 3, 4

2. 1, 6

3. 4, 4

4. 2, 5

So you have now eliminated the traditional dice rolls from the game. However, you are going to add in a single roll at the very beginning. At the start of the game, each player rolls 2D6. The player may then remove from the board a number of pieces equal to that roll and place them home instantly. Think about what that would mean for the game. If the first player rolled a 12 and the second rolled a 2, it would be virtually impossible for the second player to win. If you went through many games like this, it would become clear that the die roll at the beginning is all that actually matters in the game. Even though there is much less rolling of dice in this new version, it is actually driven by luck far more than the standard rules for backgammon.

The other factor that determines luck is the amount of impact that each random event has on the game. Let's look at an extreme example of modified darts to see how it differs in terms of luck from a regular game of darts.

In our changed version of darts, on every turn, before a player throws their darts, they roll a 1D20. The result of the roll is the active space. Players only score points by hitting the active space for that turn. The game goes for 20 turns, scores are added, and a winner is determined. In this game, there is a lot of dice rolling. A die is rolled each turn for 20 turns, but is the game luck dominant? No. If anything, the game has no more luck in it than a standard game of darts. The player who can hit the target more often will win more often, regardless of what dice rolls occur. In this case, we would say that even though there are dice rolls, they have little impact on the game. This is in contrast to the change in backgammon in the previous example. In that example, there is only one roll of the dice, but it had an extremely large impact.

All this means that when you modify a game to make it more or less luck influenced, you need to think about not just the number of random events but how much impact those events have. If you find that your game is too luck dominant, you shouldn't just remove some random events. Instead, you should look at how the random events you have add up and average out, and you should consider what impact each one has on the game. You may actually find that adding more random events makes the game less luck dominant overall.

Chaos Factor

Even in games of pure ability, there is no way to 100% guarantee the outcome. If the outcome of a game were certain, there would be little point in actually playing the game. Players can have a bad day. Exceptional streaks of probability do happen. External factors can be involved. Bugs in video games can have a dramatic effect. You should therefore never run a test and make a sweeping assumption based on any given set of data. As much as this book has tried to guide you to making logical, fact-based decisions, when it comes to the feel of a game, human intuition will always be required.

Further Steps

After completing this chapter, you should take some time to practice in the real world with the concepts covered here. Try these exercises to further explore the concepts of probability in games.

- List some real-world game examples of probabilities you can calculate. Loot tables, chance to hit, and so on are good candidates. Use the graphing system explained in this chapter to graph each of these events. What patterns do you see emerge?
- Pick a game that involves some random events and change the way the probability works. For example, how would backgammon work with cards instead of dice? How would it work with 1D12 instead of 2D6?
- In your own games, do some experiments with multiple random event attributes. Be sure to test thoroughly as initial results could easily be probability outliers.

Chapter 20

Next Steps

Now that you are at the end of this book, where should you go from here? This book is designed to be an introduction to many different subjects but not a definitive resource for any individual topic. I put a lot of effort into covering as many subjects as possible and showing you the basics, but the goal of the book was not to cover each subject in great depth. Given that, it's important that you see reading this book as the first step in your journey into game system design. This brief chapter describes a few next steps that you should take to continue the journey.

Practice

Many of the concepts in this book are more than basic facts that can be memorized and understood. The techniques described in this book need to be actively performed and practiced multiple times to be fully understood. I highly recommend that you go through each of the chapters again and actively practice using the concepts described. Then try taking each concept further. For example, experiment with modifying the exponential growth formula to see what happens. Try combining multiple concepts covered in the book, such as graphing and communication noise, and see what happens. By taking the concepts from the book further, you can be sure you actually understand the concepts and are not just able to copy what you have read.

Analyze Existing Games

What are your favorite computer and analog games? What games that you love make heavy use of game systems? Go back and play those games with your newly gained knowledge and see what aspects of the games you notice now that you never have before. Start analyzing your favorite games as if

they were your own. By analyzing complete games that have already been through the development and testing process, you can start to see the correlation between how the game feels as you play it and how the systems were created to induce that feeling.

Play New Games

Another valuable exercise is to play new games, ideally in genres that you don't know well, and observe how you learn the systems. Take some time to record your feelings before you play a game for the first time and then after the first play and again after multiple plays. Notice how the systems go from being completely mysterious to being systems that you understand. To take this exercise a step further, find a game that you have never played and do some data analysis on it before playing it even a single time. Guess how the game should feel, based on what you see in the data. Finally, play the game and note how the game actually plays and feels.

Modify Existing Games

There are a few published games (such as the RPG The Battle for Wesnoth) whose data is in a state that can be modified. Seek out these games and study them thoroughly. When you understand their systems, try to modify them for a specific result. Doing this kind of exercise is a very fast and easy way to learn about modifying systems without the required overhead of building your own game. The existing game gives you a stable base to start with. Its code and game engine are fully functioning, and when you edit them, you can see what changes your edits cause. Once you are comfortable changing the data of an existing game in simple ways, try causing strange and experimental results. As with testing, you often can't know how far to take a change until you have gone too far. An existing game gives you a very easy way to do these kinds of experiments.

Work on Your Game

If you have completed this book, I presume that you are interested in creating a game or some systems within a game. There is no such thing as starting to work on a game too early. Take what you have learned in this book and apply it to the game you are working on. Whether your game is a

video game in full production or a board game that is currently just an idea, you can start to apply the techniques outlined here to move your game forward.

Keep Learning

This book provides a basic introduction to game system design, but there is much more you can learn. From this point on, it's likely that you will need to study a number of subjects in more depth to really learn how they work. For example, you might seek out one of the many good books on using spreadsheets at a more advanced level. Similarly, you might look for books on game theory, psychology, and probability. By doing Internet searches, you can find much more and deeper information on each of the subjects introduced in this book.

Index

Numerics

2D6 “or higher” cumulative probability, calculating, [309-310](#)
3D modeling tools, [46-47](#)

A

absolute referencing, [75](#)
acquiescence bias, [32](#)
adjusting influence of luck, [336-338](#)
advertising, [16](#)
 social media, [16-17](#)
 word of mouth, [16](#)
ampersand (&), [59-60](#)
analyzing game data, [229-230](#)
 canaries, [241-244](#)
 comparison analysis, [240-241](#)
 existing games, [342](#)
 next-level deep analysis, [238-240](#)
 overview analysis, [230-238](#)
 practicing, [240](#)
animators, [26](#)
arguments
 grouping, [90](#)
 in more complex functions, [93-95](#)
asking questions, [36-37](#). *See also* defining a question for data analysis;
scientific method; writing a good question
acquiescence bias, [32](#)
bad question example, [40](#)

good question example, 39
for help with a problem, 36
theoretical, 32
 determining numbers to use, 34
 form an explanatory hypothesis, 35
 test the hypothesis, 35

assistants, 28

attribute(s), 158

 bank, 138

brainstorming, 135-136

comparison analysis, 240-241

damage per minute, 174

data and, 44

defining, 139-141

DPS (damage per second), 173, 174, 175

grouping, 141-143

intertwined, 175-176

listing, 134-135

mechanics and, 134

naming conventions, 183-184

next-level deep analysis, 238-240

numbers

accuracy and, 165-167

determining granularity, 158

for easier calculations, 160-161

fractions and decimals, 163

for granularity, 161-162

individual balance, 201

range balancing, 196-203

relating to probability, 158-159

relating to real-world measurements, 159-160

systemic balance, 201

tension trick, 163-165
very large, 162-163
overview analysis, 230-238
in past games, researching, 137-138
placing in spreadsheets, 148-149
of random events
 computation use, 329-330
 outcome dependency, 331
 probability distribution, 331
 types of randomness, 330
real-world, researching, 137
researching, 136-137
weights, 170-172
 balance and, 172-173
 calculating based on probability, 325-327
audience, 6
audio engineers, 27
AVERAGE function, 97

B

bad storming, 126
balance, 258
 handshake formula, 188-194
 applications, 193-194
 possibility grid, 189-192
 importance of, 258-259
 indicators of, 259-261
 judged contests and, 261-262
 prototyping and, 263
 reciprocity and, 287-288
 testing, 267-268
 weighted attributes and, 172-173

basic exponential growth formula, 218-220
building blocks, 220-226
tweaking, 226-227

Battle for Wesnoth, The, target audience profile, 20

Bejeweled, target audience profile, 20-21

beta/postlaunch telemetry testing, 273
data hooks, 273-274
examples, 274-275

binary searching, 176
boss fight example, 178-179
jump distance example, 179
lacking a viable range, 179-180
maximum number of guesses, 178
requirements, 176

blue-sky brainstorming, 136

brainstorming
avoiding criticism, 124
blue-sky, 136
capturing the creativity, 125
don't accept the first answer, 123-124
gathering the troops, 122
goals and, 121-122
keeping expectations reasonable, 125
keeping on topic, 124-125
listing attributes, 135-136
methods to force creativity, 126
bad storming, 126
building blocks, 127
future past, 127
halfway between method, 128-129
iterative stepping, 127-128
jokes, 126

opposite of method, 129
random connections, 130
stream of consciousness writing, 130
percolating, 125
time and, 123
breaking your data, 261
bug testing, 268
bug-tracking software, 49

C

calculating probability
2D6 “or higher” cumulative, 309-310
compound, 301-309
dependent event, 318-321
of doubles, 310-311
enumerated probability with an even distribution, 321-322
enumerated probability with an uneven distribution, 322-325
multi-dimensional, 316-318
mutually exclusive event, 321
one-dimensional even-distribution, 293-299
one-dimensional uneven-distribution, 299-300
of a series of single events, 311-316
canaries, 241-244
casual gamers, 7
cells, 54
 address, 54-55
 formula bar, 55-56
 references, 146-147
 value, 55
chaos factor, 338
character artists, 26
chess, target audience profile, 18

choosing, functions, 106-107
columns, 60-61, 148-149
 avoiding unnecessary, 151-152
COMBIN function, 194
communication, 279
 language and, 281
 noise and, 284-286
 reciprocity, 286
 balance and, 287-288
 overstepping bounds, 286-287
 reward expectations, 288-289
 shallow relationship, 287
 word meanings and, 281-284
comparison analysis, 240-241
compound probability, calculating, 301-309
computation use, 329-330
concept artists, 26
conversations, games as, 280-281
coordinators, 28
core management team, 24
COUNT function, 100, 233
COUNTA function, 100
COUNTIF function, 94-95, 101, 238, 304-305
COUNTUNIQUE function, 100
crafting, prototypes, 263-264
creating, data fulcrums, 204
creativity
 brainstorming
 avoiding criticism, 124
 bad storming, 126
 blue-sky, 136
 building blocks, 127

don't accept the first answer, 123-124
future past, 127
gathering the troops, 122
goals and, 121-122
halfway between method, 128-129
iterative stepping, 127-128
jokes and, 126
keeping expectations reasonable, 125
keeping on topic, 124-125
listing attributes, 135-136
opposite of method, 129
percolating, 125
random connections, 130
stream of consciousness writing, 130
time and, 123
capturing, 125
coming up with ideas, 119-120
idea buffet, 120-121
methods to force, 126
criticism, brainstorming and, 124
cross-feeding, 254-255
cross-testing, fulcrums, 208-209

D

damage per minute, 174
converting to DPS (damage per second), 174
data, 44
labelling, 147-148
validating, 148
validation, 80-81
data designer, 29
data fulcrums, 203. *See also* hierarchical design

creating, 204
cross-testing, 208-209
for data creation, 206-208
locking, 206
progression, 209-210
testing, 204-205
databases, 48-49
DDA (dynamic difficulty adjustment), 251-252
defining
 attributes, 139-141
 questions for data analysis, 35-36
dependent event probability, calculating, 318-321
difficulty adjustment, 246
 balancing combinations, 255
 cross-feeding, 254-255
 dynamic, 251-252
 flat balancing, 246-247
 layered, 253-254
 negative feedback loops, 249-251
 positive feedback loops, 247-248
diminishing returns, 215-216
documentation tools, 45
doubles, calculating probability of, 310-311
DPS (damage per second), 173, 174, 175
 calculating, 174

E

enumerated probability
 with an even distribution, calculating, 321-322
 with an uneven distribution, calculating, 322-325
environmental artists, 26
equal sign (=), 56-57

exponential growth, 215, 216, 217-218
basic exponential growth formula, 218-220
building blocks, 220-226
tweaking, 226-227
iterations and, 227

F

filters, spreadsheet, 77-79
FIND function, 102-103
flat balancing, 246-247
flowchart tools, 47-48
formfill, 71-76
formula bar, 55-56
equal sign (=), 56-57
formulas. *See also* functions
ampersand (&), 59-60
basic exponential growth, 218-220
building blocks, 220-226
tweaking, 226-227
mathematical symbols, 59
parentheses, 58
fulcrums, 203-204. *See also* hierarchical design
creating, 204
cross-testing, 208-209
for data creation, 206-208
locking, 206
progression, 209-210
testing, 204-205
functions, 89, 106
arguments, 89
grouping, 90
AVERAGE, 97

choosing, 106-107
COMBIN, 194
complex, 93-94
COUNT, 100, 233
COUNTA, 100
COUNTIF, 94-95, 101, 238, 304-305
COUNTUNIQUE, 100
FIND, 102-103
IF, 101
LEN, 100
MAX, 99
MEDIAN, 97-98
MID, 103
MIN, 99
MODE, 98, 234
NOW, 103-104
RAND, 104
RANDBETWEEN, 105
RANK, 99
ROUND, 105
SORT, 238
structure, 90-93
SUM, 91-93, 96-97
syntax, 93
UNIQUE, 238
VLOOKUP, 102
future past method, 127

G

Galaga, 134
target audience profile, 18-19
game development, 24

art

animation, 26
character art, 26
concept art, 26
environmental art, 26
interface art, 26
technical art, 27

core management team, 24

design, 28

data designer, 29
game system designer, 28-29
level designer, 28
scripter, 29
technical designer, 29

engineering

audio engineer, 27
gameplay engineer, 27
graphics engineer, 27
network engineer, 27
scripter, 27
tools engineer, 27

game developer, 24

hierarchical design, 210-211

advantages of, 212-213
exponential charts and, 227-228
starting the hierarchy, 211-212

lead designer, 25

lead engineer, 25

lead sound designer, 25

narrative designer, 30

producer, 25

production, 28

assistants, 28
coordinators, 28
management, 28
QA team, 29-30
scientific method and, 32
 analyze the data, 35
 define a question for playtesting, 32-34
 form an explanatory hypothesis, 35
 gather information and resources, 34
 interpret the data, draw conclusions, and publish results, 35
 retest, 35
 test the hypothesis, 35
sound team, 29
tools
 3D modeling, 46-47
 bug-tracking software, 49
 databases, 48-49
 documentation, 45
 flowchart, 47-48
 game engines, 49-50
 image editing, 45-46
vision holder, 24-25
game engines, 49-50
game mechanic(s), 112-114
 attributes and, 134
 putting together, 115-116
 running, 115
 sticks, 115
 teamwork, 115
 throwing, 114-115
game system design(er), 28-29, 52, 116-117. *See also* asking questions; functions; spreadsheets; tools

attribute weights, 170-172
balance, 258
 attribute weights and, 172-173
 importance of, 258-259
 indicators of, 259-261
 judged contests and, 261-262
 prototyping and, 263
binary searching, 176-179
 boss fight example, 178-179
 jump distance example, 179
 lacking a viable range, 179-180
 maximum number of guesses, 178
 requirements, 176
damage per minute, 174
 converting to DPS (damage per second), 174
DPS (damage per second), 173, 174, 175
 calculating, 174
inclusivity and, 13
intertwined attributes, 175-176
naming conventions, 180-185
 attribute(s) and, 183-184
 object iterations, 185, 186
 spaces and, 183
 special case words, 187-188
 using “new”, 185-186
gameplay engineers, 27
games, 2, 342-343
 adjusting influence of luck, 336-338
 analyzing, 342
 attributes, 5
 finite sessions, 4
 intrinsic rewards, 4-5

people can opt out, 4
players have an on the outcome, 3-4
rules, 2-3
chaos factor, 338
as conversations, 280-281
lead artist, 25
luck dominant, 335-336
luck-influenced, 336
modifying, 342-343
new, 342
puzzles and, 5
session time, 10
stories, 116-117
total time, 10
goals, brainstorming and, 121-122
Google Sheets, 62. *See also spreadsheets*
graphics engineers, 27
grouping, attributes, 141-143

H

halfway between method, 128-129
handshake formula, 188-194
 applications, 193-194
 possibility grid, 189-192
hardcore gamers, 7
hiding, spreadsheet data, 65-66
hierarchical design, 210-211
 advantages of, 212-213
 exponential charts and, 227-228
 starting the hierarchy, 211-212

I

idea buffet, 120-121. *See also* creativity
IF function, 101
image editing tools, 45-46
imperfect information probability, 327-328
inclusivity, 13
interface artists, 26
intertwined attributes, 175-176
introduction sheet, 153-154
iterative stepping, 127-128

J-K

jokes, creativity and, 126
judged contests, balancing, 261-262
jumping, prototypes, 264-265

L

labelling data, 147-148
language, 281
layered difficulty adjustment, 253-254
lead artist, 25
lead designer, 25
lead engineer, 25
lead sound designer, 25
LEN function, 100
level designer, 28
leveling up, prototypes, 264
linear growth, 216-217. *See also* exponential growth
list validation, 84
listing, attributes, 134-135
locking, fulcrums, 206
luck
adjusting influence of, 336-338

dominant, testing for, 335-336
influence, 336
measuring, 334-335
pure, testing for, 335

M

macrosystem difficulty adjustment. *See* difficulty adjustment
mapping probability, 329
Mario Kart, target audience profile, 19
mathematical symbols, 59
MAX function, 99
measuring luck, 334-335
mechanic(s), 112-114
 attributes and, 134
 putting together, 115-116
 running, 115
 sticks, 115
 teamwork, 115
 throwing, 114-115
MEDIAN function, 97-98
MID function, 103
MIN function, 99
minimum viable testing, 266-267
MODE function, 98, 234
multi-dimensional probability, 316-318
mutually exclusive event probability, calculating, 321

N

named ranges, 84-87
naming conventions, 180-185
 object iterations, 185, 186-187
 spaces and, 183

- special case words
 - date or time*, 188
 - “*deleteme*”, 187
 - “*deprecated*”, 187-188
 - “*test*”, 187
 - using “new”, 185-186
 - negative feedback loops, 249-251
 - network engineers, 27
 - next-level deep analysis, 238-240
 - noncompetitive games, 11-12
 - NOW function, 103-104
-
- ## O
- objects, 152
 - categories, 182
 - data fulcrums, 203
 - creating*, 204
 - cross-testing*, 208-209
 - for data creation*, 206-208
 - locking*, 206
 - progression*, 209-210
 - testing*, 204-205
 - naming conventions, 180-187
 - special case words*, 187-188
 - naming iterations, 185
 - placing in spreadsheets, 148-149
 - one-dimensional even-distribution probability, calculating, 293-299
 - one-dimensional uneven-distribution probability, calculating, 299-300
 - one-time purchase(s), 14
 - opposite of method, 129
 - outcome dependency, 331
 - output/visualization sheet, 154-155

overview analysis, 230-238

P

parentheses, 58

payment, 13

advertising and, 16

expansions, 14-15

microtransactions, 15-16

one-time purchase, 14

other forms of value, 16

content creation, 17

market numbers, 17

player interaction, 17

popularity contests, 17

ranking sites, 17

social media, 16-17

word of mouth, 16

perception of probability, 328

platforms, 12

players

attributes, 6

age, 6-7

competitiveness, 11-12

desired time investment, 10

gender, 7

genre/art/setting/narrative preference, 13

interest in challenge, 9-10

pace preference, 11

platform preference, 12

skill level, 12-13

tolerance for learning rules, 7-9

value gained from, 13

playtesting, 265-266
balance testing, 267-268
beta/postlaunch telemetry testing, 273
data hooks, 273-274
examples, 274-275
bug testing, 268
defining a question for, 32-34
minimum viable testing, 266-267
user testing, 269-273
positive feedback loops, 247-248
possibility grid, 189-192
practicing data analysis, 240
probability, 291-292. *See also calculating; luck*
2D6 “or higher” cumulative, 309-310
calculating attribute weights based on, 325-327
compound, 301-309
dependent event, 318-321
dice, 293
distribution, 331
of doubles, 310-311
enumerated probability with an even distribution, 321-322
enumerated probability with an uneven distribution, 322-325
imperfect information, 327-328
mapping, 329, 331-334
multi-dimensional, 316-318
mutually exclusive event, 321
notation, 292-293
one-dimensional even-distribution, 293-299
one-dimensional uneven-distribution, 299-300
perception of, 328
relating attribute numbers to, 158-159
of a series of single events, 311-316

uncertainty and, 328-329
producers, 25
prototypes, 263
 crafting, 263-264
 jumping, 264-265
 leveling up, 264
 macrosystems, 265
pure luck, testing for, 335
puzzles, 5

Q-R

QA team, 29-30
questions, writing, 37-40
quotation marks, 58-59
RAND function, 104
RANDBETWEEN function, 105
random connections, brainstorming and, 130
random events
 computation use, 329-330
 outcome dependency, 331
 probability distribution, 331
 types of randomness, 330
randomness, 292
range balancing, 196-203
 individual balance, 201
 systemic balance, 201
RANK function, 99
real-world attributes, researching, 137
reciprocity, 286
 balance and, 287-288
 overstepping bounds, 286-287
 reward expectations, 288-289

- shallow relationship, 287
- ref sheet, 152-153
- relative referencing, 73, 75
- researching, attributes, 136-137
 - in past games, 137-138
 - real-world, 137
- rewards, 288-289
- ROUND function, 105
- rows, 60-61, 148-149
 - avoiding unnecessary, 151-152
- rules, 2-3
 - data and, 44
 - tolerance for learning, 7-9
- running, 115

S

- scientific method
 - analyze the data, 35
 - define a question for playtesting, 32-34
 - form an explanatory hypothesis, 35
 - gather information and resources, 34
 - interpret the data, draw conclusions, and publish results, 35
 - retest, 35
 - test the hypothesis, 35
- scratch sheet, 155
- scripters, 27, 29
- series of probability events, calculating, 311-316
- session time, 10
- sheets
 - data objects and, 152
 - introduction, 153-154
 - output/visualization, 154-155

ref, 152-153
scratch, 155
skills, 12-13
social media, 16-17
solving problems, 275-276
canaries and, 241-244
come up with solutions, 277
communicate with the team, 277
document the changes, 277
eliminate variables, 277
identify the problem, 276
prototype and test, 277
SORT function, 238
sound team, 29
special case words
date or time, 188
“deleteme”, 187
“test”, 187
spreadsheets, 49, 52-53, 54, 68, 146. *See also* functions
! operator, 64-65
absolute referencing, 75
calculating probability
2D6 “or higher” cumulative, 309-310
compound, 301-309
dependent event, 318-321
of doubles, 310-311
enumerated probability with an even distribution, 321-322
enumerated probability with an uneven distribution, 322-325
multi-dimensional, 316-318
mutually exclusive event, 321
one-dimensional even-distribution, 293-299
one-dimensional uneven-distribution, 299-300

of a series of single events, 311-316

cells, 54

address, 54-55

formula bar, 55-56

value, 55

color coding, 149-150

columns and rows, 60-61

comments, 68-70

data

labelling, 147-148

validating, 148

data validation, 80-81

data validation dialog, 81-83

filters, 77-79

formfill, 71-76

freezing part of a sheet, 66-67

functions, 89, 106

AVERAGE, 97

choosing, 106-107

complex, 93-94

COUNTA, 100

COUNTIF, 94-95, 101

COUNTUNIQUE, 100

FIND, 102-103

grouping, 90

IF, 101

LEN, 100

MAX, 99

MEDIAN, 97-98

MID, 103

MIN, 99

MODE, 98

NOW, 103-104
RAND, 104
RANDBETWEEN, 105
RANK, 99
ROUND, 105
structure, 90-93
SUM, 96-97
syntax, 93
VLOOKUP, 102
hiding data, 65-66
list validation, 84
named ranges, 84-87
notes, 70-71
possibility grid, 189-192
references, 146-147
referencing a separate sheet, 64-65
relative referencing, 73, 75
sheets, 61, 152
 introduction, 153-154
 output/visualization, 154-155
 ref, 152-153
 scratch, 155
symbols
 ampersand (&), 59-60
 equal sign (=), 56-57
 mathematical, 59
 parentheses, 58
 quotation marks, 58-59
time validation, 83-84
VisiCalc, 53
workbooks, 60, 61-63
sticks, 115

stories, 116-117
stream of consciousness writing, 130
SUM function, 91-93, 96-97
surveys, acquiescence bias, 32
systems, 109-112

T

target audience
age and, 6-7
competitiveness, 11-12
desired time investment, 10
gender, 7
genre/art/setting/narrative preference, 13
interest in challenge, 9-10
pace preference, 11
platform preference, 12
profiles, 21-22
The Battle for Wesnoth, 20
Bejeweled, 20-21
chess, 18
Galaga, 18-19
Mario Kart, 19
skill level, 12-13
tolerance for learning rules, 7-9
value gained from, 13
value of, 17-18
teamwork, 115
technical artists, 27
technical designer, 29
telemetry testing, 273
 data hooks, 273-274
 examples, 274-275

testing
canaries and, 241-244
fulcrums, 204-205
handshake formula, 188-194
applications, 193-194
possibility grid, 189-192
for luck dominant, 335-336
for luck influence, 336
for pure luck, 335
throwing, 114-115
time validation, 83-84
tools. *See also spreadsheets*
3D modeling, 46-47
bug-tracking software, 49
databases, 48-49
documentation, 45
flowchart, 47-48
game engines, 49-50
image editing, 45-46
tools engineers, 27
total time, 10
toys, 5
troubleshooting. *See solving problems*

U-V

UNIQUE function, 238
user testing, 269-273
validation, 148
video game industry, 23
VisiCalc, 53
vision holder, 24-25
VLOOKUP function, 102

W-X-Y-Z

weapons

 damage per minute, [174](#)

 DPS (damage per second), [173](#), [174](#), [175](#)

weighted attributes, [170-172](#)

 balance and, [172-173](#)

 calculating based on probability, [325-327](#)

 DPS (damage per second), [173](#), [174](#), [175](#)

word meanings, [281-284](#)

word of mouth, [16](#)

workbooks, [60](#), [61-63](#)

writing a good question, [37-39](#)



Pearson | livelessons[®]

VIDEO TRAINING FOR THE **IT PROFESSIONAL**



LEARN QUICKLY

Learn a new technology in just hours. Video training can teach more in less time, and material is generally easier to absorb and remember.



WATCH AND LEARN

Instructors demonstrate concepts so you see technology in action.



TEST YOURSELF

Our Complete Video Courses offer self-assessment quizzes throughout.



CONVENIENT

Most videos are streaming with an option to download lessons for offline viewing.

Learn more, browse our store, and watch free, sample lessons at
informit.com/video

Save 50%* off the list price of video courses with discount code **VIDBOB**



*Discount code VIDBOB confers a 50% discount off the list price of eligible titles purchased on informit.com. Eligible titles include most full-course video titles. Book + eBook bundles, book/eBook + video bundles, individual video lessons, Rough Cuts, Safari Books Online, non-discountable titles, titles on promotion with our retail partners, and any title featured as eBook Deal of the Day or Video Deal of the Week is not eligible for discount. Discount may not be combined with any other offer and is not redeemable for cash. Offer subject to change.



Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

*Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community



Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Que • Sams • Peachpit Press



Contents

[Cover Page](#)

[About This eBook](#)

[Halftitle Page](#)

[Title Page](#)

[Copyright Page](#)

[Dedication Page](#)

[Contents at a Glance](#)

[Contents](#)

[Preface](#)

[1. Who This Book Is For](#)

[2. How To Use This Book](#)

[3. What This Book Covers](#)

[Acknowledgments](#)

[About the Author](#)

[Chapter 1. Games and Players: Defined](#)

[1. Defining Game](#)

[2. Finding the Target Audience for a Game: Player Attributes](#)

[3. Value Gained from Players](#)

[4. Target Audience Composite](#)

[5. What to Do with a Target Audience Profile](#)

[6. Further Steps](#)

[Chapter 2. Roles in the Game Industry](#)

[1. Core Management Team](#)

2. Team Subdisciplines

3. Further Steps

Chapter 3. Asking Questions

1. How to Ask a Theoretical Question

2. How to Ask for Help with a Problem

3. Further Steps

Chapter 4. System Design Tools

1. What Is Data?

2. Game Industry Tools

3. Further Steps

Chapter 5. Spreadsheet Basics

1. Why Spreadsheets?

2. What Is a Spreadsheet?

3. Spreadsheet Cells: The Building Blocks of Data

4. Data Containers in Spreadsheets

5. Spreadsheet Operations

6. Data Validation

7. Further Steps

Chapter 6. Spreadsheet Functions

1. Grouping Arguments

2. Function Structure

3. More Complex Functions

4. Functions for System Designers

5. How to Choose the Right Function

6. Further Steps

Chapter 7. Distilling Life into Systems

1. An Abstract Example
2. Story in Games
3. Further Steps

Chapter 8. Coming Up with Ideas

1. Idea Buffet
2. Running a Brainstorming Session
3. Methods to Force Creativity
4. Further Steps

Chapter 9. Attributes: Creating and Quantifying Life

1. Mechanics Versus Attributes
2. Listing Attributes
3. Defining an Attribute
4. Grouping Attributes
5. Further Steps

Chapter 10. Organizing Data in Spreadsheets

1. Create a Spreadsheet to Be Read by an Outsider
2. Avoid Typing Numbers
3. Label Data
4. Validate Your Data
5. Use Columns for Attributes and Rows for Objects
6. Color Coding
7. Avoid Adding Unneeded Columns or Rows or Blank Cells
8. Separate Data Objects with Sheets
9. Spreadsheet Example
10. Further Steps

Chapter 11. Attribute Numbers

1. Getting a Feel for Your Attributes
2. Determining the Granularity for Numbers
3. The Tension Trick
4. Searching for the Right Numbers
5. Further Steps

Chapter 12. System Design Foundations

1. Attribute Weights
2. DPS and Intertwined Attributes
3. Binary Searching
4. Naming Conventions
5. Naming Object Iterations
6. Using the Handshake Formula
7. Further Steps

Chapter 13. Range Balancing, Data Fulcrums, and Hierarchical Design

1. Range Balancing
2. Data Fulcrums
3. Hierarchical Design
4. Further Steps

Chapter 14. Exponential Growth and Diminishing Returns

1. Linear Growth
2. Exponential Growth
3. Further Steps

Chapter 15. Analyzing Game Data

1. Overview Analysis
2. Next-Level Deep Analysis

- 3. Practicing Data Analysis
- 4. Comparison Analysis
- 5. Canaries
- 6. Further Steps

Chapter 16. Macrosystems and Player Engagement

- 1. Macrosystem Difficulty Adjustment
- 2. Balancing Combinations
- 3. Further Steps

Chapter 17. Fine-Tuning Balance, Testing, and Problem Solving

- 1. Balance
- 2. Performing Playtests
- 3. Solving Problems
- 4. Further Steps

Chapter 18. Systems Communication and Psychology

- 1. Games as Conversations
- 2. Word Meanings
- 3. Noise
- 4. Reciprocity
- 5. Reward Expectations
- 6. Further Steps

Chapter 19. Probability

- 1. Basic Probability
- 2. Mapping Probability
- 3. Measuring Luck in a Game
- 4. Further Steps

Chapter 20. Next Steps

1. Practice
2. Analyze Existing Games
3. Play New Games
4. Modify Existing Games
5. Work on Your Game
6. Keep Learning

Index

- 1.i
- 2.ii
- 3.iii
- 4.iv
- 5.v
- 6.vi
- 7.vii
- 8.viii
- 9.ix
- 10.x
- 11.xi
- 12.xii
- 13.xiii
- 14.xiv
- 15.xv
- 16.xvi
- 17.xvii
- 18.xviii
- 19.xix

20. xx

21. xxi

22. xxii

23. xxiii

24. xxiv

25. xxv

26. xxvi

27. xxvii

28. xxviii

29. 1

30. 2

31. 3

32. 4

33. 5

34. 6

35. 7

36. 8

37. 9

38. 10

39. 11

40. 12

41. 13

42. 14

43. 15

44. 16

45. 17

46.18

47.19

48.20

49.21

50.22

51.23

52.24

53.25

54.26

55.27

56.28

57.29

58.30

59.31

60.32

61.33

62.34

63.35

64.36

65.37

66.38

67.39

68.40

69.41

70.42

71.43

72.44

73.45

74.46

75.47

76.48

77.49

78.50

79.51

80.52

81.53

82.54

83.55

84.56

85.57

86.58

87.59

88.60

89.61

90.62

91.63

92.64

93.65

94.66

95.67

96.68

97.69

98. 70

99. 71

100. 72

101. 73

102. 74

103. 75

104. 76

105. 77

106. 78

107. 79

108. 80

109. 81

110. 82

111. 83

112. 84

113. 85

114. 86

115. 87

116. 88

117. 89

118. 90

119. 91

120. 92

121. 93

122. 94

123. 95

124. 96

125. 97

126. 98

127. 99

128. 100

129. 101

130. 102

131. 103

132. 104

133. 105

134. 106

135. 107

136. 108

137. 109

138. 110

139. 111

140. 112

141. 113

142. 114

143. 115

144. 116

145. 117

146. 118

147. 119

148. 120

149. 121

150. [122](#)

151. [123](#)

152. [124](#)

153. [125](#)

154. [126](#)

155. [127](#)

156. [128](#)

157. [129](#)

158. [130](#)

159. [131](#)

160. [132](#)

161. [133](#)

162. [134](#)

163. [135](#)

164. [136](#)

165. [137](#)

166. [138](#)

167. [139](#)

168. [140](#)

169. [141](#)

170. [142](#)

171. [143](#)

172. [144](#)

173. [145](#)

174. [146](#)

175. [147](#)

176. 148

177. 149

178. 150

179. 151

180. 152

181. 153

182. 154

183. 155

184. 156

185. 157

186. 158

187. 159

188. 160

189. 161

190. 162

191. 163

192. 164

193. 165

194. 166

195. 167

196. 168

197. 169

198. 170

199. 171

200. 172

201. 173

202. [174](#)

203. [175](#)

204. [176](#)

205. [177](#)

206. [178](#)

207. [179](#)

208. [180](#)

209. [181](#)

210. [182](#)

211. [183](#)

212. [184](#)

213. [185](#)

214. [186](#)

215. [187](#)

216. [188](#)

217. [189](#)

218. [190](#)

219. [191](#)

220. [192](#)

221. [193](#)

222. [194](#)

223. [195](#)

224. [196](#)

225. [197](#)

226. [198](#)

227. [199](#)

228.200

229.201

230.202

231.203

232.204

233.205

234.206

235.207

236.208

237.209

238.210

239.211

240.212

241.213

242.214

243.215

244.216

245.217

246.218

247.219

248.220

249.221

250.222

251.223

252.224

253.225

254. 226

255. 227

256. 228

257. 229

258. 230

259. 231

260. 232

261. 233

262. 234

263. 235

264. 236

265. 237

266. 238

267. 239

268. 240

269. 241

270. 242

271. 243

272. 244

273. 245

274. 246

275. 247

276. 248

277. 249

278. 250

279. 251

280. 252

281. 253

282. 254

283. 255

284. 256

285. 257

286. 258

287. 259

288. 260

289. 261

290. 262

291. 263

292. 264

293. 265

294. 266

295. 267

296. 268

297. 269

298. 270

299. 271

300. 272

301. 273

302. 274

303. 275

304. 276

305. 277

306.278

307.279

308.280

309.281

310.282

311.283

312.284

313.285

314.286

315.287

316.288

317.289

318.290

319.291

320.292

321.293

322.294

323.295

324.296

325.297

326.298

327.299

328.300

329.301

330.302

331.303

332.304

333.305

334.306

335.307

336.308

337.309

338.310

339.311

340.312

341.313

342.314

343.315

344.316

345.317

346.318

347.319

348.320

349.321

350.322

351.323

352.324

353.325

354.326

355.327

356.328

357.329

358.330

359.331

360.332

361.333

362.334

363.335

364.336

365.337

366.338

367.339

368.340

369.341

370.342

371.343

372.344

373.345

374.346

375.347

376.348

377.349

378.350

379.351

380.352

381.353

382.354

383.355

384.356